Monitor Classification

Peter A. Buhr and Michel Fortier

Dept. of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Michael H. Coffin
EDS Research and Development, 901 Tower Drive, 1st Floor, Troy Michigan 48007-7019, U. S. A.

Abstract

One of the most natural, elegant, and efficient mechanisms for synchronization and communication, especially for systems with shared memory, is the *monitor*. Over the past twenty years many kinds of monitors have been proposed and implemented, and many modern programming languages provide some form of monitor for concurrency control. This paper presents a taxonomy of monitors that encompasses all the extant monitors and suggests others not found in the literature or in existing programming languages. It discusses the semantics and performance of the various kinds of monitors suggested by the taxonomy, and it discusses programming techniques suitable to each.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures, control structures*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, mutual exclusion, scheduling, synchronization*; Performance—*simulation*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*control primitives*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Monitors, Classification

1 Introduction

Many modern software systems consist of collections of cooperating tasks. In such systems, mechanisms for arranging exclusive access to resources, and for synchronizing and communicating among tasks, are needed. Many such mechanisms have been proposed, including semaphores [Dijkstra 1968], path expressions [Campbell and Habermann 1974], and various forms of message passing [Cheriton 1982; Gentleman 1985; Cheriton 1988]. One of the most natural, elegant, and efficient mechanisms for synchronization and communication, especially for systems with shared memory, is the *monitor*. Monitors were first proposed by Brinch Hansen [1973] and later described and extended by C.A.R. Hoare [1974]. Many programming languages—for example, Concurrent Pascal [Brinch Hansen 1975], Mesa [Mitchell *et al.* 1979], Modula [Wirth 1985], Turing [Holt and Cordy 1988], Modula-3 [Cardelli *et al.* 1988], NeWS [Gosling *et al.* 1989], Emerald [Raj *et al.* 1991] and μC++ [Buhr *et al.* 1992]—provide monitors as explicit language constructs. In addition, software entities such as operating-system kernels and device drivers have a monitor-like structure, although they may use lower-level primitives such as semaphores or locks to simulate monitors. Monitors are an important concurrency-control mechanism now, and will continue to be in the foreseeable future.

Many different kinds of monitors have been proposed and implemented, and several comparisons of various kinds of monitors have been published. Howard [1976a; 1976b] developed proof rules for five kinds of monitors and showed that they could be used to simulate one another. Andrews and Schneider [1983, p. 18–20] classify monitors as either *blocking* or *nonblocking* and describe programming techniques appropriate to each. Andrews [1991, p. 263–325] gives by far the most complete comparison to date; he discusses five kinds of monitors (based on Howard's classification), outlines a proof that they are equivalent (in the sense that they can be used to simulate one another), and discusses programming techniques appropriate to each.

This paper extends previous classification and comparison work in the following ways. First, we develop a taxonomy of monitors that includes all extant monitors and uncovers several new ones. Second, we systematically compare the various kinds of monitors from the standpoint of the programmer. We do this by developing proof rules for each kind of monitor, comparing the complexity of the proof rules, and assessing the difficulty of fulfilling proof obligations. Both full (complete) and simplified monitor proof rules are discussed. Finally, we investigate the performance of the various monitors to determine whether there are significant performance differences among them. Although many claims have been made that one kind of monitor is more efficient than the rest, we are aware of no previous study that has attempted to measure empirically the differences in performance among the various kinds of monitors. In essence, our taxonomy produced the monitors that we surveyed, and the survey covers theoretical, objective, and subjective aspects of these monitors.

We believe that this work will be of interest to designers and implementors of programming languages and concurrent systems. To make an informed judgment as to which kind of monitor to provide in a language or concurrent system, designers need a detailed comparison of the possible choices. Differences in both expressive power and efficiency must be considered. We believe this work will also be of interest to educators since the taxonomy presented explains differences among monitors in a simple and systematic way.

A general knowledge of concurrency issues is assumed throughout our discussion. In addition, familiarity with Hoare-style axiomatic semantics of sequential programs is assumed.

2 Background

Monitors are used by tasks to ensure exclusive access to resources, and for synchronizing and communicating among tasks. A monitor consists of a set of data items and a set of routines, called *entry routines*, that operate on the data items. The monitor data items can represent any resource that is shared by multiple tasks. A resource can represent a shared hardware component, such as disk drive, or a shared software component, such as a linked list or a file. In general, monitor data can be manipulated only by the set of operations defined by its entry routines; an exception might be a read-only variable. Mutual exclusion is enforced among tasks using a monitor: only one task at a time can execute a monitor-entry routine. This task is termed the *active* task. Mutual exclusion is enforced by *locking* the monitor when execution of an entry routine begins and *unlocking* it when the active task voluntarily gives up control of the monitor. If another task invokes an entry routine while the monitor is locked, the task is *blocked* until the monitor becomes unlocked.

In general, monitors are structured on the class concept [Dahl *et al.* 1970], which allows direct association of the entry routines with the shared data they manipulate. This structure allows entry-routine calls to serve as the main point at which mutual exclusion is established for the monitor data. Monitors appear in languages as either object generators or objects. In this paper, the term *monitor* always refers to a monitor object. The main difference between an object and a monitor is that no mutual exclusion is enforced during execution of an object's routines whereas it is for a monitor's entry routines.

When a monitor is used for task interaction, the interaction is indirect: a monitor is created independently of the tasks that use it and the tasks must interact with one another through it. This is in contrast to direct interaction, where tasks synchronize and communicate directly with one another (called *rendezvous*).

Tasks communicate with a monitor by passing arguments to entry-routine parameters. This approach ensures that the communication is type safe since the data to be transferred can be type checked. When a group of tasks uses a monitor to mediate communication, data is passed indirectly through monitor data items. In the distributed case, calls to the entry routines of a monitor can be remote procedure calls. Hence, communication among tasks using monitors requires creation of a potentially large number of passive monitor objects, which must be properly managed by the user and the system. Therefore, monitors are best used to serialize access to passive objects and resources—objects such as data structures and files—without their own thread of execution.

2.1 Explicit Synchronization

For many purposes, the mutual exclusion, which is provided automatically by monitors, is all that is needed. However, it is sometimes necessary to synchronize tasks within the monitor. For that purpose, monitors provide *condition variables* (also called *event queues*) and the associated operations **signal** and **wait**. A condition variable can be thought of as a queue of waiting tasks. To join such a queue, the active task in a monitor executes a wait statement. For example, if a task executes the statement:

```
wait q
```

the task is blocked on condition variable q and the monitor is unlocked, which allows another task to use the monitor.

A task is reactivated from a condition variable when another (active) task executes a **signal** statement, for example:

```
signal q
```

```
uMonitor BoundedBuffer {
#   define QueueSize 10
    int front = 0, back = 0;             /* location of front and back of queue */
    int count = 0;                       /* number of used elements in the queue */
    int queue[QueueSize];                /* queue of integers */
    uCondition NotFull = U_CONDITION, NotEmpty = U_CONDITION;

    int query( void ) { return count; }  /* no mutual exclusion */

    uEntry void insert( int elem ) {
        if ( count == QueueSize ) uWait NotFull; /* buffer full ? */
        queue[back] = elem;              /* insert element into buffer */
        back = (back + 1) % QueueSize;
        count += 1;
        uSignal NotEmpty;                /* inform tasks, buffer not empty */
    }

    uEntry int remove( void ) {
        int elem;
        if (count == 0) uWait NotEmpty;  /* buffer empty ? */
        elem = queue[front];             /* remove element from buffer */
        front = (front + 1) % QueueSize;
        count -= 1;
        uSignal NotFull;                 /* inform tasks, buffer not full */
        return elem;
    }
}
```

Figure 1: Bounded Buffer – Explicit Signal

The effect of a **signal** statement is to remove one task from the specified condition variable (if such a task exists) and make it ready to run again. (The question of whether the signalling task or the signalled task runs first is discussed shortly.)

Condition variables and other lists of blocked tasks associated with a monitor can be implemented by various data structures, but queues are used most often, giving first-in first-out (FIFO) scheduling. This gives priority to the task that has been waiting the longest on a condition. Another possibility is to assign a priority to each waiting task [Hoare 1974, p. 553].

The **wait** and **signal** operations on condition variables in a monitor are similar to P and V operations on counting semaphores. The **wait** statement can block a task's execution, while a **signal** statement can cause another task to be unblocked. There are, however, differences between them. When a task executes a P operation, it does not necessarily block since the semaphore counter may be greater than zero. In contrast, when a task executes a **wait** statement it always blocks. When a task executes a V operation on a semaphore it either unblocks a task waiting on that semaphore or, if there is no task to unblock, increments the semaphore counter. In contrast, if a task executes a **signal** statement when there is no task to unblock, there is no effect on the condition variable. Another difference between semaphores and monitors is that tasks awoken by a V can resume execution without delay. In contrast, because tasks execute with mutual exclusion within a monitor, tasks awoken from a condition variable are restarted only when the monitor is unlocked.

When programming with monitors, it is common to associate with each condition variable an assertion about the state of the monitor. For example, in a bounded buffer, a condition variable might be associated with the assertion "the buffer is not full." Waiting on that condition variable would correspond to waiting until the condition is satisfied—that is, until the buffer is not full. Correspondingly, the active task would signal the condition variable only when the buffer is not full. The association between assertion and condition variable is usually implicit and not part of the language. Figure 1 illustrates a monitor for a bounded buffer using explicit synchronization. (Exact syntactic details are given in Section 6.2.)

```
uMonitor BoundedBuffer {
#   define QueueSize 10
    int front = 0, back = 0;              /* location of front and back of queue */
    int count = 0;                        /* number of used elements in the queue */
    int queue[QueueSize];                 /* queue of integers */

    int query( void ) { return count; }   /* no mutual exclusion */

    uEntry insert( int elem ) {
        uWaitUntil count != QueueSize;    /* buffer full ? */
        queue[back] = elem;               /* insert element into buffer */
        back = (back + 1) % QueueSize;
        count += 1;
    }

    uEntry int remove( void ) {
        int elem;
        uWaitUntil count != 0;            /* buffer empty ? */
        elem = queue[front];              /* remove element from buffer */
        front = (front + 1) % QueueSize;
        count -= 1;
        return( elem );
    }
}
```

Figure 2: Bounded Buffer – Implicit Signal

2.2 Implicit Synchronization

An alternative to using condition variables for monitor synchronization is to specify the associated assertions directly. *Automatic-signal* monitors, proposed by Hoare [1974, p. 556], eliminate condition variables and **signal** statements by modifying the **wait** statement to use a conditional expression:

wait conditional_expression

If the conditional expression is false, the task blocks and the monitor is unlocked. A waiting task is unblocked implicitly when the expression it is waiting on becomes true.

The programming language Edison [Hansen 1981a; Hansen 1981b; Hansen 1981c], designed by Per Brinch Hansen, used automatic-signal monitors. Automatic signalling was accomplished by arranging for waiting tasks to wake up repeatedly (in round-robin order) to check their conditions. This can lead to a large amount of context-switching overhead if tasks remain blocked for long periods, but this was felt to be of little consequence on a microprocessor system [Hansen 1981a, page 371]. The implementation of automatic signalling is further discussed in Section 3.2. Figure 2 illustrates a monitor for a bounded buffer using implicit synchronization.

2.3 Monitor Scheduling

A monitor is not a task and hence has no thread of control. Therefore, monitor routines are executed by the thread of the calling task. The state of the monitor, including whether it is locked, determines whether a calling task may or may not continue execution. Monitor scheduling occurs when the monitor becomes unlocked. A monitor becomes unlocked when a task executes a **wait**, **signal** statement or returns from a monitor entry routine. When a monitor is unlocked, the next task to use the monitor is then chosen from one of a number of queues internal to the monitor.

Figure 3 shows the general form of a monitor with a set of tasks using, or waiting to use, the monitor. When a calling task finds the monitor locked, it is added to the "entry queue"; otherwise it enters the monitor and locks it. When a task executes a **wait** statement, it is blocked and added to a specific "condition queue" and the monitor is unlocked. When a task executes a **signal** statement, it is blocked and added to the "signaller queue"; the task that has been signalled is moved from the specified condition queue to the "waiting queue" and the monitor is unlocked.

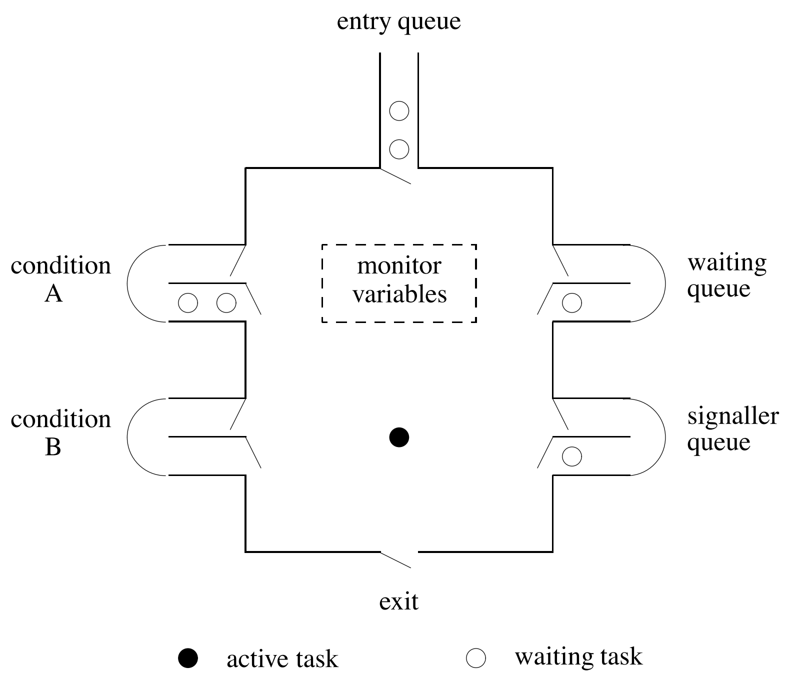


Figure 3: Processes Waiting to use a Monitor

In the case of an automatic-signal monitor, there are no condition queues and only one queue is needed to manage the tasks with false conditional expressions; we chose to put tasks with false conditional expressions on the waiting queue because each waiting task is effectively eligible to run when the monitor is unlocked so that it can recheck its conditional expression.

When a monitor becomes unlocked, it is not obvious which task should execute next; it could be a task from any of the entry, waiting, or signaller queues. Depending on the kind of monitor, a particular choice is made. All other tasks must wait until the monitor is again unlocked. Since this selection is done implicitly, the next task to resume execution in the monitor is not under direct user control. A monitor-synchronization operation may cause the monitor to unlock, but the selection of the next task to execute depends on the kind of monitor. The main difference among monitors is the algorithm used by the implicit monitor *scheduler* to select the next task to execute when the monitor is unlocked.

3 Monitor Classification

This section presents a taxonomy of monitors that encompasses all the extant monitors and suggests others not found in the literature or in existing programming languages. Initially, the monitors are divided into two groups based on whether the signal operation is explicit or implicit; within these two broad categories, several additional sub-categories are developed based on the semantics of the signal operation.

3.1 Explicit-Signal Monitors

An explicit-signal monitor is a monitor with an explicit **signal** statement (in contrast to an automatic-signal monitor, which has no **signal** statement). Several kinds of explicit-signal monitors have been presented in the literature, all of which can be categorized using the following classification scheme [Fortier 1989]. The classification scheme is based on an exhaustive case analysis of the scheduling possibilities for the three internal monitor queues—the entry, waiting and signaller queues—when a monitor is unlocked. The different kinds of monitors are classified based on the relative priorities associated with these three queues. Each queue has a specific priority, referred to as entry priority (E_p), waiting priority (W_p), and signaller priority (S_p), respectively. The relative orderings of these three priorities yields 13 different possibilities, which are given in Table 1. There is one case in which the 3 priorities are equal. There are $\binom{3}{2} = 3$ cases in which exactly two priorities are equal, and each equal pair can be either greater than or less than the third priority, hence $\binom{3}{2} \times 2 = 6$ cases. Finally, there are $3! = 6$ cases in which the priorities are all different. The right column of Table 1 shows how the traditional explicit-signal monitors fall into the categorization scheme. (One point of clarification about traditional monitor names. The names given in the right column come from Howard's [1976a] classification. However, Andrews and Schneider [1983, p. 19] and Andrews [1991, pp. 266-267] used the term *signal and continue* to describe Howard's *wait and notify* monitor.)

Queue priorities must not be confused with task priorities, nor should the monitor scheduler be confused with the operating-system task scheduler. Queue priorities are fixed, and a monitor scheduler uses queue priorities to arbitrate among tasks using a particular monitor. Task priorities are often variable and are used by the operating-system scheduler to arbitrate among tasks on a system-wide basis.

	relative priority	traditional monitor name
1	$E_p = W_p = S_p$	
2	$E_p = W_p < S_p$	Wait and Notify [Lampson and Redell 1980]
3	$E_p = S_p < W_p$	Signal and Wait [Howard 1976a]
4	$E_p < W_p = S_p$	
5	$E_p < W_p < S_p$	Signal and Continue [Howard 1976b]
6	$E_p < S_p < W_p$	Signal and Urgent Wait [Hoare 1974]
7	$E_p > W_p = S_p$	(rejected)
8	$E_p = S_p > W_p$	(rejected)
9	$S_p > E_p > W_p$	(rejected)
10	$E_p = W_p > S_p$	(rejected)
11	$W_p > E_p > S_p$	(rejected)
12	$E_p > S_p > W_p$	(rejected)
13	$E_p > W_p > S_p$	(rejected)

Table 1: Relative Priorities for Internal Monitor Queues

Of the 13 cases, we reject cases 7–13 for the following reasons. Consider the cases where the entry queue has the highest priority (cases 7, 12 and 13 of Table 1). In such monitors, a signalled or signaller task cannot resume execution until there are no calling tasks. This property creates the potential for an unbounded wait for signalled or signaller tasks, and it inhibits concurrency by preventing signalled or signaller tasks from getting out of the monitor and continuing execution. For example, if there are a large number of tasks calling the monitor, and each of them has to wait or signal, there will soon be a large number of tasks blocked on conditions inside the monitor, and the number of tasks accomplishing useful work will correspondingly diminish.

The same problem occurs if the entry queue has priority over the waiting queue or over the signaller queue (cases 8–11 of Table 1). If the entry queue has priority over the waiting queue, a signalled task resumes only when there are no more signaller or calling tasks; this creates the potential for an unbounded wait for signalled tasks if a continuous stream of tasks are calling the monitor. If the calling tasks have priority over the signaller tasks, a signaller task resumes only when there are no more signalled and calling tasks; this property creates the potential for an unbounded wait for signaller tasks if a continuous stream of tasks are calling the monitor. Since allowing the entry queue to have a priority greater then either of the internal queues has significant disadvantages and few if any compensating advantages, they are eliminated from further discussion. Therefore, only cases 1–6 of Table 1 are examined.

If two or more queues have equal priority, the scheduler chooses one arbitrarily. This encourages a style of programming where signals are used as "hints" [Lampson and Redell 1980, p. 111][Nelson 1991, p. 102]. In this approach, a task may signal a condition whenever it *might* be true; the signalled task is responsible for checking whether it actually *is* true. This is a cautious approach to concurrency control, where the signalled and signaller tasks can make no assumptions about order of execution, even within the monitor.

3.1.1 Immediate-Return Monitors

Brinch-Hansen and Hoare both discuss a restricted monitor in which the signal statement is permitted only before a return from a monitor-entry routine. We call this kind of monitor an *immediate-return* monitor. The semantics of an immediate-return signal are that *both* the signaller and signalled tasks continue execution. Because only one task can be active in the monitor, one of the two tasks must leave the monitor immediately; in this case, it is the signaller. Such a monitor does not allow a task to signal more than one condition or execute code after the **signal**; in essence, the **signal** statement acts as a **return** statement from the entry routine. This kind of monitor was suggested because it was observed that many monitor programs use **signal** statements only immediately before **return** statements [Brinch Hansen 1975]. However, immediate-return monitors are significantly less expressive than other explicit-signal monitors. In fact, as has been pointed out by Howard [1976b, p. 51] and Andrews [1991, p. 312], some monitor programs that can be written with ordinary explicit-signal monitors cannot be written with immediate-return monitors unless the interface to the monitor is changed.

Howard proposed an "extended immediate-return" monitor that also allows a **signal** to immediately precede **wait**; in this case, the signaller does not leave the monitor. Instead, the signaller moves a task from the condition variable of the **signal** to the waiting queue, blocks on the condition variable of the **wait**, and unlocks the monitor. The extended immediate-return monitor is as general as the other explicit-signal monitors (see Section 5).

The monitor-categorization scheme is applicable to both the immediate-return and extended immediate-return monitors; however, there is no signaller queue because either the signaller task leaves the monitor immediately or it is put on a condition queue. Therefore, the next active task is either a calling or a signalled task. Table 2 shows these possibilities. Again, the case where the entry queue has priority over an internal queue is rejected.

	relative priority	traditional monitor name
1	$E_p = W_p$	
2	$E_p < W_p$	Signal and Return [Brinch Hansen 1975]
3	$E_p > W_p$	(rejected)

Table 2: Relative Priorities for Extended Immediate-Return Monitor

3.2 Automatic-Signal Monitors

An automatic-signal monitor provides a **wait** statement of the form "**wait** conditional_expression." The kinds of variables allowed in the conditional expression can be used to further classify this kind of monitor. If both monitor variables and local variables of a monitor routine may appear in the conditional expression, the monitor is called a *general* automatic-signal monitor. If only monitor variables are allowed in the conditional expression, the monitor is called a *restricted* automatic-signal monitor.

When a general automatic-signal monitor is unlocked, finding the next task to execute can be expensive because, in the worst case, it involves re-evaluating the conditional expressions of all waiting tasks. Since a conditional expression can potentially depend on local variables of a task, including formal parameters of a monitor-entry routine, the local context of a task must be accessed to evaluate its conditional expression. The simplest way to accomplish this is to awaken tasks from the waiting queue one at a time so each can re-evaluate its conditional expression. If a task evaluates its condition and finds the condition true, it proceeds; otherwise it again blocks on the waiting queue and allows another task to try. The problem with this approach is that many context switches may occur before some task can proceed.

An alternative implementation is to have each waiting task copy into a global data area all the local context necessary to evaluate its conditional expression and provide a pointer to the code for its conditional_expression (or build a closure at the point of the **wait** statement if the language supports closures). This data must be linked together, possibly by attaching it to the waiting queue, so that it can be accessed by any waiting task. Now any task can check whether other tasks can execute by invoking the conditional_expressions until one evaluates to true or the end of list is reached. However, creating the necessary data (closure plus link field) and searching and evaluating the list is both complicated and runtime expensive. We are not aware of any implementation that takes this approach. In both this implementation and the simpler one described above, the time it takes to find the next task to execute is determined by the number of waiting tasks and the cost of re-evaluating their conditional expressions.

Restricted automatic-signal monitors have a much more efficient implementation. Since all variables in the conditional expressions are monitor variables, and hence do not depend on the context of individual tasks, the conditional expressions can be evaluated efficiently by the task that is about to unlock the monitor. The efficiency can be further improved by noting which conditional expressions represent distinct conditions; if two or more tasks are waiting for the same condition, it need only be evaluated once. Kessels [1977] proposed a notation that both restricts conditional expressions to use only monitor variables and also allows the programmer to specify which conditional expressions represent distinct conditions. His notation moves the conditional expressions from the **wait** statement into the monitor declarations and gives each a name; these names are then used in the **wait** statement instead of an expression, for example:

```
monitor
    var a, b : . . .
    var c : condexpr(a > b)    /* only monitor variables allowed in the expression */

    proc e(. . .)
        . . .
        wait c                 /* wait until the conditional expression, c, is true */
```

Since only monitor variables are allowed in a conditional expression, the time it takes to find the next task to execute is determined by the cost of re-evaluating the conditional expressions. Thus, compared to general automatic-signal monitors, there is the potential for significant execution time saving in determining the next task to execute. The drawback is that local entry-routine information, including formal parameters, cannot be used in a conditional expression of a restricted automatic-signal **wait**. (This drawback is similar to the restriction in Ada that the parameters of an **accept** statement cannot be used in its **when** clause to control accepting callers.)

The monitor-categorization scheme is applicable to automatic-signal monitors; however, there are no condition queues and no signaller queue. Therefore, when the monitor is unlocked the next active task is either a calling task or a task on the waiting queue waiting for its conditional expression to evaluate to true. Table 3 shows these possibilities. Again, the case where the entry queue has priority over an internal queue is rejected.

	relative priority	traditional monitor name
1	$E_p = W_p$	
2	$E_p < W_p$	Automatic Signal [Hoare 1974]
3	$E_p > W_p$	(rejected)

Table 3: Relative Priorities for Automatic-Signal Monitor

3.3 Simplified Classification

The remaining "useful" monitors are now organized along the following lines (see Table 4). First, the monitors are divided into two groups based on the priority of the entry queue. This division is useful because when the priority of the calling tasks is equal to either the signaller tasks or signalled tasks, it may make writing a monitor more complex. (This complexity is discussed further in Sections 4 and 7.) These two groups are called the Priority and No-Priority monitors: in priority monitors, tasks that have already entered the monitor have priority over calling tasks; in no-priority monitors, they do not. While the priority aspect of a monitor is discussed in both Howard's and Andrews's monitor analysis, the importance of this crucial property has often been highly underrated and even ignored in the explanation of a monitor's semantic behaviour.

Within the two groups, it is possible to pair the monitors based on aspects of the **signal** statement. In both "Signal and Wait" and the "Signal and Urgent Wait" monitors, the signaller task blocks on the signaller queue while the signalled task re-enters the monitor. In contrast, in both "Signal and Continue" and the "Wait and Notify" monitors, the signaller task continues execution in the monitor and signalled tasks can re-enter the monitor only when the monitor is unlocked. The next two monitors in Table 4 do not give priority to either signaller or signalled tasks; a signalling task may or may not block. These monitors are called *quasi-blocking*. Finally, the "Automatic Signal" and "Extended Immediate Return" monitors form two pairs. This organization makes it easier to remember and understand the different kinds of monitors. New names, given in italics, are suggested by the re-organization to reflect the categorization scheme. Notice there are four new kinds of monitors identified by the classification scheme.

4 Formal Semantics

The monitor classification presented in Section 3 is based on operational semantics. In our experience, most programmers find this approach easy to understand and use. However, for other purposes, especially programming-language definition, a more formal and less implementation-biased description is preferable. This section gives Hoare-style

Signal Characteristics	Priority	No Priority
Blocking	Signal and Urgent Wait $E_p < S_p < W_p$ *Priority Blocking (PB)*	Signal and Wait $E_p = S_p < W_p$ *No Priority Blocking (NPB)*
Non-Blocking	Signal and Continue $E_p < W_p < S_p$ *Priority Non-Blocking (PNB)*	Wait and Notify $E_p = W_p < S_p$ *No Priority Non-Blocking (NPNB)*
Quasi-Blocking	$E_p < W_p = S_p$ *Priority Quasi-Blocking (PQB)*	$E_p = W_p = S_p$ *No Priority Quasi-Blocking (NPQB)*
Extended Immediate Return	Signal and Return $E_p < W_p$ *Priority Immediate Return (PRET)*	$E_p = W_p$ *No Priority Immediate Return (NPRET)*
Automatic Signal	Automatic Signal $E_p < W_p$ *Priority Automatic Signal (PAS)*	$E_p = W_p$ *No Priority Automatic Signal (NPAS)*

Table 4: Useful Monitors

proof rules for the monitor types described in Section 3 and discusses the differences among them. In some cases, simplified proof rules are also given, which may be preferable for some purposes.

Another reason to examine proof rules is that the complexity of the semantics of a language construct is related, albeit somewhat loosely, to the ease of use of that construct. So comparing the axioms for various kinds of monitors provides one indication of how difficult the various kinds of monitors are to use.

Our proof methodology has the following limitations. We assume that program correctness does not depend on a specific queuing discipline such as first-in first-out (FIFO). In our experience, the correctness of programs usually does not depend on a specific queuing discipline, although the choice of queuing discipline may substantially affect performance. (The readers and writer solution of Section 6.3, which relies on FIFO queuing to avoid the problem of stale readers, is an exception.) Also, the issues of termination (liveness) and timing are not dealt with explicitly, although the use of history variables such as step counters is not precluded.

4.1 Notation

In this section and the next, it is assumed that the reader is familiar with standard axiomatic semantics of sequential programs. (For a general introduction to axiomatic semantics we recommend [Gries 1981].)

The symbol $\wedge$ denotes "and", $\Rightarrow$ denotes "implies", and $\equiv$ denotes logical equivalence. Substitution of b for a in a predicate P is denoted $\mathsf{P}^{\mathrm{b}}_{\mathrm{a}}$. A bracketed Boolean condition such as $[|\mathrm{q}| > 0]$ denotes 1 if the condition is true, 0 otherwise.

The signaller queue is denoted s, the waiting queue is denoted w, and the entry queue is denoted e. The set of all condition variables is denoted Q; this set includes only condition queues, not the signaller or waiting queues.

The length of any queue q is denoted $|\mathrm{q}|$. Queue lengths are automatically updated by the scheduler. We assume that the variables $|\mathrm{q}|$, for all condition variables q, as well as $|\mathrm{s}|$ and $|\mathrm{w}|$ can be read, but not modified, within the monitor. If the queue lengths are not directly available, it is easy to arrange to keep track of them explicitly by incrementing and decrementing variables. We do not assume that the length of the entry queue is available.

For purposes of developing proof rules, the tasks on the waiting queue are classified as to the condition queues from which they came. The "sub-queue" of tasks on w that came from condition queue q is denoted $\mathrm{w.q}$. The following equation always holds:

$$|\mathrm{w}| = \sum_{\mathrm{q} \in \mathrm{Q}} |\mathrm{w.q}|.$$

By default, quantifications range over the set of all condition queues; thus

$$\forall \mathrm{r} : (\mathsf{P})$$

is shorthand for

$\forall r \in Q : (\mathsf{P})$

The proof rules for monitors presented here are based on standard Hoare-style proof rules for sequential programs. Four additional rules are added, one for each of the monitor primitives **enter**, **signal**, **wait**, and **return**. In the proof rules for these four primitives, several predicates are important. (These predicates are similar to those used by Howard [1976a; 1976b].)

- the predicate I is the monitor invariant; it must hold whenever a task locks or unlocks the monitor.
- the predicate W_q must hold whenever a task that waited on condition q is about to be awakened,
- the predicate E must hold whenever a task is about to be awakened from the entry queue, and
- the predicate S must hold when a task is about to be awakened from the signaller queue.

These predicates are useful for proving both external and internal properties of monitors. An *external property* is a property visible to clients of the monitor; changing an external property could cause client programs to malfunction. An *internal property* is a property that cannot affect the correctness of client programs. In a monitor that implements a solution to the readers and writer problem, an example of an external property is that mutual exclusion is enforced. An example of an internal property is that read access is not unnecessarily delayed when the number of active writers is zero. The latter property rules out, for example, the trivial solution to the readers and writer problem in which no concurrency among readers is allowed.

4.2 Proof Rules for Monitors

Several authors have given proof rules for various kinds of monitors [Hoare 1974; Howard 1976a; Howard 1976b]. However, since our intent is to compare various scheduling disciplines, we have developed a methodology for deriving proof rules directly from an operational description of the scheduler. The main advantage of using this method is that the proof rules obtained for various kinds of monitors are all expressed in terms of the same predicates—I, W_q, E, and S, as described in Section 4.1. This makes it possible to compare proof rules for various kinds of monitors in a consistent fashion.

The first step is to decompose the monitor primitives **signal**, **wait**, and **return** into more primitive operations. The **signal** statement is decomposed as

```
signal q ≡
    <
      if   |q| > 0 →  w.q.enque(q.deque)
      []   |q| = 0 →  skip
      fi
      s.enque(self)
      schedule
    >
```

The **signal** operation is atomic, as indicated by the angle brackets. The deque primitive removes an arbitrary task from the appropriate queue and returns it. The enque primitive takes a task as its parameter; it puts that task on the appropriate queue. The read-only variable self is the active task. The **schedule** statement chooses the next task to execute; different monitors have different schedulers. The guarded form of alternation statement is used instead of the more standard if-then-else statement because the latter precludes nondeterminism. This capability is not important for **signal**, but is needed in the definition of **schedule**, below.

The **wait** statement is decomposed as

```
wait q ≡ q.enque(self); schedule
```

and the **return** statement is decomposed as

```
return ≡ schedule; exit
```

By definition, $s.\mathrm{deque}$ has the precondition $I \wedge S$, $e.\mathrm{deque}$ has the precondition $I \wedge E$, and $q.\mathrm{deque}$ has the precondition $I \wedge W_q$.

The details of the **schedule** statement depend on the kind of monitor under consideration. The schedule statement for the priority quasi-blocking monitor, which is fairly typical, is as follows:

```
$\mathrm{nextTask} :=$
    if  $|s| > 0 \rightarrow s.\mathrm{deque}$
    []  $|w.q_1| > 0 \rightarrow w.q_1.\mathrm{deque}$
    []  ...
    []  $|w.q_n| > 0 \rightarrow w.q_n.\mathrm{deque}$
    []  $|s| = 0 \wedge |w| = 0 \rightarrow e.\mathrm{deque}$
    fi
```

In the above statement, many guards can simultaneously be true, in which case the scheduler chooses among eligible tasks nondeterministically.

The enque and deque primitives have simple proof rules since only the length of the queues, not their actual content, is important. (It is possible to make this simplification only because queuing discipline is ignored.) In its effect on queue lengths, the primitive $q.\mathrm{enque}$ is equivalent to the statement:

$$|q| := |q| + 1$$

Similarly, in its effect on queue lengths, $q.\mathrm{deque}$ is equivalent to:

$$|q| := |q| - 1$$

All the schedulers ensure that deque is not executed on an empty queue.

These substitutions reduce the proof rules for **signal** and **wait** to well-known proof rules for alternation and assignment. After these substitutions are made, the weakest pre-condition of the scheduler is computed. For example, to obtain the weakest pre-condition of the priority quasi-blocking scheduler, begin with the expanded form:

```
$\mathrm{nextTask} :=$
    if  $|s| > 0 \rightarrow \{I \wedge S\}\ s.\mathrm{deque}$
    []  $|w.q_1| > 0 \rightarrow \{I \wedge W_{q_1}\}\ w.q_1.\mathrm{deque}$
    []  ...
    []  $|w.q_n| > 0 \rightarrow \{I \wedge W_{q_n}\}\ w.q_n.\mathrm{deque}$
    []  $|s| = 0 \wedge |w| = 0 \rightarrow \{I \wedge E\}\ e.\mathrm{deque}$
    fi
```

Then substitute appropriate assignments for enque and deque operations:

```
if  $|s| > 0 \rightarrow \{I \wedge S\}\ |s| := |s| - 1$
[]  $|w.q_1| > 0 \rightarrow \{I \wedge W_{q_1}\}\ |w.q_1| := |w.q_1| - 1$
[]  ...
[]  $|w.q_n| > 0 \rightarrow \{I \wedge W_{q_n}\}\ |w.q_n| := |w.q_n| - 1$
[]  $|s| = 0 \wedge |w| = 0 \rightarrow \{I \wedge E\}$ skip
fi
```

The assignment to $\mathrm{nextTask}$ has been dropped since it has no effect on the lengths of queues.

Now standard techniques can be used to calculate the weakest precondition of the scheduler. In this case, the weakest precondition is quite complicated:

$$\{(|s| > 0 \Rightarrow (I \wedge S)) \wedge \forall r : (|w.r| > 0 \Rightarrow (I \wedge W_r)) \wedge ((|s| = 0 \wedge |w| = 0) \Rightarrow (I \wedge E))\}$$

which simplifies to:

$$\{I \wedge (|s| > 0 \Rightarrow S) \wedge \forall r : (|w.r| > 0 \Rightarrow W_r) \wedge ((|s| = 0 \wedge |w| = 0) \Rightarrow E)\}$$

After the weakest pre-condition of the scheduler has been found, it is easy to work backward to obtain the weakest pre-condition of **signal** and **wait**. For example, **wait** q is equivalent to:

$$\mathrm{q.enque(self);}\ \textsf{schedule}$$

which, in its effect on the lengths of queues, is in turn equivalent to:

$$|\mathrm{q}| := |\mathrm{q}| + 1;\ \textsf{schedule}$$

So the weakest precondition of **wait** q is

$$\{\mathrm{I} \wedge (|\mathrm{s}| > 0 \Rightarrow \mathrm{S}) \wedge \forall \mathrm{r} : (|\mathrm{w.r}| > 0 \Rightarrow \mathrm{W_r}) \wedge ((|\mathrm{s}| = 0 \wedge |\mathrm{w}| = 0) \Rightarrow \mathrm{E})\}^{|\mathrm{q}|+1}_{|\mathrm{q}|}$$

The weakest precondition of signal can be obtained in similar fashion.

Post-conditions can be found by working forward through the scheduler. The strongest post-condition of **signal** is the same as the strongest post-condition of s.deque in the scheduler since it is at exactly this point that a signaller awakens. For the priority quasi-blocking monitor, the strongest post-condition of **signal** q is

$$\{\mathrm{I} \wedge \mathrm{S} \wedge \forall \mathrm{r} : (|\mathrm{w.r}| > 0 \Rightarrow \mathrm{W_r})\}^{|\mathrm{s}|+1}_{|\mathrm{s}|}$$

Similarly, the strongest post-condition of **wait** q is the same as the strongest post-condition of q.deque:

$$\{\mathrm{I} \wedge \forall \mathrm{r} : (|\mathrm{w.r}| > 0 \Rightarrow \mathrm{W_r}) \wedge (|\mathrm{s}| > 0 \Rightarrow \mathrm{S})\}^{|\mathrm{w.q}|+1}_{|\mathrm{w.q}|}$$

and the strongest postcondition of **enter** is the same as the strongest postcondition of e.deque:

$$\{\mathrm{I} \wedge \mathrm{E} \wedge |\mathrm{s}| = 0 \wedge |\mathrm{w}| = 0)\}$$

Finally, the postcondition of **return** is always **false** since control never reaches the point after **return**.

The following sections give proof rules for each kind of useful monitor. Derivations of these proof rules are not given since, for each kind of monitor, they follow almost directly from the definition of **schedule**.

4.2.1 No-Priority Blocking Monitor (NPB): $\mathrm{S_p} = \mathrm{E_p} < \mathrm{W_p}$

The proof rules for the no-priority blocking monitor are as follows:

$$\{\textsf{true}\}\ \textsf{enter}\ \{\mathrm{I} \wedge \mathrm{E} \wedge (|\mathrm{s}| > 0 \Rightarrow \mathrm{S})\}$$
$$\{\mathrm{I}^{|\mathrm{s}|+1,|\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{s}|,|\mathrm{q}|} \wedge (|\mathrm{q}| > 0 \Rightarrow \mathrm{W_q}{}^{|\mathrm{s}|+1,|\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{s}|,|\mathrm{q}|}) \wedge (|\mathrm{q}| = 0 \Rightarrow (\mathrm{S} \wedge \mathrm{E})^{|\mathrm{s}|+1,|\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{s}|,|\mathrm{q}|})\ \}$$
$$\textsf{signal}\ \mathrm{q} \quad \{\mathrm{I} \wedge \mathrm{S} \wedge \mathrm{E}\}^{|\mathrm{s}|+1}_{|\mathrm{s}|}$$
$$\{\mathrm{I} \wedge \mathrm{E} \wedge (|\mathrm{s}| > 0 \Rightarrow \mathrm{S})\}^{|\mathrm{q}|+1}_{|\mathrm{q}|} \quad \textsf{wait}\ \mathrm{q} \quad \{\mathrm{I} \wedge \mathrm{W_q}\}$$
$$\{\mathrm{I} \wedge \mathrm{E} \wedge (|\mathrm{s}| > 0 \Rightarrow \mathrm{S})\} \quad \textsf{return} \quad \{\textsf{false}\}$$

(Recall that $[|\mathrm{q}| > 0]$ is 1 if $|\mathrm{q}| > 0$ and 0 otherwise.) At scheduling points, the scheduler chooses a task from w if $|\mathrm{w}|$ is positive. Otherwise, the scheduler chooses a task from either the signaller queue s or the entry queue e; if both queues contain tasks, the scheduler chooses between them nondeterministically. In effect, the queues e and s are coalesced. (Our implementation of the NPB monitor actually combines these two queues.)

Initially, $|\mathrm{w}| = 0$. Whenever $|\mathrm{w}|$ is incremented (in **signal**) the scheduler immediately decrements it. Since **signal** is atomic, this implies that the length of w is always zero in any observable state. Hence the variable $|\mathrm{w.q}|$ is not useful and has been dropped from the proof rules. (Our implementation takes advantage of this fact by eliminating the w queue.)

The chief difficulty in programming with this kind of monitor is that predicates involving S must be proven in numerous places in order to conclude S after **signal**. If S is taken to be the same as E, and if $|\mathrm{s}|$ is not used, the proof rules simplify to:

$$\{\textsf{true}\}\ \textsf{enter}\ \{\mathrm{I} \wedge \mathrm{E}\}$$
$$\{\mathrm{I}^{|\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{q}|} \wedge (|\mathrm{q}| > 0 \Rightarrow \mathrm{W_q}{}^{|\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{q}|}) \wedge (|\mathrm{q}| = 0 \Rightarrow \mathrm{E}^{|\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{q}|})\}\ \textsf{signal}\ \mathrm{q} \quad \{\mathrm{I} \wedge \mathrm{E}\}$$
$$\{\mathrm{I} \wedge \mathrm{E}\}^{|\mathrm{q}|+1}_{|\mathrm{q}|} \quad \textsf{wait}\ \mathrm{q} \quad \{\mathrm{I} \wedge \mathrm{W_q}\}$$
$$\{\mathrm{I} \wedge \mathrm{E}\} \quad \textsf{return} \quad \{\textsf{false}\}$$

However, this does not eliminate the fundamental problem: the monitor is in a rather unconstrained state when a signaller awakens. In our experience, incorrect use of **signal** is the chief source of difficulty in using this kind of monitor; programmers seem naturally to think of signals as nonblocking and often make unwarranted assumptions about the monitor state when a signaller awakens.

4.2.2 Priority Blocking (PB): $\mathrm{E_p} < \mathrm{S_p} < \mathrm{W_p}$

The proof rules for the priority blocking monitor are as follows:

$$\begin{array}{l}
\{\textsf{true}\}\ \ \textsf{enter}\ \{\mathrm{I} \wedge \mathrm{E} \wedge (|\mathrm{s}| = 0)\} \\
\{\mathrm{I}^{|\mathrm{s}|+1, |\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{s}|, |\mathrm{q}|} \wedge (|\mathrm{q}| > 0 \Rightarrow \mathrm{W_q}{}^{|\mathrm{s}|+1, |\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{s}|, |\mathrm{q}|}) \wedge (|\mathrm{q}| = 0 \Rightarrow \mathrm{S}^{|\mathrm{s}|+1, |\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{s}|, |\mathrm{q}|})\} \\
\qquad \textsf{signal}\ \mathrm{q} \quad \{\mathrm{I} \wedge \mathrm{S}\}^{|\mathrm{s}|+1}_{|\mathrm{s}|} \\
\{\mathrm{I} \wedge (|\mathrm{s}| > 0 \Rightarrow \mathrm{S}) \wedge (|\mathrm{s}| = 0 \Rightarrow \mathrm{E})\}^{|\mathrm{q}|+1}_{|\mathrm{q}|}\ \ \textsf{wait}\ \mathrm{q} \quad \{\mathrm{I} \wedge \mathrm{W_q}\} \\
\{\mathrm{I} \wedge (|\mathrm{s}| > 0 \Rightarrow \mathrm{S}) \wedge (|\mathrm{s}| = 0 \Rightarrow \mathrm{E})\}\ \ \textsf{return}\ \{\textsf{false}\}
\end{array}$$

At scheduling points, the priority blocking monitor (or *Hoare* monitor) chooses a task from w if $|\mathrm{w}| > 0$, otherwise from s if $|\mathrm{s}| > 0$, otherwise from e. The variable $|\mathrm{w.q}|$ does not appear in the proof rules, for the same reason given in the previous section.

There has been considerable discussion about the precise semantics of this monitor. Hoare's original paper [1974] contains both an operational and an axiomatic description. It is clear from the operational description, especially the implementation using semaphores, that Hoare intended that signalling an empty condition variable be allowed, and intended that waiting signallers execute before new tasks from the entry queue. However, Hoare's proof rules, which are equivalent to:

$$\begin{array}{l}
\{\mathrm{I}\}\ \ \textsf{wait}\ \mathrm{q}\ \{\mathrm{I} \wedge \mathrm{W_q}\} \\
\{\mathrm{I} \wedge \mathrm{W_q}\}\ \ \textsf{signal}\ \mathrm{q}\ \{\mathrm{I}\}
\end{array}$$

in our notation, are not strong enough to prove some facts about monitors that use these two features [Howard 1976a; Adams and Black 1982]. Howard's strengthened proof rules [1976a] for the Hoare monitor address the second point—that waiting signallers should execute before new tasks—but do not address the first: signalling an empty condition variable is still forbidden. Adams and Black [1982] propose still stronger proof rules that do allow signalling empty condition variables. (Unfortunately, this paper contained a mistake about the applicability of Howard's proof rules [Howard 1982; Adams and Black 1983].)

The priority blocking monitor suffers from almost the same difficulties as the no-priority blocking monitor—the monitor is in a relatively unconstrained state when a signaller awakens because a great deal of activity may take place before the signaller resumes execution. As in the no-priority blocking monitor, the axioms and proof obligations can be simplified by setting $\mathrm{S} \equiv \mathrm{E}$ and ignoring $|\mathrm{s}|$, which results in simpler proof rules:

$$\begin{array}{l}
\{\textsf{true}\}\ \ \textsf{enter}\ \{\mathrm{I} \wedge \mathrm{E}\} \\
\{\mathrm{I}^{|\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{q}|} \wedge (|\mathrm{q}| > 0 \Rightarrow \mathrm{W_q}{}^{|\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{q}|}) \wedge (|\mathrm{q}| = 0 \Rightarrow \mathrm{E}^{|\mathrm{q}|-[|\mathrm{q}|>0]}_{|\mathrm{q}|})\}\ \ \textsf{signal}\ \mathrm{q} \quad \{\mathrm{I} \wedge \mathrm{E}\} \\
\{\mathrm{I} \wedge \mathrm{E}\}^{|\mathrm{q}|+1}_{|\mathrm{q}|}\ \ \textsf{wait}\ \mathrm{q} \quad \{\mathrm{I} \wedge \mathrm{W_q}\} \\
\{\mathrm{I} \wedge \mathrm{E}\}\ \ \textsf{return}\ \{\textsf{false}\}
\end{array}$$

By further weakening the proof rules—setting E to true, disregarding the length of q, and assuming that **signal** q is never executed if q is empty—Hoare's original proof rules for **signal** and **wait** are obtained.

4.2.3 No-Priority Nonblocking (NPNB): $\mathrm{E_p} = \mathrm{W_p} < \mathrm{S_p}$

The axioms for the no-priority nonblocking monitor are as follows:

$$\begin{array}{l}
\{\textsf{true}\}\ \ \textsf{enter}\ \{\mathrm{I} \wedge \mathrm{E} \wedge \forall \mathrm{r} : (|\mathrm{w.r}| > 0 \Rightarrow \mathrm{W_r})\} \\
\{\mathrm{I} \wedge \mathrm{S}\}^{|\mathrm{q}|-[|\mathrm{q}|>0], |\mathrm{w.q}|+[|\mathrm{q}|>0]}_{|\mathrm{q}|, |\mathrm{w.q}|}\ \ \textsf{signal}\ \mathrm{q}\ \{\mathrm{I} \wedge \mathrm{S}\} \\
\{\mathrm{I} \wedge \forall \mathrm{r} : (|\mathrm{w.r}| > 0 \Rightarrow \mathrm{W_r}) \wedge \mathrm{E}\}^{|\mathrm{q}|+1}_{|\mathrm{q}|}\ \ \textsf{wait}\ \mathrm{q}\ \{\mathrm{I} \wedge \forall \mathrm{r} : (|\mathrm{w.r}| > 0 \Rightarrow \mathrm{W_r}) \wedge \mathrm{E}\}^{|\mathrm{w.q}|+1}_{|\mathrm{w.q}|} \\
\{\mathrm{I} \wedge \mathrm{E} \wedge \forall \mathrm{r} : (|\mathrm{w.r}| > 0 \Rightarrow \mathrm{W_r})\}\ \ \textsf{return}\ \{\textsf{false}\}
\end{array}$$

The proof rules for **signal** can be simplified by noting that, since signal never transfers control to another task, there is no need to meet any particular precondition before signalling. This leads to the following simple proof rule for signal:

$$\{\mathrm{A}\}^{|\mathrm{q}|-[|\mathrm{q}|>0], |\mathrm{w.q}|+[|\mathrm{q}|>0]}_{|\mathrm{q}|, |\mathrm{w.q}|}\ \ \textsf{signal}\ \mathrm{q}\ \{\mathrm{A}\}$$

where A is an arbitrary predicate. (Our implementation takes advantage of this fact by eliminating the s queue.)

The proof rules can be further simplified by setting E and all W_q to **true** and disregarding the variables $|q|$ and $|w.q|$, resulting in the following extremely simple rules:

$$\begin{array}{l}\{\textsf{true}\}\ \textsf{enter}\ \{I\}\\ \{A\}\ \textsf{signal}\,q\ \{A\}\\ \{I\}\ \textsf{wait}\,q\ \{\ I\}\\ \{I\}\ \textsf{return}\ \{\textsf{false}\}\end{array}$$

It may seem that this is an oversimplification since the important predicates W_q have been lost. There is, however, a way around this; if each **wait** statement is embedded in a loop such as

$$\textsf{do}\ \textsf{not}\,W_q \rightarrow\ \textsf{wait}\,q\quad \textsf{od}$$

the postcondition W_q is established regardless of the precondition. This coding style corresponds to using signals as "hints" (see Section 3.1).

These simplified semantics are essentially the ones adopted in Modula-2+ and Modula-3 [Nelson 1991]. A point worth noting is that, since all references to the lengths of queues have been eliminated, an implementation is free to awaken more than one task with one signal. The implementation of condition variables in Modula-2+ and Modula-3 takes advantage of this and occasionally, for the sake of efficiency, awakens two tasks instead of one [Cardelli *et al.* 1988; Nelson 1991].

In blocking monitors, the axiom for **wait** is simple at the expense of complicating the axiom for **signal**. In non-blocking monitors, the transfer of control is deferred until the signaller unlocks the monitor, making the axiom for **signal** simple; the drawback is that a more complicated predicate must be proven when a task unlocks the monitor. In our experience, this is the chief practical difficulty with nonblocking monitors: programmers sometimes fail to cope with the fact that the transfer of control is not immediate. For example, a common error is to write a monitor-entry routine that signals a condition variable when some predicate is satisfied, only to negate the predicate before vacating the monitor. In addition, in the no-priority nonblocking monitor, any new tasks entering the monitor must ensure that they do not falsify any predicates expected by tasks on w. It is also possible to signal a condition q when W_q is false but then ensure that it becomes true before vacating the monitor; this practice can lead to programs that—although perhaps correct—are extremely difficult to understand.

4.2.4 Priority Nonblocking (PNB): $E_p < W_p < S_p$

The axioms for the priority nonblocking monitor are as follows:

$$\begin{array}{l}\{\textsf{true}\}\ \textsf{enter}\ \{I \wedge E \wedge |w| = 0\}\\ \{I \wedge S\}^{|q|-[|q|>0],|w.q|+[|q|>0]}_{|q|,|w.q|}\ \textsf{signal}\,q\ \{I \wedge S\}\\ \{I \wedge \forall r : (|w.r| > 0 \Rightarrow W_r) \wedge (|w| = 0 \Rightarrow E)\}^{|q|+1}_{|q|}\ \textsf{wait}\,q\ \{I \wedge \forall r : (|w.r| > 0 \Rightarrow W_r)\}^{|w.q|+1}_{|w.q|}\\ \{I \wedge \forall r : (|w.r| > 0 \Rightarrow W_r) \wedge (|w| = 0 \Rightarrow E\}\,\textsf{return}\,\{\textsf{false}\}\end{array}$$

As in the NPNB monitor, signal never results in a transfer of control so the proof rule for signal can be simplified to

$$\{A\}^{|q|-[|q|>0],|w.q|+[|q|>0]}_{|q|,|w.q|}\ \textsf{signal}\,q\ \{A\}$$

for arbitrary predicate A.

As with the no-priority nonblocking monitor, these rules can be simplified by setting W_q to **true** for all condition variables q, yielding much simpler proof rules. However, a less drastic way to simplify the proof rules for priority nonblocking monitors is to use the **signal** statement in a disciplined way, adopting the following rules:

- empty condition variables are never signalled,
- at any time, there is only one pending signal, and
- **signal** is executed only prior to cleaning up and unlocking the monitor.

In other words, **signal** is used only in the following ways:

$$\begin{array}{l}
\{\mathrm{I} \wedge \mathrm{W_q}\}^{|q|-1,|w.q|+1}_{|w.q|}\ \ \textsf{signal}\ q\ \{\mathrm{I} \wedge \mathrm{W_q}\}\ \mathrm{cleanup}\ \{\mathrm{I} \wedge \mathrm{W_q}\}\ \ \textsf{wait}\ q \\
\{\mathrm{I} \wedge \mathrm{W_q}\}^{|q|-1,|w.q|+1}_{|w.q|}\ \ \textsf{signal}\ q\ \{\mathrm{I} \wedge \mathrm{W_q}\}\ \mathrm{cleanup}\ \{\mathrm{I} \wedge \mathrm{W_q}\}\ \ \textsf{return}
\end{array}$$

where cleanup is a fragment of code that leaves $\mathrm{I} \wedge \mathrm{W_q}$ invariant. Using this discipline, the priority nonblocking monitor is treated somewhat like a priority blocking monitor; control is passed (almost) directly from the signaller to the waiting task. However, there are two important differences. First, using the rules above, the signaller can "clean up" before transferring control to the signalled task; this is not possible in a priority blocking monitor. In linear code, this capability is useful only in a cosmetic sense since the **signal** could be placed after the cleanup. However, the signal could also occur in nested subroutine calls, with cleanup happening as the calls complete. The second important difference, using the rule above, is that the signalling task exits the monitor before the signalled task enters; in contrast, the signaller in a priority blocking monitor waits on the signaller queue until one or more signalled tasks unlock the monitor. Hence, in a priority blocking monitor, the signaller becomes blocked just prior to executing **return**, which inhibits concurrency.

4.2.5 No-Priority Quasi-Blocking (NPQB): $\mathrm{E_p} = \mathrm{S_p} = \mathrm{W_p}$

The axioms of the no-priority quasi-blocking monitor are as follows:

$$\begin{array}{l}
\{\textsf{true}\}\ \ \textsf{enter}\ \{\mathrm{I} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge (|s| > 0 \Rightarrow \mathrm{S}) \wedge \mathrm{E}\} \\
\{\mathrm{I} \wedge \mathrm{S} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge \mathrm{E}\}^{|s|+1,|q|-[|q|>0],|w.q|+[|q|>0]}_{|s|,|q|,|w.q|} \\
\qquad \textsf{signal}\ q\quad \{\mathrm{I} \wedge \mathrm{S} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge \mathrm{E}\}^{|s|+1}_{|s|} \\
\{\mathrm{I} \wedge (|s| > 0 \Rightarrow \mathrm{S}) \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge \mathrm{E}\}^{|q|+1}_{|q|} \\
\qquad \textsf{wait}\ q\quad \{\mathrm{I} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge (|s| > 0 \Rightarrow \mathrm{S}) \wedge \mathrm{E}\}^{|w.q|+1}_{|w.q|} \\
\{\mathrm{I} \wedge (|s| > 0 \Rightarrow \mathrm{S}) \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge \mathrm{E}\}\quad \textsf{return}\quad \{\textsf{false}\}
\end{array}$$

Since the predicates I and E appear only in conjunction with each other, neither has an independent value, and E can be folded into I, which simplifies the proof rules somewhat:

$$\begin{array}{l}
\{\textsf{true}\}\ \ \textsf{enter}\ \{\mathrm{I} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge (|s| > 0 \Rightarrow \mathrm{S})\} \\
\{\mathrm{I} \wedge \mathrm{S} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r})\}^{|s|+1,|q|-[|q|>0],|w.q|+[|q|>0]}_{|s|,|q|,|w.q|} \\
\qquad \textsf{signal}\ q\ \{\mathrm{I} \wedge \mathrm{S} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r})\}^{|s|+1}_{|s|} \\
\{\mathrm{I} \wedge (|s| > 0 \Rightarrow \mathrm{S}) \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r})\}^{|q|+1}_{|q|} \\
\qquad \textsf{wait}\ q\quad \{\mathrm{I} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge (|s| > 0 \Rightarrow \mathrm{S})\}^{|w.q|+1}_{|w.q|} \\
\{\mathrm{I} \wedge (|s| > 0 \Rightarrow \mathrm{S}) \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r})\}\quad \textsf{return}\quad \{\textsf{false}\}
\end{array}$$

However, the rules remain complicated. Quasi-blocking monitors are discussed further in the next section.

4.2.6 Priority Quasi-Blocking (PQB): $\mathrm{E_p} < \mathrm{S_p} = \mathrm{W_p}$

The axioms of the priority quasi-blocking monitor are as follows:

$$\begin{array}{l}
\{\textsf{true}\}\ \ \textsf{enter}\ \{\mathrm{I} \wedge \mathrm{E} \wedge |s| = 0 \wedge |w| = 0\} \\
\{\mathrm{I} \wedge \mathrm{S} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r})\}^{|s|+1,|q|-[|q|>0],|w.q|+[|q|>0]}_{|s|,|q|,|w.q|} \\
\qquad \textsf{signal}\ q\quad \{\mathrm{I} \wedge \mathrm{S} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r})\}^{|s|+1}_{|s|} \\
\{\mathrm{I} \wedge (|s| > 0 \Rightarrow \mathrm{S}) \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge (|w| = 0 \wedge |s| = 0) \Rightarrow \mathrm{E})\}^{|q|+1}_{|q|} \\
\qquad \textsf{wait}\ q\quad \{\mathrm{I} \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge (|s| > 0 \Rightarrow \mathrm{S})\}^{|w.q|+1}_{|w.q|} \\
\{\mathrm{I} \wedge (|s| > 0 \Rightarrow \mathrm{S}) \wedge \forall r : (|w.r| > 0 \Rightarrow \mathrm{W_r}) \wedge (|w| = 0 \wedge |s| = 0) \Rightarrow \mathrm{E})\}\quad \textsf{return}\quad \{\textsf{false}\}
\end{array}$$

Quasi-blocking monitors, both priority and no-priority, have complex proof rules. In practice, the complexity of using the predicates S, $\mathrm{W_q}$, and E is often prohibitive, and they are often all set to **true**. This yields proof rules that are almost trivial:

$\{\textsf{true}\}$ enter $\{I\}$
$\{I\}$ signal q $\{I\}$
$\{I\}$ wait q $\{I\}$
$\{I\}$ return $\{\textsf{false}\}$

Again, this corresponds to using signals as hints; wait statements can be embedded in loops to establish predicates.

One advantage of quasi-blocking monitors is that the scheduler is allowed maximal freedom to choose among ready tasks; thus the semantics of quasi-blocking monitors do not interfere with other scheduling considerations such as task priority. The usefulness of this approach is demonstrated by the fact that most operating-system kernels are essentially no-priority quasi-blocking monitors in which the monitor entry points are system calls and device interrupts. In most operating systems, some variant of priority scheduling is used to choose among ready tasks.

4.2.7 Extended Immediate Return (PRET, NPRET): $E_p = W_p$ and $E_p < W_p$

The priority and no-priority extended immediate-return monitors have proof rules identical to the priority and no-priority nonblocking monitors, respectively, since they are merely restricted forms of those monitors.

4.2.8 No-priority General Automatic Signal Monitors (NPAS): $E_p = W_p$

Automatic signal monitors do not have a signal statement; instead signalling is done implicitly when tasks execute wait or return. Thus, neither the predicate S nor the variable $|s|$ appears in proofs of automatic signal monitors.

The proof rules for the no-priority automatic signal monitor are as follows:

$\{\textsf{true}\}$ enter $\{I \wedge E\}$
$\{I \wedge E\}$ wait W_q $\{I \wedge E \wedge W_q\}$
$\{I \wedge E\}$ return $\{\textsf{false}\}$

Since neither I nor E appears except in conjunction with the other, they have no independent value and E is generally folded into I. This yields the following extremely simple proof rules:

$\{\textsf{true}\}$ enter $\{I\}$
$\{I\}$ wait W_q $\{I \wedge W_q\}$
$\{I\}$ return $\{\textsf{false}\}$

4.2.9 Priority General Automatic Signal Monitor (PAS): $E_p < W_p$

The proof rules for the priority version of the automatic signal monitor are as follows:

$\{\textsf{true}\}$ enter $\{I \wedge E \wedge \forall r : (|W_r| > 0 \Rightarrow \textsf{not}\, W_r\}$
$\{I \wedge (\forall r : (\textsf{not}\, W_r)) \Rightarrow E\}$ wait W_q $\{I \wedge W_q\}$
$\{I \wedge (\forall r : (\textsf{not}\, W_r)) \Rightarrow E\}$ return $\{\textsf{false}\}$

(Note that $|W_r|$ means the number of tasks that are waiting for W_r.) These proof rules are considerably more complex than the no-priority version of the automatic signal monitor. The additional complexity does not seem warranted since the priority version seems to have very little additional expressive power over the no-priority version. Setting $E \equiv \textsf{true}$ and dropping the last conjunct from the postcondition of enter again yields extremely simple proof rules. There may, of course, be performance reasons for choosing the priority version.

4.2.10 No-priority Restricted Automatic Signal Monitors (NPRAS): $E_p = W_p$

The proof rules for the no-priority restricted automatic-signal monitor are as follows:

$\{\textsf{true}\}$ enter $\{I \wedge E\}$
$\{I \wedge E\}_{|q|}^{|q|+1}$ wait q $\{I \wedge E \wedge W_q\}$
$\{I \wedge E\}$ return $\{\textsf{false}\}$

Since neither I nor E appears except in conjunction with the other, they have no independent value and E is generally folded into I. Also, since signalling is automatic, the variable $|q|$ is of little value and is usually ignored. These considerations yield the following extremely simple proof rules:

$$\{\textsf{true}\}\ \textsf{enter}\ \{I\}$$
$$\{I\}\ \textsf{wait}\ q\ \{I \wedge W_q\}$$
$$\{I\}\ \textsf{return}\ \{\textsf{false}\}$$

4.2.11 Priority Restricted Automatic Signal Monitors (PRAS), $E_p < W_p$

The proof rules for the priority restricted automatic-signal monitor are as follows:

$$\{\textsf{true}\}\ \textsf{enter}\ \{I \wedge E\}$$
$$\{I \wedge (\forall r : (\textsf{not}\ W_r)) \Rightarrow E\}^{|q|+1}_{|q|}\ \textsf{wait}\ q\ \{I \wedge W_q\}$$
$$\{I \wedge (\forall r : (\textsf{not}\ W_r)) \Rightarrow E\}\ \textsf{return}\ \{\textsf{false}\}$$

In practice, the predicate E is usually dropped and $|q|$ ignored, yielding the same simplified proof rules as for the no-priority automatic signal monitor.

Since wait conditions in restricted automatic signal monitors cannot depend on local variables or parameters of monitor entries, any wait condition can be evaluated by any task. Hence, the restricted automatic-signal monitor is much more efficient than the general version. Thus restricted automatic-signal monitors are a compromise between general automatic-signal monitors, which are easy to use but very inefficient, and explicit signal monitors, which are efficient but more difficult to use. However, the inability to use local entry routine information in the conditional expression precludes simple solutions to certain kinds of important problems (e.g., certain disk schedulers).

5 Monitor Equivalence

All monitors are equivalent in the weak sense that any monitor program written for one kind of monitor can be implemented using any other kind of monitor. One way to demonstrate this weak equivalence is to note that any kind of monitor can be implemented with semaphores and can in turn be used to implement semaphores. Thus any monitor program for one kind of monitor can be mechanically translated into a program for any other kind of monitor by "compiling" the monitor program into code that synchronizes using semaphore operations and then using the other kind of monitor to implement the semaphore operations. However, this transformation is unsatisfying because it yields a much lower level program than the original. (This kind of weak equivalence between language features has been called the "Turing tar pit" by Howard [1976b, p. 49].) We are interested in transformations that preserve the basic structure of the monitor program. In particular, our transformations do not introduce any new types, nor do they change the monitor interface or its overall structure. These transformations are amenable to language translators when converting from one kind of monitor in one language to a different kind of monitor in another language. In addition, examining transformations between different kinds of monitors provides some insight into how to use various kinds of monitors to achieve certain effects.

The simplest program transformation is the identity transformation. Some scheduling disciplines are refinements of others. At each scheduling point—each **signal**, **wait**, or **return**—a monitor must choose, perhaps nondeterministically, a task to execute next. We say that a scheduler A is a *refinement* of scheduler B if, at each scheduling point, scheduler A chooses a task that is also a valid choice for B. For example, the priority blocking scheduler is a refinement of the priority quasi-blocking scheduler because the priority blocking scheduler always chooses a task that *could* have been chosen by the priority quasi-blocking scheduler. The practical effect of scheduler refinement is that a correct program written with a particular scheduling discipline in mind is also correct if the scheduling discipline is changed to any refinement of the original scheduler. To see this, it is sufficient to note that any state that the refined monitor enters is a possible state of the monitor from which it is refined; thus, if the refined monitor enters an erroneous state, the monitor from which it is refined could have done the same. (As pointed out in Section 4, the question of termination is not explicitly considered. However, the transformations presented below do not introduce liveness problems.)

In Figure 4, solid arrows between monitor kinds indicate scheduler refinement. Since scheduler refinement is a transitive property, it is always possible to "follow the arrows" (towards more refined schedulers) without modifying the monitor program. For example, a program written to work with a priority quasi-blocking scheduler works without modification with either a priority blocking or priority nonblocking scheduler.

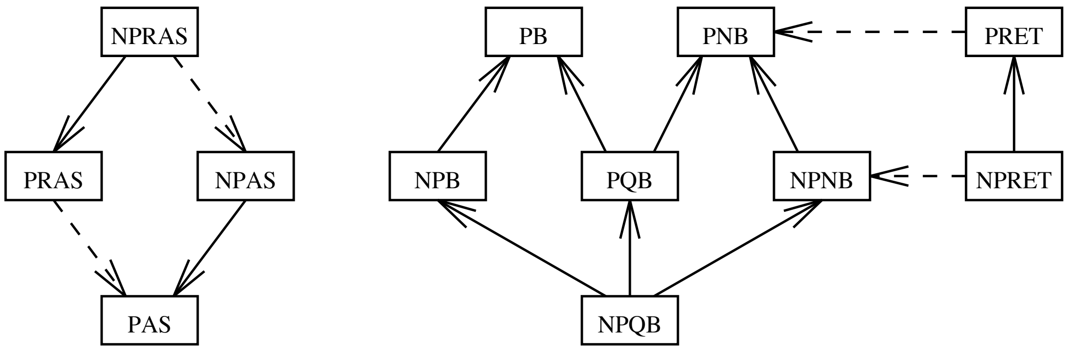


Figure 4: The refinement relationships between monitor schedulers. Solid arrows indicate scheduler refinement; dashed arrows indicate code refinement.

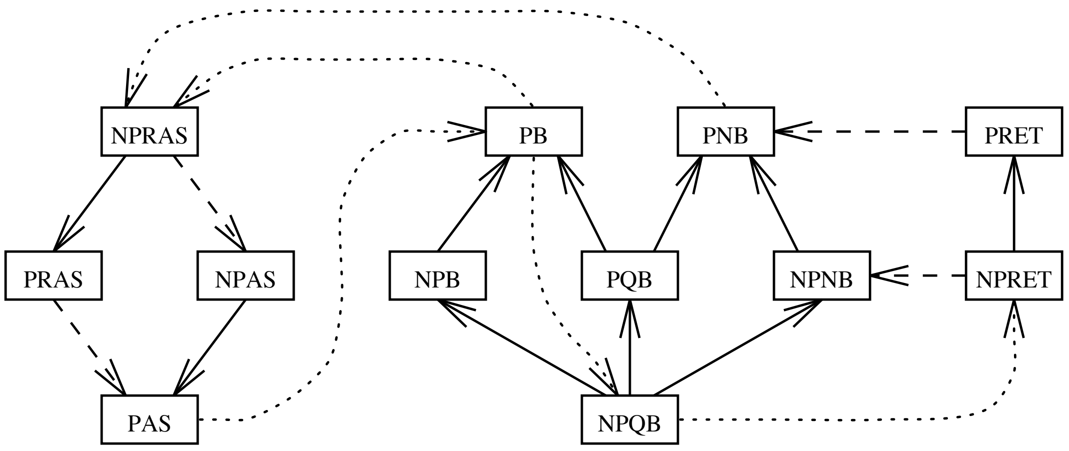


Figure 5: Transformations between monitors. Dotted arrows represent non-identity transformations.

In addition to the identity transformations that result from refining the scheduler, additional identity transformations arise from placing restrictions on the program code. Extended immediate-return monitors are restricted versions of nonblocking monitors, and restricted automatic signal monitors are restricted versions of general automatic-signal monitors. A program written using a priority extended immediate return monitor works without modification with a priority nonblocking monitor. The dashed arrows in Figure 4 indicate code refinement.

In other cases, where a refinement relationship does not hold between schedulers, monitor programs often must be modified to work correctly. However, the modifications can be done while maintaining the overall program structure. To demonstrate this, it is necessary to show how monitor programs written for any scheduler can be transformed to execute correctly with any other scheduler. Figure 5 illustrates our strategy for accomplishing this. The identity transformations are shown as solid and dashed arrows, as in Figure 4. Dotted arrows represent non-identity transformations. Since any kind of monitor can be reached from any other by following a path through the directed graph, any monitor program can be transformed into a valid program for any kind of monitor. The next few sections demonstrate that the transformations represented by dotted arrows exist.

5.1 PAS using PB

The transformation would be relatively straightforward if it were possible to evaluate any `wait` condition from any monitor entry: before a task unlocks the monitor, code could be inserted to check all `wait` conditions and signal a task if some condition were true. (This strategy is used by Howard [1976b]; however, he does not address the problem of waiting on conditions that cannot be evaluated from other monitor entries and thus his transformation works only for restricted automatic signal monitors.) To transform general PAS monitors to use explicit signals, the simplest approach is to have each task check its own `wait` condition. Unfortunately, this involves repeatedly waking all waiting tasks to

PAS operation	PB operation
(inserted)	d : condition $W_d \equiv$ **true**
enter	**enter**

wait W_q	do **not** $W_q \rightarrow$
	    signal d
	    if **not** $W_q \rightarrow$ **wait** d
	    [] $W_q \rightarrow$ **skip**
	    fi
	od

return	**signal** d; **return**

Figure 6: Transforming a general PAS monitor to a PB monitor.

let them check their conditions, a very inefficient process.

Figure 6 contains the details of the transformation. The condition variable d is used to block all waiting tasks. When a task unlocks the monitor, d is signalled. Each task in turn checks its condition and either proceeds (if the condition holds) or signals the next task on d and then waits again. Since signallers are blocked on the signaller queue, the signalling stops when d becomes empty; at this point the tasks awaken from the signaller queue and again wait on d.

5.2 PB using NPQB

This transformation illustrates the difficulty of controlling task-execution order precisely in NPQB monitors. The idea behind the transformation is simple: follow every **signal** by a **wait** on an explicit signaller queue and then signal the explicit signaller queue before vacating the monitor. The complexity in the transformation shown in Figure 7 is mostly due to the fact that it is necessary to record what is to be done before actually doing it. Thus it is necessary to introduce variables to record the "virtual" length and number of signals pending for each condition variable. These variables must be updated explicitly before signal or wait is executed.

For each condition q in the PB monitor, a variable $\langle q\rangle$ is introduced to represent the virtual length of q; all occurrences of $|q|$ in the PB monitor are replaced by $\langle q\rangle$ in the NPQB monitor. Similarly, $\langle w.q\rangle$ represents the number tasks (virtually) waiting on q that have been signalled. The condition t is introduced to delay signallers, and the condition f is used to delay entering tasks. After signalling, a task waits on condition t until there are no signals pending ($\langle w\rangle = 0$). The variable $\langle s\rangle$ represents the virtual length of the signaller queue (including those tasks blocked on either s and t) and is substituted for the variable $|s|$ in the original program. When a task enters the monitor, it waits on condition f until the signaller queue is empty ($\langle s\rangle = 0$) and there are no pending signals on any condition queue ($\langle w\rangle = 0$). (Since the proof rules for a PB monitor do not mention $|w.q|$, the transformed versions of the original predicates does not mention $\langle w.q\rangle$.) Note that after these substitutions are made, none of the predicates mentions the variables $|q|$ or $|w.q|$ for any queue q.

5.3 NPQB using NPRET

To simulate a no-priority quasi-blocking monitor using a no-priority extended immediate-return monitor, a **wait** must be inserted after every **signal** so the signalling task does not leave the monitor. Thus an extra condition variable t is introduced to block signallers. The condition variable t is signalled immediately before **wait** and **return**. Figure 8 contains the details of this transformation.

5.4 PNB using NPRAS

Since an NPRAS monitor does not have explicit signals (but does have explicit condition variables), a variable $\langle w.q\rangle$ is introduced to simulate signals. All uses of $|w.q|$ in the PNB proof outline are changed to use $\langle w.q\rangle$ and **wait** q is translated to waiting until $\langle w.q\rangle$ is positive. In addition, new tasks must be prevented from jumping ahead of other

PB operation	NPQB operation
q : condition	q : condition; $\langle q\rangle, \langle w.q\rangle$: integer
(inserted)	f, t : condition $\langle w\rangle, \langle s\rangle$: integer

enter	**enter**
	if $\langle s\rangle > 0$ **or** $\langle w\rangle > 0 \rightarrow$ **wait** f
	[] $\langle s\rangle = 0$ **and** $\langle w\rangle = 0 \rightarrow$ **skip**
	fi

signal q	$\langle s\rangle := \langle s\rangle + 1;$ **if** $\langle q\rangle > 0 \rightarrow$ $\langle q\rangle := \langle q\rangle - 1$ $\langle w.q\rangle := \langle w.q\rangle + 1$ $\langle w\rangle := \langle w\rangle + 1$ **signal** q [] $\langle q\rangle = 0 \rightarrow$ **skip** **fi** **if** $\langle w\rangle > 0 \rightarrow$ **wait** t [] $\langle w\rangle = 0 \rightarrow$ **skip** **fi** $\langle s\rangle := \langle s\rangle - 1$

wait q	$\langle q\rangle := \langle q\rangle + 1$
	if $\langle s\rangle > 0 \rightarrow$ **signal** t
	[] $\langle s\rangle = 0 \wedge \lvert f\rvert > 0 \rightarrow$ **signal** f
	fi
	if $\langle w.q\rangle = 0 \rightarrow$ **wait** q
	[] $\langle w.q\rangle > 0 \rightarrow$ **skip**
	fi
	$\langle w.q\rangle := \langle w.q\rangle - 1$

return	**if** $\lvert t\rvert > 0 \rightarrow$ **signal** t
	[] $\lvert t\rvert = 0 \rightarrow$ **signal** f
	fi
	return

Figure 7: Transforming a PB monitor to a NPQB monitor.

NPQB operation	NPRET operation
(inserted)	t : condition
enter	**enter**
signal q	**signal** q; **wait** t
wait q	**signal** t; **wait** q;
return	**signal** t; **return**

Figure 8: Transforming a NPQB monitor to a NPRET monitor.

PNB operation	NPRAS operation
$\mathrm{q_i}$: condition	$\langle \mathrm{w.q_i} \rangle$: integer := 0 $\mathrm{q_i}$: **condexpr** ($\langle \mathrm{w.q_i} \rangle > 0$)
(inserted)	$\langle \mathrm{w} \rangle$: integer := 0 f : **condexpr** ($\langle \mathrm{w} \rangle = 0$)
enter	**enter**; **wait** f
signal q	**if** $\lvert q \rvert > 0 \rightarrow$ $\lvert q \rvert := \lvert q \rvert - 1$; $\langle \mathrm{w.q} \rangle := \langle \mathrm{w.q} \rangle + 1$ $\langle \mathrm{w} \rangle := \langle \mathrm{w} \rangle + 1$ [] $\lvert q \rvert = 0 \rightarrow$ **skip** **fi**
wait q	**wait** q; $\langle \mathrm{w.q} \rangle := \langle \mathrm{w.q} \rangle - 1; \langle \mathrm{w} \rangle := \langle \mathrm{w} \rangle - 1$;
return	**return**

Figure 9: Transforming a PNB monitor to a NPRAS monitor.

ready tasks. This is accomplished by introducing a condition f to delay entering tasks until there are no pending signals. (In other words, until $\langle \mathrm{w} \rangle = 0$.) Figure 9 contains the details of this transformation.

5.5 PB using NPRAS

This transformation is similar to the one above; however, signallers must be blocked until all signalled tasks have unlocked the monitor. This is accomplished by introducing an explicit signaller queue t that is used to block signallers until the number of pending signals is zero. All uses of $\lvert s \rvert$ in the PB monitor are replaced by $\lvert t \rvert$ in the transformed monitor. As in the previous section, the queue f is used to prevent newly entering tasks from jumping ahead of ready tasks. Figure 10 contains the details of the transformation.

6 Monitor Performance

Since monitor access is serialized, monitors inhibit concurrency and are potential system bottlenecks. This makes it particularly important that monitors be efficient. To gain some insight into the relative performance of different kinds of monitors, the response time of various kinds of monitors was measured under various conditions.

6.1 General Structure of the Experimental Testbed

The general structure of our experimental testbed is illustrated in Figure 11. The testbed is a simple queuing system with two servers. The number of tasks in the system (N) is fixed. The gate keeper is a server with exponential service time; the average service time of the gate keeper (S_g) can be varied to provide different loads to the monitor. The other server is the monitor to be studied. Measuring the average response time (R_m) of the monitor as S_g is varied allows us to draw inferences about the monitor.

The general behaviour of such systems is well understood. When S_g is very small, the gate keeper releases tasks quickly, and very few tasks are queued at the gate keeper. In this region, R_m is large since almost all the tasks are queued waiting to enter the monitor. When S_g is very large, the queue at the monitor is short; thus R_m is small. Somewhere between the two extremes, a transition occurs, called the *saturation point*.

Other interesting quantities can be derived from this data. For example, for any server, the average response time (R), the average throughput (X), and the average number of tasks ($\bar{n}$) are related by a well known formula

$$X = \bar{n}/R \qquad \text{(Little's Law)}. \tag{1}$$

In particular, this applies to the monitor, which allows us to derive the throughput of the monitor under heavy load as follows. The heaviest load on the monitor occurs when R_g is very small. In this state, nearly all the tasks are in, or

PB operation	NPRAS operation
q_i : condition	$\langle w.q_i \rangle$: integer := 0 q_i : **condexpr** $(\langle w.q_i \rangle > 0)$
(inserted)	$\langle w \rangle$: integer := 0 t : **condexpr** $(\langle w \rangle = 0)$ f : **condexpr** $(\langle w \rangle = 0$ **and** $\lvert t \rvert = 0)$
enter	**enter**; **wait** f
signal q	**if** $\lvert q \rvert > 0 \rightarrow$ $\quad \lvert q \rvert := \lvert q \rvert - 1$; $\quad \langle w.q \rangle := \langle w.q \rangle + 1$ $\quad \langle w \rangle := \langle w \rangle + 1$ [] $\lvert q \rvert = 0 \rightarrow$ **skip** **fi** **wait** t
wait q	**wait** q $\langle w.q \rangle := \langle w.q \rangle - 1$ $\langle w \rangle := \langle w \rangle - 1$
return	**return**

Figure 10: Transforming a PB monitor to a NPRAS monitor

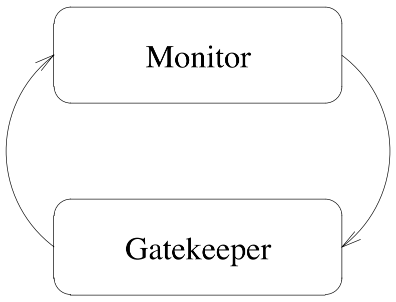


Figure 11: Experimental testbed

waiting to enter, the monitor; i.e., $\bar{n}_m \simeq N$. Thus, for small values of R_g,

$$\begin{aligned} X_m &= \bar{n}_m / R_m && (2) \\ &\simeq N/R_m. && (3) \end{aligned}$$

Since N is a known constant and R_m under heavy load has been measured, the throughput of the monitor under heavy load can be easily calculated.

6.2 Language Considerations

Programming languages typically provide only one kind of monitor; therefore, performance comparisons among different monitors are complicated by the need to control all of the inter-language factors. To avoid this problem, all of the monitors described here (except the restricted automatic-signal monitor) have been implemented in C [Kernighan and Ritchie 1988], using a concurrency kernel that supports light-weight tasks on uniprocessors and multiprocessors [Buhr and Stroobosscher 1990]. The monitors are implemented by a preprocessor [Fortier 1989] that converts the monitor statements into appropriate declarations of semaphores and calls to P and V operations using these semaphores. The simulation using semaphores is faithful to directly implemented monitors in that the amount and duration of blocking is identical. While a direct implementation might affect the absolute performance, the relative performance amongst the monitors does not change because the duration of blocking establishes almost all the differences in performance. Our monitors are traditional in design except for the ability to specify the kind of the monitor, the ability to determine the number of blocked tasks on a condition variable, and a unique way of marking tasks on condition queues to aid in programming.

A monitor is introduced by a **uMonitor** statement, as in:

```
uMonitor name ( kind ) {
    /* monitor declarations */

    /* monitor routines */
}
```

The current implementation is a package with no visibility control. Therefore, a programming discipline of *not* accessing monitor variables or internal routines outside of the monitor must be used to guarantee that the monitor functions correctly. The kind of monitor is specified using one of the new monitor names from Table 4 (e.g., **uMonitor** RW(**PriorityBlocking**)). There are two kinds of routines allowed in a monitor: **uEntry** routines, which obtain the monitor lock and are called from outside the monitor, and **uLocal** routines, which do not acquire the monitor lock and are called from within the monitor. There is a type **uCondition** for declaring condition variables (arrays of conditions and pointers to conditions are supported) and the condition queues are FIFO. The content of a condition variable is private; it is *not* meaningful to read or to assign to a condition variable, except for initialization, or to copy a condition variable (e.g. pass it as a value parameter), or to use the condition variable outside of the monitor in which it is declared. There are two forms of **wait** statement, for explicit-signal and automatic-signal monitors, respectively. The **uWait** statement is used in explicit-signal monitors to wait on a condition variable. The **uWaitUntil** statement is used in automatic-signal monitors to wait until a conditional expression is true. The **uSignal** statement is used in explicit-signal monitors to signal a condition. The precise implementation of **uSignal** depends on the kind of monitor specified in the **uMonitor** statement. The routine **uCondLength**(condition_variable) returns the number of tasks blocked on the condition_variable. (Turing [Holt 1992, p. 118] provides a similar capability with an empty routine that returns true if the condition variable has no tasks waiting on it and false otherwise.) An & must precede the condition_variable name because condition variables must always be passed by reference.

The explicit-signal **uWait** statement allows an optional value to be stored with each task on a condition queue. This is done by including a value after the condition, for example:

```
uWait condition with integer_expression
```

(This value must not be confused with Hoare's argument to a condition, which is a priority for use in subsequent selection of a task when the condition is signalled [Hoare 1974, p. 553].) The integer value can be accessed by other tasks using the routine **uCondFront**(condition_variable), which returns the value from the first blocked task on condition_variable. If no value is specified, a default value of 0 is assumed. This information can be used to provide more precise information about a waiting task than can be inferred from its presence on a particular condition variable. For example, the value of the front task on a condition can be examined by a signaller to help make a decision about which condition variable it should signal next. (This facility is mimicked by creating and managing an explicit queue in the monitor that contains the values. However, since the condition variable already manages a queue, it is convenient to allow users to take advantage of it. The usefulness of this facility is demonstrated in the next section.)

6.3 Test Problem

We chose the readers and writer problem because it has moderate complexity and is a fairly representative use of a monitor. The readers and writer problem deals with controlling access to a resource that can be shared by multiple readers, but that only one writer at a time can use (e.g., a sequential file). While there are many possible solutions to this problem, each solution must be fair in the face of a continuous stream of one kind of task arriving at the monitor. For example, if readers are currently using the resource, a continuous stream of reader tasks should not make an arriving writer task wait forever. Furthermore, a solution to the readers and writer problem should provide FIFO execution of the tasks so that a read that is requested after a write does not execute before the write, thus reading old information. This phenomenon is called the *stale readers* problem. Two solutions using P and V primitives are given by Courtois et al [1971], but both solutions have a fairness problem that may result in an unbounded wait for one of the kinds of tasks; as well, there is non-FIFO execution of tasks. Hoare's [1974] monitor solution is fair to both kinds of tasks, but has non-FIFO execution. (There are many other published solutions.)

An additional difficulty arises when comparing different kinds of monitors: different kinds of monitors suggest different coding styles, and may support certain coding styles more efficiently. This is true in the case of the readers and writers problem. To avoid any bias, we tested two solutions to the readers and writer problem. Both service readers and writers in FIFO order; hence there is neither a fairness problem nor a stale read problem. The two solutions presented differ from one another in the coding style that is prompted by the blocking or nonblocking nature of a signal statement.

A blocking signal tends to suggest a style where a signaller delegates responsibility for clean up and for further signalling to the signalled task. This style is suggested because the signalled task is (usually) the next to execute, and so the signalled task should perform any work needed to be done next. For example, in the readers and writer problem, when the last reader of a group of readers or a writer leaves the monitor, it checks the front of the condition queue and, if possible, signals another task that may restart execution. If the signalled task is a reader, that reader checks the front of the queue and signals at most one other reader task, which in turn, may signal more if that is appropriate. Each reader is delegated the responsibility to start another reader on the front of the condition queue.

A nonblocking signal tends to suggest a different coding style, where a signaller takes responsibility for clean up and for signalling other tasks that can execute. This style is suggested because the signaller continues execution after the signal, and so it can perform any necessary work before the signalled tasks enter the monitor. For example, in the readers and writer problem, the last task to use the resource signals as many tasks as is appropriate at the front of the condition queue. Thus, when a writer task is finished with the resource, it will potentially signal multiple reader tasks, which then do no further signalling.

The coding styles suggested by the blocking and nonblocking signal are called coding styles 1 and 2, respectively. Notice, that these coding styles are suggested, but not required, by the kind of signal. For many monitors, the same coding style works regardless of the kind of monitor. Nevertheless, both coding styles are worthy of examination.

Monitor solutions for the readers and writer problem in both coding styles appear in appendix A. The monitors do not actually do the reading and writing, they act as controls to delay tasks if the resource is busy. Notice the use of the optional value stored with each waiting task to distinguish between reader and writer tasks on the same condition variable. The task in the monitor examines this information for the task on the front of a condition queue and makes signalling decisions based on it.

6.4 Structure of the Experiment

Four monitor types were tested: blocking (B), nonblocking (NB), quasi-blocking (QB), and extended immediate return (RET). The general automatic-signal monitor was not tested because its execution performance is not comparable to the explicit-signal monitors (10-50 times slower). This poor performance is because of the need to wake up (potentially all of) the tasks in the monitor to recheck their conditional expression. As well, to test the coding style, both solutions to the readers and writer problem, coding style 1 ($CS1$) and coding style 2 ($CS2$), were tested. Finally, the priority and no-priority versions of each monitor were tested. All combinations of monitor, priority and coding style were tested, leading to 16 experiments in all. The experiments were run in a random order to avoid any possible interactions.

The experiments were run on a multiprocessor (Sequent Symmetry) with a timer that was accurate to 1 microsecond. This timer allows accurate timings of short events, such as the time for a task to enter and exit from the monitor. The tests were run stand-alone with only the minimal operating-system processes running; nevertheless, a small amount of interference did occur. Interference occurred when the operating system would preempt one or more of the processors during execution of an experimental trial for some period of time. Fortunately, interference was extremely rare, resulting in only a few outlying values that did not influence the results.

The experiment is a simple simulation system illustrated by Figure 12. A pool of 250 worker tasks are started and each blocks behind a barrier controlled by a gate keeper. The gate keeper acts as a server with service time that is exponentially distributed with mean S_g. The simulation has two phases: the first phase brings the system to a steady state and the second phase has the workers determine the length of time to be serviced by the monitor. Steady state is reached by the gate-keeper opening the gate 3,000 times. The value 3,000 was determined by a pilot experiment in which the R_m was plotted as a function of time for different values of S_g to see when the response times reached a steady state. Then the gate-keeper sets a shared variable so that the worker tasks start recording response times. The gate is then opened 5,000 more times and then the shared variable is reset causing the workers to stop recording

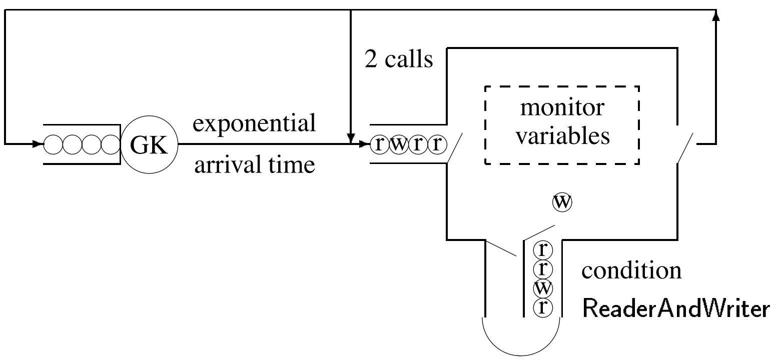


Figure 12: Simulation System

response times. The simulation is then shutdown and all the response times are collected from the worker tasks. There may be times when the gate is opened and no worker enters the system; this is because all worker tasks may be in the simulation system, i.e., the simulation is a closed system. This situation happens when the gate-keeper is opening the gate faster than the service time of the monitor. The value 5,000 was chosen because it was large enough to get over 2,000 observations when the value of S_g was small, and yet it was small enough so that the simulation did not take too long when the value of S_g was large.

The results of a few pilot experiments showed that the different monitors saturated (the point where the monitor service time equals the arrival rate) in the range S_g = {300-600} microseconds. Detailed statistical analysis was not done in this range because of the large variance there, which would mask results on either side where the behaviour is stable. Therefore, it was decided to analyze in detail the data for S_g = {100-250} and S_g = {850-1,000} because these areas were outside of the large variance region, and contained S_g values that were not so small as to preclude accurate measurement and not so large that the simulation took too long. This resulted in two distinct analyses: pre- and post-saturation.

Once released by the gate-keeper, a worker task randomly chooses between being a reader or writer with probability RW, and then makes a pair of calls to the monitor. The probabilities tested were RW = {50%, 70%, 90%}, which we believe represent a broad range of realistic situations.

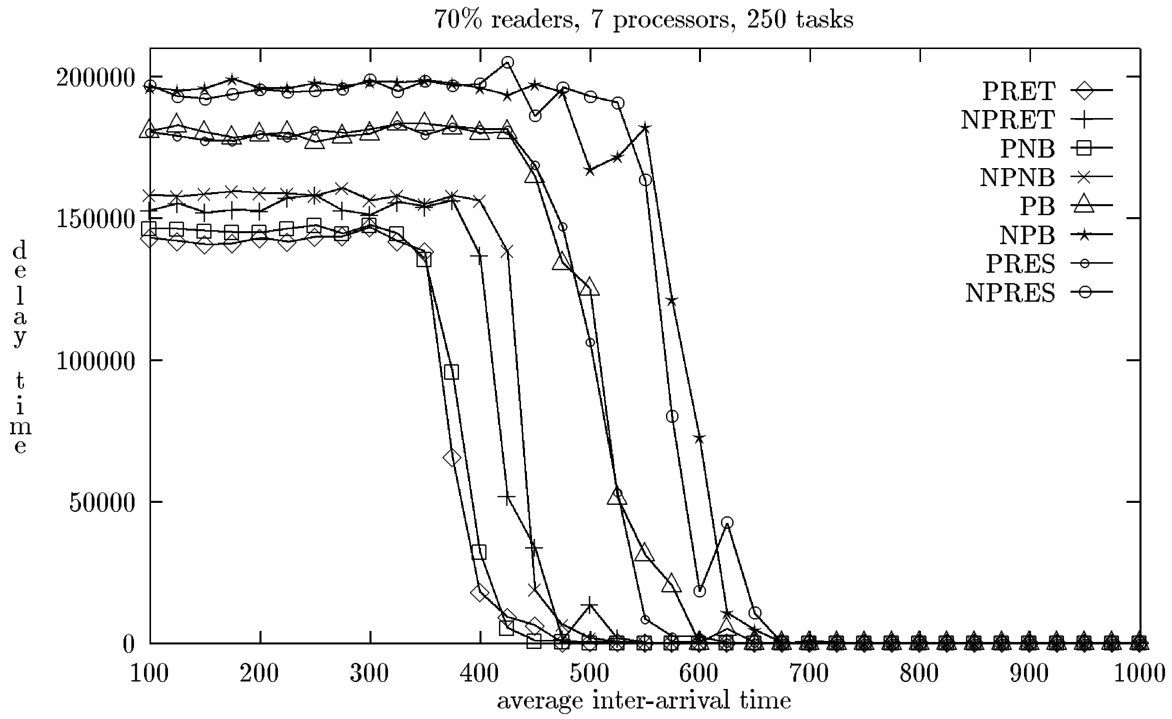
The time taken for the calls to StartRead and EndRead or to StartWrite and EndWrite was measured. This time represents the time that a task is delayed in gaining access to the particular kind of monitor in the experiment. In a real system, there would be an additional delay between the two monitor calls to simulate the actual access to a shared resource. However, this time is independent of the kind of monitor, and would have extended the simulation time without providing additional information.

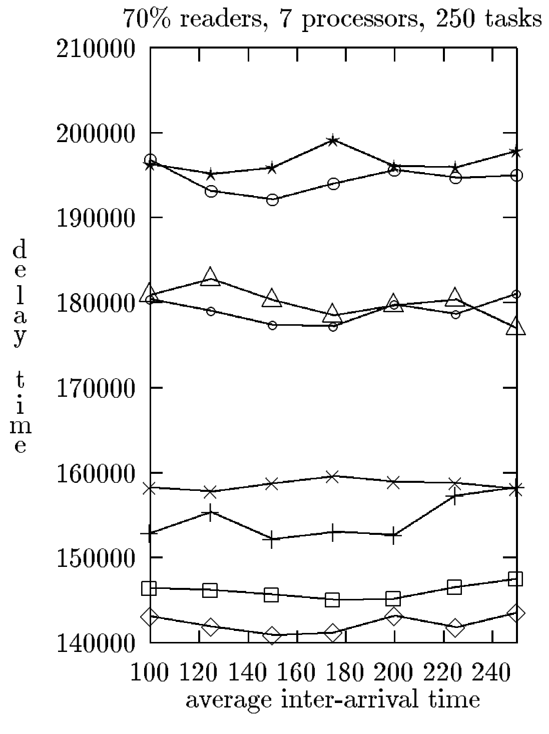
Finally, there were P physical processors involved in executing the simulation to determine if the number of processors affects the phenomenon we are examining. The number of processors tested were P = {5, 7}. The values 5 and 7 were chosen because they are representative of the number of processors available on many shared-memory multiprocessor computers.

Eight trials were run for each combination of values $\{P,\ RW,\ S_g\}$. The 8 trials mitigated the small amount of interference from the underlying operating system during execution. The order in which the trials were run was randomized to avoid any interaction with the operating system.

As an illustration of the overall behaviour of the different kinds of monitors when executing around their saturation points, Figures 13 and 14 show a graph of coding style 1 and 2, respectively, for the tuples {5, 70, 100–1,000 by 25}. The top graph is the complete experiment and the bottom graphs are the regions S_g = {100–250} and S_g = {850–1,000} of the top graph. The graphs show the classic transition of a queuing system when driven to saturation. When a monitor can no longer service requests faster than they are arriving, the tasks begin to spend large amounts of time waiting to enter the monitor. Because the system is closed, the response times reached a plateau with a maximum at 250 times the average service time of each monitor (plus a constant for the simulation system overhead). Increasing or decreasing the number of worker tasks simply has a linear effect on the level of the plateau.

All combinations of the above variables were tested, leading to a 4 (monitor type) x 2 (priority) x 2 (coding style) x 7 (average arrival) x 2 (processors) x 3 (percent readers) complete factorial experimental design. Data was analyzed using analysis of variance by standard statistical software (SAS, program ANOVA).





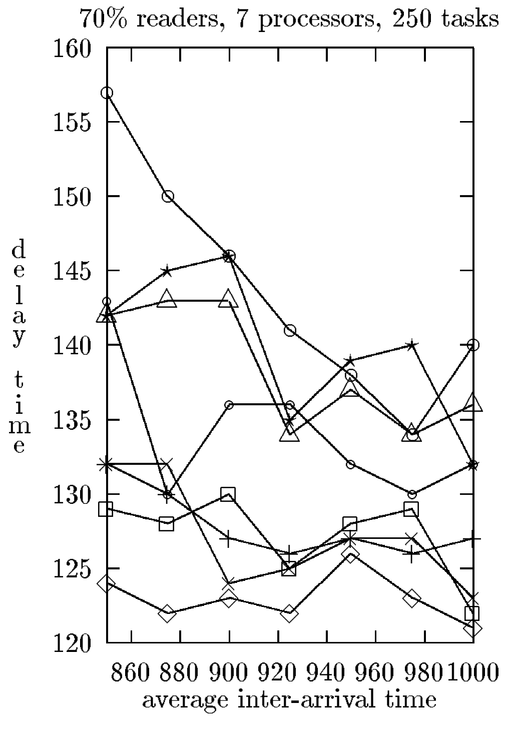
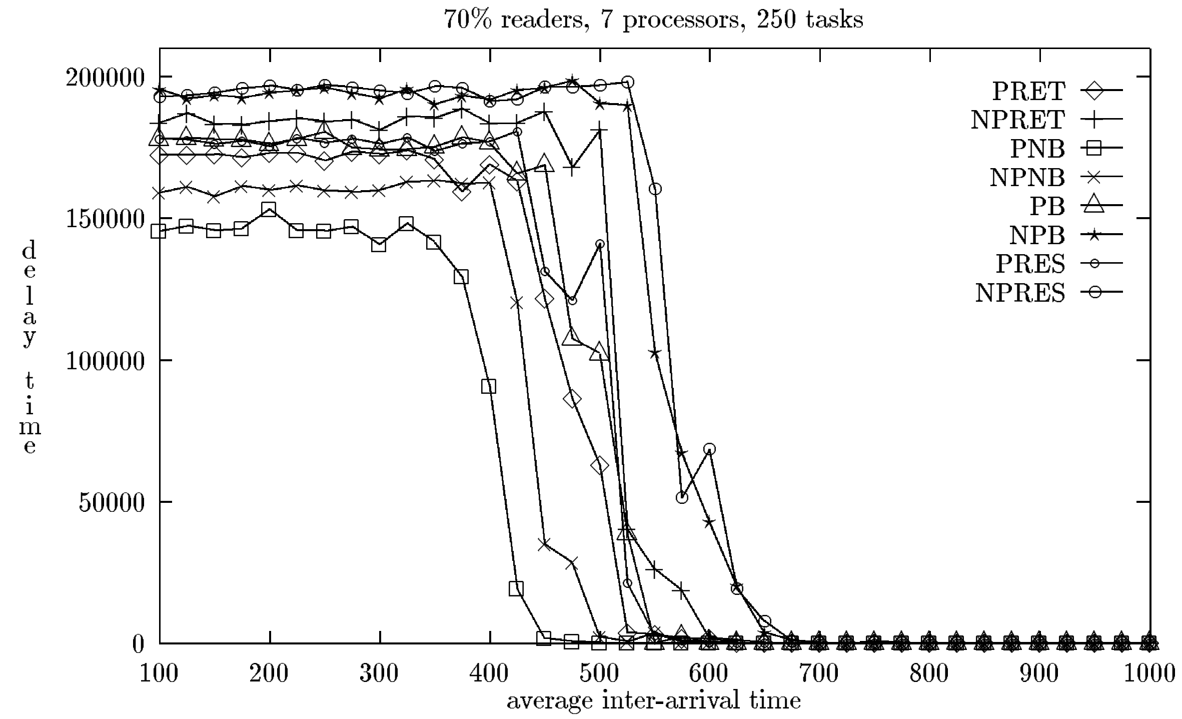
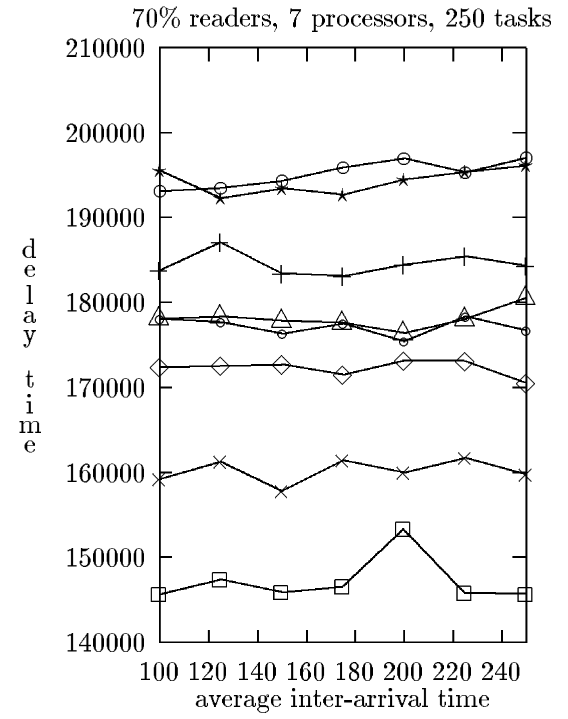


Figure 13: Coding Style 1, Representative Graph (times in microseconds)





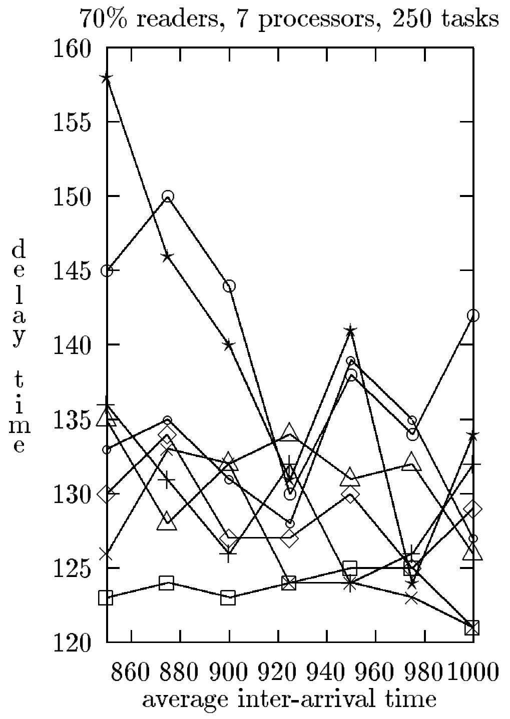


Figure 14: Coding Style 2, Representative Graph (times in microseconds)

Because the distribution of response times, R_m, was highly skewed, the analysis was done on the transformed statistic $ln(ln(R_m))$. This transformation is a common statistical method that has the effect of transforming a skewed distribution into a distribution that is more nearly normal making it amenable to standard statistical tests. Further, because of the several thousand degrees of freedom in the error term, it was possible to detect extremely small differences among groups. However, we considered statistical differences of less than 1% to be insignificant from a practical standpoint.

6.5 Sources of Differences

The following sections present a discussion of the effect of different kinds of monitors on the readers and writer problem. This discussion is applicable to any problem where a group of tasks must wait for a resource to become free. Tasks requesting use of the resource are designated by R for a read request and W for a write request. In the following tables, context switches occur between the lines.

6.5.1 Priority Blocking Signal

This section describes the behaviour of monitor coding styles 1 and 2 when using a blocking signal. The first monitor examined is the priority-blocking (Hoare) monitor. Table 5 shows one particular situation that might arise during the execution of monitor coding style 1. In this scenario, a writer, W_0, has just finished using the resource and has called the monitor to release the resource. During W_0's use of the resource, a queue of tasks has formed to use the monitor, waiting on condition variable ReaderAndWriter.

exit	**Monitor**	ReaderAndWriter **Condition Queue**	**Signaller Queue**
	W_0	$R_1, R_2, R_3, R_4, R_5, W_1, \dots$	$\emptyset$
	R_1	$R_2, R_3, R_4, R_5, W_1, \dots$	W_0
	R_2	$R_3, R_4, R_5, W_1, \dots$	W_0, R_1
	R_3	$R_4, R_5, W_1, \dots$	W_0, R_1, R_2
	R_4	$R_5, W_1, \dots$	W_0, R_1, R_2, R_3
R'_5	R_5	$W_1, \dots$	W_0, R_1, R_2, R_3, R_4
W'_0	W_0	$W_1, \dots$	R_1, R_2, R_3, R_4
R'_1	R_1	$W_1, \dots$	R_2, R_3, R_4
R'_2	R_2	$W_1, \dots$	R_3, R_4
R'_3	R_3	$W_1, \dots$	R_4
R'_4	R_4	$W_1, \dots$	$\emptyset$

Table 5: Priority Blocking Signal, Monitor Coding Style 1

For the priority-blocking monitor, 10 context switches are needed to move tasks from the ReaderAndWriter queue to the signaller queue. During this time the signaller tasks are blocked. Further, W_0 and R_1–R_4 wait 5 context switches after they have been signalled before they can exit from the monitor and run concurrently. (A primed ($'$) task denotes the same task at some time in the future, but before the next context switch. For example, task R_5 enters the monitor and exits the monitor before the next context switch to allow task W_0 to enter the monitor.) Notice, that reader R_5, the last reader in the group of readers, exits from the monitor before readers R_1–R_4. This is because it finds a writer at the front of the ReaderAndWriter queue and therefore does not have to signal another reader that would cause it to block on the signaller queue. Hence, access to the resource is FIFO, but concurrent readers may exit in an arbitrary order. This is not a problem in the readers and writer problem, but should be noted as it could be important in other problems.

The problem of reduced concurrency for reader tasks can be mitigated simply by changing the coding style. Instead of having each reader task signal the next reader task, the last task using the resource does all the signalling, as in monitor coding style 2. Table 6 shows this scenario, using a priority-blocking monitor, during the execution of monitor coding style 2.

In this example, ten context switches are still needed to move tasks from the ReaderAndWriter queue to the signaller queue, but only W_0 waits 5 context switches before it can exit from the monitor and run concurrently. Also,

exit	Monitor	ReaderAndWriter Condition Queue	Signaller Queue
	W_0	$R_1, R_2, R_3, R_4, R_5, W_1, \ldots$	$\emptyset$
R'_1	R_1	$R_2, R_3, R_4, R_5, W_1, \ldots$	W_0
	W_0	$R_2, R_3, R_4, R_5, W_1, \ldots$	$\emptyset$
R'_2	R_2	$R_3, R_4, R_5, W_1, \ldots$	W_0
	W_0	$R_3, R_4, R_5, W_1, \ldots$	$\emptyset$
R'_3	R_3	$R_4, R_5, W_1, \ldots$	W_0
	W_0	$R_4, R_5, W_1, \ldots$	$\emptyset$
R'_4	R_4	$R_5, W_1, \ldots$	W_0
	W_0	$R_5, W_1, \ldots$	$\emptyset$
R'_5	R_5	$W_1, \ldots$	W_0
W'_0	W_0	$W_1, \ldots$	$\emptyset$

Table 6: Priority Blocking Signal, Monitor Coding Style 2

the readers exit in FIFO order. In this case, the concurrency delay has been shifted from the reader tasks to the writer task, but depending on the problem, this shift might be extremely important.

6.5.2 Priority Nonblocking Signal

Using the nonblocking signal, the execution of the above scenario for monitor coding style 1 is shown in Table 7 and the execution for monitor coding style 2 is shown in Table 8.

exit	Monitor	ReaderAndWriter Condition Queue	Signalled Queue
W'_0	W_0	$R_1, R_2, R_3, R_4, R_5, W_1, \ldots$	R'_1
R'_1	R_1	$R_2, R_3, R_4, R_5, W_1, \ldots$	R'_2
R'_2	R_2	$R_3, R_4, R_5, W_1, \ldots$	R'_3
R'_3	R_3	$R_4, R_5, W_1, \ldots$	R'_4
R'_4	R_4	$R_5, W_1, \ldots$	R'_5
R'_5	R_5	$W_1, \ldots$	$\emptyset$

Table 7: Priority Nonblocking Signal, Monitor Coding Style 1

exit	Monitor	ReaderAndWriter Condition Queue	Signalled Queue
W'_0	W_0	$R_1, R_2, R_3, R_4, R_5, W_1, \ldots$	$R'_1, R'_2, R'_3, R'_4, R'_5$
R'_1	R_1	$W_1, \ldots$	R_2, R_3, R_4, R_5
R'_2	R_2	$W_1, \ldots$	R_3, R_4, R_5
R'_3	R_3	$W_1, \ldots$	R_4, R_5
R'_4	R_4	$W_1, \ldots$	R_5
R'_5	R_5	$W_1, \ldots$	$\emptyset$

Table 8: Priority Nonblocking Signal, Monitor Coding Style 2

Notice that several tasks can be moved from a condition queue to the signalled queue without a context switch, which is why a task appears to be on two queues at the same time in Tables 7 and 8. The tasks that are moved remain blocked. There are 5 context switches needed to move tasks from the ReaderAndWriter queue to the signaller queue. Concurrency is not inhibited for the signaller and FIFO ordering is maintained for exiting readers.

6.5.3 No-Priority Scheduling Problems

No-priority scheduling makes implementation of particular scheduling schemes among tasks difficult. For example, FIFO scheduling is difficult to implement since there is no guarantee that a signalled or signaller task executes next in the monitor. When ordering is important, special care must be taken to ensure that arriving tasks that *barge* into the monitor always queue. (A task that is scheduled ahead of tasks already waiting in the monitor is referred to as a "barging task".) In general, this is accomplished by having a calling task check if there are tasks waiting in the monitor. If there are waiting tasks, the calling task waits on a special condition, which is subsequently signalled by the last waiting task (see Section 5.2). In practise, a special condition variable is unnecessary because there is usually a condition variable(s) on which the calling task can wait (e.g., the ReaderAndWriter condition). However, the order in which tasks arrive is lost if there are multiple condition variables, so it would not be possible to perform an operation on every Nth arriving task without marking it or having an additional queue.

Determining if there are tasks waiting in the monitor is non-trivial because tasks may be waiting on the internal monitor queues. Normally, it is not possible to examine the lengths of these internal queues so the monitor must keep track of this fact explicitly. For example, when the last task is signalled on the ReaderAndWriter condition, both signaller and signalled task are placed on internal queues. A barging caller will find no tasks on the condition variables, and therefore, it might use the resource. This situation can be handled by a counter variable that keeps track of the tasks on the internal queues so barging tasks can truly know if tasks are waiting in the monitor (see Section 5.2). Appendix B contains the modified versions of monitor coding styles 1 and 2 that guarantee FIFO ordering when no-priority is used (i.e. make a priority monitor from a no-priority). Fortunately, the only addition to these monitors is the setting and testing of a flag variable to know if tasks are waiting in the monitor; additional signalling and waiting is unnecessary.

6.6 Analysis of Results

The results of the analysis are divided into three regions: saturation, pre-saturation, and post-saturation. Of these, the first two are most important since most systems do not drive monitors into saturation. Briefly, the results of our experiment are as follows. There are fairly large differences among the saturation points of different monitors. There were also statistically significant differences in response times of different monitors in both pre- and post-saturation. However, the differences are fairly small.

6.6.1 Saturation

In the range $S_g = \{300–600\}$, the monitors saturated in the order blocking, quasi-blocking and nonblocking, respectively. The blocking monitors saturated approximately 30% before the nonblocking monitors. This large difference results from the effects discussed in Section 6.5, that is, blocking monitors do not release signallers as quickly as do nonblocking monitors. Hence, the nonblocking monitors can handle a faster arrival rate than the quasi-blocking and blocking monitors before saturating.

This result is important; the saturation point determines the maximum rate at which a monitor can process requests. In our experiments, nonblocking monitors could handle request rates considerably higher than quasi-blocking or blocking monitors.

6.6.2 Pre-Saturation

When S_g is much larger than the saturation point, the average queue length at the monitor is short, and thus R_m is short. The variance of R_m is moderate; most tasks do not encounter a queue, but a queue length of 1 or 2 is not uncommon. As S_g decreases and approaches the saturation point, the average queue length increases. Most tasks encounter a short queue at the monitor, but a few encounter much longer queues because of random fluctuations. The variance in queue length increases, as does the variance of the response times.

Blocking As discussed in Section 3.3, NB is always a nonblocking monitor, QB is always a quasi-blocking monitor, and B is always a blocking monitor. However, RET is a non-blocking monitor only in $CS1$ where its signals appear before returns; in $CS2$ it becomes a quasi-blocking monitor because it has a signal that does not occur before a return so it must be followed by a wait. This change between non-blocking and quasi-blocking can be seen by the different locations of the PRET and NPRET curves in $CS1$ and $CS2$ in Figures 13 and 14, respectively. We predict that the smallest response times would occur for the non-blocking monitors, followed by quasi-blocking and blocking monitors, respectively.

The ordering from lowest to highest response times for the different kinds of monitors was: $CS2/NB$, $CS1/RET$, $CS1/NB$, $CS2/RET$, $CS2/QB$, $CS2/B$, $CS1/B$, $CS1/QB$. Follow-up comparisons of differences between means, a Tukey post-hoc comparison with $p = 0.01$, found the following statistically significant groupings: $\{CS2/NB$, $CS1/RET$, $CS1/NB\}$, $\{CS2/RET\}$, $\{CS2/QB$, $CS2/B$, $CS1/B$, $CS1/QB\}$, which correspond to nonblocking, quasi-blocking and blocking, respectively. The mean response time for the nonblocking group is 2.1% less than the quasi-blocking group, and the mean response time for the quasi-blocking group is 1.6% less than the blocking group. The quasi-blocking monitors are grouped with the blocking monitors because the implementation only randomizes the selection from the signaller and waiting queues if there are intervening calling tasks; therefore, our quasi-blocking monitors behave like blocking monitors in many cases.

Coding Style The analysis in Section 6.5 suggests that the $CS2$ response times should be less than the $CS1$ response times because the signaller starts multiple tasks that execute in parallel outside of the monitor, that is, the signaller takes on the responsibility for signalling other tasks to execute. We did not feel that the full-model analysis of variance permitted a fair test of this hypothesis because the RET monitor changed from blocking to quasi-blocking between $CS1$ and $CS2$. Therefore, a separate analysis was conducted without the RET monitor.

There was a significant interaction between $CS1$ and $CS2$, $p = 0.0001$, across all trials for S_g = {850-1,000}. The mean $CS2$ monitor response time is 1.6% less than the mean $CS1$ monitor response time.

Priority The priority monitors should have lower response times than their no-priority counterparts because of the increased execution time for the no-priority monitors. This increase is the result of the extra code that is added to the no-priority monitor so that it behaves like a priority monitor for the readers and writer problem. Further, any calling tasks that barge into the monitor may prevent a signalled or signaller task from making immediate progress.

There was a significant interaction between priority and no-priority, $p = 0.0001$, across all trials for S_g = {850-1,000}. The mean priority monitor response time is 1.2% less than the mean no-priority monitors response time.

Other Effects Other statistically significant effects appeared in pre-saturation. The number of processors, P, had an effect, $p = 0.0001$, because when a group of readers is released, they execute faster with more processors. The faster the reader tasks execute, the faster they finish, so their response times are longer, which is slightly counter intuitive. The percentage of readers, RW, had an effect, $p = 0.0001$, because more writers means smaller groups of readers and longer waiting times because the writers are serialized. Thus, the response times should be shorter as the percentage of readers increased as more readers can move through the monitor more quickly. This decrease in response time was present in the data for the 90% reader experiments. The average arrival, S_g, had an effect, $p = 0.0001$, because the monitors had not completely stabilized in the range S_g = {850–1,000} so that the response times were continuing to decrease slightly. This instability can be seen in the detailed graphs of the the range S_g = {850–1,000} in Figures 13 and 14. Any effect from these variables did not affect the relationships among the monitors tested for in this experiment; therefore, it is possible to generalize the statements about monitor type, coding style and priority across these other variables.

6.6.3 Post-Saturation

In this region, the behaviour of the system is simple. All the tasks spend most of the time on the monitor entry queue or on the ReaderAndWriter condition because the gate-keeper is releasing tasks faster than the monitor service time. The simulation system is very stable in this state.

Blocking As for pre-saturation, we predict that the smallest response times would occur for the non-blocking monitors, followed by quasi-blocking and blocking monitors, respectively. The ordering from lowest to highest response times for the different kinds of monitors was: $CS1/RET$, $CS1/NB$, $CS2/NB$, $CS2/RET$, $CS2/QB$, $CS2/B$, $CS1/QB$, $CS1/B$. Follow-up comparisons of differences between means found the following statistically significant groupings: $\{CS1/RET$, $CS1/NB$, $CS2/NB\}$, $\{CS2/RET\}$, $\{CS2/QB$, $CS2/B$, $CS1/QB$, $CS1/B\}$, which correspond to nonblocking, quasi-blocking and blocking, respectively. The mean response time for the nonblocking group is 5.6% less than the quasi-blocking group, and the mean response time for the quasi-blocking group is 12.9% less than the blocking group. Again, this matches the predicted ordering, except for the quasi-blocking monitors because of our implementation.

Coding Style As for pre-saturation, we predict the $CS2$ response times should be less than the $CS1$ response times. There was a significant interaction between $CS1$ and $CS2$, $p = 0.0004$, across all trials for S_g = {100-250}. The mean $CS2$ monitor response time is 0.7% less than the mean $CS1$ monitor response time.

Priority As for pre-saturation, we predict the priority monitors should have lower response times than their no-priority counterparts. There was a significant interaction between priority and no-priority, $p = 0.0001$, across all trials for S_g = {100-250}. The mean priority monitor response time is 8.6% less than the mean no-priority monitors response time.

Other Effects The number of processors and reader percentage had an effect for the same reasons as in the pre-saturation region; however, there was no effect from average arrival because the worker tasks had stabilized in the range S_g = {100–250}.

6.6.4 Summary

While all the differences outlined above are statistically significant, most are small, except for differences among saturation points. The difference between the no-priority and priority monitors would have been significantly greater if additional signaling and waiting had been necessary to ensure FIFO ordering (only setting and testing a flag variable was needed). Hence, most concurrent programs should not show a significant performance benefit by changing the kind of monitor; only programs that are running close to the saturation point because of hardware limitations would benefit significantly from changing the kind of monitor. Therefore, in the general case, other considerations are as important, or more important, in choosing the kind of monitor for a concurrency system. It is noteworthy that the nonblocking monitors have execution times and saturation points almost identical to the immediate-return monitors but do not impose any restrictions on the location of the **signal** statement.

7 Comparison

The following is a summation of the criteria a programmer or programming language designer can use in selecting a particular kind of monitor to solve a problem or to implement in a programming language.

Proof Rules Programming-language constructs with simple semantics tend to be easier to use and less error-prone than those with complex semantics. Since proof rules are a formal specification of semantics, constructs with simple proof rules tend to be easier to use than constructs with complex proof rules. For example, the proof rules that describe the semantics of the while statement are simple compared to the proof rules describing a general goto statement. Thus one way to compare programming language constructs is to compare the complexity of their proof rules.

However, the complexity of programming language constructs is not the only consideration in choosing among them. What is important is the "power-to-weight" ratio, where "power" is expressive power of a construct and "weight" is the cost—in both programming complexity and execution time—of using the construct. For example, automatic signal monitors have a high degree of expressive power, but also a high cost in execution time. Thus the simplicity of their proof rules is offset by the cost in execution time.

An additional difficulty with comparing proof rules is that programming language constructs are often used in restricted ways—perhaps only in the context of certain "idioms". For example, a programming language that lacked high-level looping constructs, such as a while statement, must construct loops using the goto statement. In such languages, it is common practice to use goto statements only in certain stereotypical ways, such as structured programming. This approach yields programs that are much easier to understand than if goto is used in its full generality. In effect, the proof rules for goto are simplified by weakening them. The result is that programs and their proofs of correctness are simplified considerably. As has been pointed out (e.g., Sections 4.2.1, 4.2.2, and 4.2.3), the proof rules for **signal** and **wait** are often simplified in this fashion for certain kinds of monitors; this tends to simplify both programs and proofs. Thus one must be careful not to draw unwarranted conclusions from the comparison of unsimplified proof rules. (A drawback of simplified proof rules is that programs that execute correctly may not be provably correct under the simplified proof rules; thus simplified proof rules may not be adequate for verifying existing monitor programs.)

Priority versus No-Priority The proof rules for priority monitors are slightly more complex than those of no-priority monitors. However, the added complexity in priority monitors is due to significantly *weakening* the preconditions for wait and return, which makes them easier to satisfy. In our experience, the slight additional complexity is more than offset by the less stringent proof obligations for wait and return.

When a condition is signalled in a priority monitor, control transfers directly or indirectly to a task waiting for the condition to occur (unless the condition variable is empty). In either case, this transfer allows the signaller and signalled tasks to establish an internal protocol for communication through monitor variables. In contrast, a signal in a no-priority monitor, as in the programming languages Mesa, Modula-2+, and Modula-3, act as a "hint" to a waiting task to resume execution at some convenient future time. (Modula-3 takes this one step further by defining the Signal routine to wake up at least one task, which implies that it may wake up more than one task. This can be modelled in our taxonomy as a no-priority nonblocking monitor with a loop around each signal statement that executes a random number of times.) Hence, for no-priority monitors, the assertion for the condition that is signalled may no longer hold by the time the signalled task gains control of the monitor making internal protocols difficult or impossible. Thus, when a waiting task restarts, it must determine if the monitor state it was waiting for is true instead of assuming it is true because it was signalled. This often results in a wait statement being enclosed in a while loop so that a task can re-check if the event for which it was waiting has occurred (as is done implicitly in the automatic-signal monitor). This coding style can result in needless context switches when tasks awake, only to wait again, which can potentially inhibit concurrency. Lampson [1980, p. 111] attempts to disarm this criticism by saying that there are typically no tasks waiting for a monitor lock. However, as systems introduce more concurrency, particularly through light-weight tasks, and as machines introduce more processors, there will definitely be many monitors that are heavily used. Furthermore, whenever a no-priority monitor is unlocked, the programmer must ensure that the monitor invariant is satisfied in case a calling task enters the monitor.

As well, it is difficult to implement certain scheduling schemes using no-priority monitors because a calling task can barge ahead of tasks that have already been waiting in the monitor. Unless special care is taken by including extra code to deal with this situation, it is not possible to guarantee FIFO scheduling, which may be critical to a particular problem (e.g., readers and writer problem). In such cases, it is necessary to simulate a priority monitor to prevent barging (different simulation techniques are discussed in Section 6.5.3), which increases the execution cost.

Finally, there is the potential for unbounded waiting in no-priority monitors. If the implementation chooses randomly among internal queues of the same priority, there is no bound on the number of tasks that are serviced before a task is selected from a particular queue. In practise, however, this problem is usually solved by the implementation, which combines queues with the same priority so that tasks placed on them have a bounded number of tasks ahead of them. Thus, the only advantage of a no-priority monitor is that a task does not have to wait too long on average before entering the monitor.

Blocking versus Nonblocking Blocking monitors have simple rules for wait and return but more complicated rules for signal, while nonblocking monitors have more complicated rules for wait and return but very simple rules for signal.

A blocking signal guarantees that control goes immediately to the signalled task. This direct transfer is conceptually appealing because it guarantees that control transfers to a task that is waiting for a particular assertion to be true; thus, the signaller does not have an opportunity to inadvertently falsify the assertion that was true at the time of the signal. This kind of signal often results in monitor programs in which a signaller delegates all responsibility for clean up and for further signalling to the signalled task.

A nonblocking signal leaves the signaller in control of the monitor until it has unlocked the monitor. This kind of signal often results in a monitor where a signaller takes on the responsibility for clean up and for signalling other tasks that can execute. As suggested by Howard [1976b, p. 52] and as we have also observed, this form of blocking is what programmers naturally think of. For a nonblocking signal, the signaller is obligated to establish the proper monitor state only when the monitor is unlocked; the monitor state at the time of the signal is irrelevant. In the situation where the signaller performs multiple signals, it is possible for one of the signalled tasks to subsequently alter the monitor state for the other signalled tasks. A similar problem exists for the blocking signal as it is possible for the signalled task to subsequently alter the monitor state prior to resumption of the signaller. Fortunately, both situations can be handled by judicious manipulation of the monitor data by the signalled tasks.

The main differences between the two kinds of monitors are coding problems and performance. First, in the blocking monitor, a signal before a return or wait involves additional runtime overhead because of the context switch to the signalled task and subsequently back to the signaller before the **wait** or **return** can occur; this is not the case for a nonblocking monitor. Second, in the blocking case, the signalled task cannot restart the signaller task from the condition on which it will eventually wait because the signaller is currently on the signaller queue. If the signalled task signals the condition that the signaller will eventually wait on, either another task is restarted or the signal is lost because the condition is empty; the latter situation could result in problems because the signaller may not be restarted. For example, the straightforward approach to communicating data between two tasks in the monitor:

$task_1$	$task_2$
msg = ...	wait MsgAvail
signal MsgAvail	print msg
wait ReplyAvail	reply = ...
print reply	signal ReplyAvail

fails for a blocking monitor because $task_2$'s signal of condition ReplyAvail is lost. In the nonblocking case, the signaller blocks on the condition before the signalled task starts so the signalled task knows that it can restart its signaller if that is appropriate. The opposite problem can also occur for a blocking monitor, where a signaller never stops execution because the signalled tasks wait again on the same queue being signalled. For example, in the extreme case of using signals as hints, a signalled task might wake up all tasks waiting on a condition when an assertion for a condition is true, as in:

$signaller\ task$	$waiting\ tasks$
while (!empty(c)) signal c;	while (...) wait c;

However, after a waiting task restarts and falsifies the assertion, the signaller task loops forever because any waiting tasks continually put themselves back onto the condition variable. For a nonblocking monitor, the signalled tasks are moved to the waiting queue before the monitor is eventually unlocked, which guarantees that the condition variable becomes empty. Finally, in the blocking case, a **signal** before a **return** inhibits concurrency because the signaller remains blocked while the signalled task is executing in the monitor. In the nonblocking case, the signaller is allowed to return from the monitor immediately so it can execute concurrently with the signalled task (but outside of the monitor). On a uniprocessor, the extra concurrency would be noticed only as decreased turn-around time. With multiprocessors, the extra concurrency would be noticed as increased throughput. It might be possible to mitigate the problem of a blocking signal before a return by having the compiler optimize that signal into a nonblocking signal, but this changes the semantic of the monitor.

Coding Style The blocking and nonblocking monitors suggest two different coding styles and both have their place. In the readers and writer problem, coding style 2 (signaller is responsible for signalling other tasks) produced a slight performance advantage. This advantage occurred because inhibiting concurrency for a writer task to increase concurrency for readers was the best way to speed up a system where there are more readers than writers. It is possible to imagine scenarios where inhibiting concurrency for the signaller task would increase delays in the system, for example, if the signaller task was controlling other critical aspects of the system. One drawback to coding style 1 was in releasing readers in non-FIFO order for the blocking monitor. In a situation where the release order was important, this subtle variation in release order would be very difficult to notice.

Quasi-blocking versus Blocking and Non-Blocking Quasi-blocking monitors have the most complicated proof rules; their rules for **signal** are similar to blocking monitors, while their rules for vacating the monitor are similar to nonblocking monitors (the worst of both worlds).

An advantage of this monitor over the blocking is that signaller tasks do not have to wait until there are no more signalled tasks to resume execution. The signaller tasks can leave the monitor more quickly instead of being blocked for a longer time. However, since the order of execution of the signaller and signalled tasks is unknown, an internal protocol between them can be difficult to establish. Therefore, we do not believe this kind of monitor to be particularly useful, even though it appears implicitly in some situations such as operating system kernels.

Extended Immediate Return The proof rules for extended immediate-return monitors are identical to those for non-blocking monitors. Thus, from the standpoint of proof complexity, they have the same advantages and disadvantages, relative to other monitors, as non-blocking monitors.

The immediate-return monitor was invented to optimize the signal–return case that occurs frequently in monitors at the cost of restricting the ability to write certain monitor algorithms. The extended immediate-return monitor allows any monitor to be written, with the restriction that a signal must appear before a wait or return statement. When a system is not saturated, the extended immediate-return monitor performs slightly better than the nonblocking monitor. The same is true at and after saturation for the signal–return case, but not for the quasi-blocking case. (The reason the PRET monitor performed better than the PNB when all signals occur before returns is that the PNB monitor must check the waiting queue before returning, which results in more time spent in the monitor.) But most importantly, the extended immediate-return monitor mandates a particular coding style, signals before waits and returns, which may unnecessarily complicate the coding of a monitor and obscure its algorithm.

Automatic Signal Automatic-signal monitors have the simplest proof rules, and in our experience, automatic signal monitors are easier to use than explicit signal monitors. Eliminating the signal eliminates two common mistakes in explicit-signal monitor programs: performing unwarranted signals and not performing necessary signals.

Unfortunately, the general automatic-signal monitor becomes expensive when the number of tasks in the monitor is large, which is often true after saturation. However, restricted automatic-signal monitors, which also have simple proof rules, are competitive with explicit-signal monitors, especially when there are only a few condition variables, because they depend only on the number of conditional expressions and not the number of tasks in the monitor. Yet, the restricted automatic-signal monitor's conditional expressions cannot involve local variables or the parameters of a request, for example, a disk-scheduling algorithm where requests are serviced in an order based on the track number in the request. Hence, there are a class of important problems that cannot be handled by a restricted automatic-signal monitor.

8 Conclusion

The classification criterion, that is, the semantics of the signal mechanism, seems to define the fundamental characteristic of a monitor. It identifies a total of 10 useful kinds of monitors: 8 explicit-signal and 2 implicit-signal monitors based on categorizations of priority and blocking. (The restricted automatic-signal monitor is considered a variation of the automatic-signal monitor).

Howard [1976b, p. 51] stated that "none of the conventions is clearly superior to the others". Furthermore, we have shown that all the kinds of monitors except the immediate-return monitor can be made to do functionally equivalent things. Nevertheless, our analysis has uncovered several important differences among the monitors. In all cases, the no-priority property complicates the proof rules, makes performance worse, and makes programming more difficult. Using signals as hints to solve these problems has become a monitor idiom; however, this idiom seems forced to us and gives a false impression about how monitors work, in general. Therefore, we have rejected all no-priority monitors from further consideration.

Proof Rules		Performance		Subjective Ease	
		(Delay & Saturation)		in Programming	
simple	PAS	small	PRET	easy	PAS
	PB		**PNB**		**PNB**
	PNB, PRET		PB		PB
	PQB		PQB		PRET
complex		large	PAS	difficult	PQB

Table 9: Final Results

The remaining kinds of monitors are summarized in Table 9. The table shows that when designing a concurrency system using monitors for synchronization and communication there are several choices. On the basis of our analysis, we conclude that the priority nonblocking monitor has the best all-around semantics, which differs from Andrews's [1991, p. 315] choice of no priority nonblocking. First, its proof rules are about in the middle of the range in terms of complexity, but can be simplified by following a coding style of not affecting the monitor invariant between a signal and wait or exit. Second, there are no unnecessary context switches and concurrency is not inhibited so its performance is always excellent. In particular, its ability to handle higher loads before saturating is important for real-time systems. If it is necessary for a signaller to wait for the signalled task to execute (blocking semantics), it is very easy to simulate this by introducing an additional condition variable, and waiting and signalling it, appropriately. Third, because it gives priority to resuming tasks, it is straightforward to program with, allowing internal protocols to be created and scheduling schemes such as FIFO to be implemented without the extra code needed in a no-priority monitor. For these reasons, the priority nonblocking signal was adopted for monitors in μC++ [Buhr *et al.* 1992].

A Appendix – Readers and Writer Problem, Priority Monitor

A.1 Coding Style 1

This solution is suggested by the blocking signal. When the last reader of a group of readers is finished or a writer is finished with the resource, it checks the front of the condition queue and, if possible, signals another task that may restart execution. If the signalled task is a reader, it checks the front of the queue and signals at most one other reader task, which in turn may signal more if that is appropriate (i.e., a daisy-chain effect).

```
uMonitor RW( MONITOR_TYPE ) {
#   define READER 0
#   define WRITER 1

    int ReadCount = 0, WriteUsage = 0;
    uCondition ReaderAndWriter = U_CONDITION;

    uEntry void StartRead( void ) {
        if ( WriteUsage || uCondLength( &ReaderAndWriter ) != 0 ) {
            uWait ReaderAndWriter with READER;
        } /* if */
        ReadCount += 1;
        if ( uCondLength( &ReaderAndWriter ) != 0 &&
                uCondFront( &ReaderAndWriter) == READER ) {
            uSignal ReaderAndWriter;
        } /* if */
    } /* StartRead */

    uEntry void EndRead( void ) {
        ReadCount -= 1;
        if ( ReadCount == 0 && uCondLength( &ReaderAndWriter ) != 0 ) {
            uSignal ReaderAndWriter;
        } /* if */
    } /* EndRead */

    uEntry void StartWrite( void ) {
        if ( WriteUsage || ReadCount != 0 ) {
            uWait ReaderAndWriter with WRITER;
        } /* if */
        WriteUsage = 1;
    } /* StartWrite */

    uEntry void EndWrite( void ) {
        WriteUsage = 0;
        if ( uCondLength( &ReaderAndWriter ) != 0 ) {
            uSignal ReaderAndWriter;
        } /* if */
    } /* EndWrite */
} /* RW */
```

A.2 Coding Style 2

This solution is suggested by the nonblocking signal. The last task to use the resource starts as many tasks as it can at the front of the condition queue. Thus, when a task is finished with the resource, it will potentially signal multiple tasks, which do not signal other tasks.

```
uMonitor RW( MONITOR_TYPE ) {
#   define READER 0
#   define WRITER 1
```

```
    int ReadCount = 0, WriteUsage = 0;
    uCondition ReaderAndWriter = U_CONDITION;

    uEntry void StartRead( void ) {
        if ( WriteUsage || uCondLength( &ReaderAndWriter ) != 0 ) {
            uWait ReaderAndWriter with READER;
        } /* if */
        ReadCount += 1;
    } /* StartRead */

    uEntry void EndRead( void ) {
        ReadCount -= 1;
        if ( ReadCount == 0 && uCondLength( &ReaderAndWriter ) != 0 ) {
            uSignal ReaderAndWriter;
        } /* if */
    } /* EndRead */

    uEntry void StartWrite( void ) {
        if ( WriteUsage || ReadCount != 0 ) {
            uWait ReaderAndWriter with WRITER;
        } /* if */
        WriteUsage = 1;
    } /* StartWrite */

    uEntry void EndWrite( void ) {
        WriteUsage = 0;
        if ( uCondLength( &ReaderAndWriter ) != 0 ) {
            if ( uCondFront( &ReaderAndWriter ) == WRITER ) {
                uSignal ReaderAndWriter;
            } else {
                for ( ;; ) {
                    uSignal ReaderAndWriter;
                  if ( uCondLength( &ReaderAndWriter ) == 0 ) break;
                  if ( uCondFront( &ReaderAndWriter ) == WRITER ) break;
                } /* for */
            } /* if */
        } /* if */
    } /* EndWrite */
} /* RW */
```

B Appendix – Readers and Write Problem, No Priority Monitor

Because these solutions to the readers and writer problem require FIFO servicing, extra code must be added to ensure this.

B.1 Coding Style 1

This solution is suggested by the blocking signal. When the last reader of a group of readers is finished or a writer is finished with the resource, it checks the front of the condition queue and, if possible, signals another task that may restart execution. If the signalled task is a reader, it checks the front of the queue and signals at most one other reader task, which in turn may signal more if that is appropriate (i.e., a daisy-chain effect).

```
uMonitor RW( MONITOR_TYPE ) {
#   define READER 0
#   define WRITER 1

    int ReadCount = 0, WriteUsage = 0, Pending = 0;
    uCondition ReaderAndWriter = U_CONDITION;

    uEntry void StartRead( void ) {
        if ( WriteUsage || uCondLength( &ReaderAndWriter ) != 0 || Pending != 0 ) {
            uWait ReaderAndWriter with READER;
            Pending -= 1;
        } /* if */
        ReadCount += 1;
        if ( uCondLength( &ReaderAndWriter ) != 0 &&
                  uCondFront( &ReaderAndWriter)  == READER ) {
            Pending += 1;
            uSignal ReaderAndWriter;
        } /* if */
    } /* StartRead */

    uEntry void EndRead( void ) {
        ReadCount -= 1;
        if ( ReadCount == 0 && uCondLength( &ReaderAndWriter ) != 0 ) {
            Pending += 1;
            uSignal ReaderAndWriter;
        } /* if */
    } /* EndRead */

    uEntry void StartWrite( void ) {
        if ( WriteUsage || ReadCount != 0 || Pending != 0 ) {
            uWait ReaderAndWriter with WRITER;
            Pending -= 1;
        } /* if */
        WriteUsage = 1;
    } /* StartWrite */

    uEntry void EndWrite( void ) {
        WriteUsage = 0;
        if ( uCondLength( &ReaderAndWriter ) != 0 ) {
            Pending += 1;
            uSignal ReaderAndWriter;
        } /* if */
    } /* EndWrite */
} /* RW */
```

B.2 Coding Style 2

This solution is suggested by the nonblocking signal. The last task to use the resource starts as many tasks as it can at the front of the condition queue. Thus, when a task is finished with the resource, it will potentially signal multiple tasks, which do not signal other tasks.

```
uMonitor RW( MONITOR_TYPE ) {
#   define READER 0
#   define WRITER 1

    int ReadCount = 0, WriteUsage = 0, Pending = 0;
    uCondition ReaderAndWriter = U_CONDITION;

    uEntry void StartRead( void ) {
        if ( WriteUsage || uCondLength( &ReaderAndWriter ) != 0 || Pending != 0 ) {
            uWait ReaderAndWriter with READER;
            Pending -= 1;
        } /* if */
        ReadCount += 1;
    } /* StartRead */

    uEntry void EndRead( void ) {
        ReadCount -= 1;
        if ( ReadCount == 0 && uCondLength( &ReaderAndWriter ) != 0 ) {
            Pending += 1;
            uSignal ReaderAndWriter;
        } /* if */
    } /* EndRead */

    uEntry void StartWrite( void ) {
        if ( WriteUsage || ReadCount != 0 || Pending != 0 ) {
            uWait ReaderAndWriter with WRITER;
            Pending -= 1;
        } /* if */
        WriteUsage = 1;
    } /* StartWrite */

    uEntry void EndWrite( void ) {
        WriteUsage = 0;
        if ( uCondLength( &ReaderAndWriter ) != 0 ) {
            if ( uCondFront( &ReaderAndWriter ) == WRITER ) {
                Pending += 1;
                uSignal ReaderAndWriter;
            } else {
                for ( ;; ) {
                    Pending += 1;
                    uSignal ReaderAndWriter;
                  if ( uCondLength( &ReaderAndWriter ) == 0 ) break;
                  if ( uCondFront( &ReaderAndWriter ) == WRITER ) break;
                } /* for */
            } /* if */
        } /* if */
    } /* EndWrite */
} /* RW */
```

C Acknowledgments

We thank Cathy Kelley and John Fahey for helping us build a statistically valid experiment and analyze the results of the experiment. We also thank Bob Zarnke and Jo Ebergen for reading and commenting on this work.

References

ADAMS, J. M., AND BLACK, A. P. 1982. On proof rules for monitors. *Operating Systems Review 16*, 2 (Apr.), 18–27.

ADAMS, J. M., AND BLACK, A. P. 1983. Letter to the editor. *Operating Systems Review 17*, 1 (Jan.), 6–8.

ANDREWS, G. R., AND SCHNEIDER, F. B. 1983. Concepts and notations for concurrent programming. *ACM Comput. Surv. 15*, 1 (Mar.), 3–43.

ANDREWS, G. R. 1991. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, California.

BRINCH HANSEN, P. 1973. *Operating System Principles*. Prentice-Hall.

BRINCH HANSEN, P. 1975. The programming language concurrent pascal. *IEEE Trans. Softw. Eng. 2* (June), 199–206.

BUHR, P. A., AND STROOBOSSCHER, R. A. 1990. The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. *Software—Practice and Experience 20*, 9 (Sept.), 929–963.

BUHR, P. A., DITCHFIELD, G., STROOBOSSCHER, R. A., YOUNGER, B. M., AND ZARNKE, C. R. 1992. μC++: Concurrency in the object-oriented language C++. *Software—Practice and Experience 22*, 2 (Feb.), 137–172.

CAMPBELL, R. H., AND HABERMANN, A. N. 1974. *The Specification of Process Synchronization by Path Expressions*, vol. 16 of *Lecture Notes in Computer Science*. Springer-Verlag.

CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. 1988. Modula-3 report. Tech. Rep. 31, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, Aug.

CHERITON, D. R. 1982. *The Thoth System: Multi-Process Structuring and Portability*. American Elsevier.

CHERITON, D. R. 1988. The v distributed system. *Commun. ACM 31*, 3 (Mar.), 314–333.

COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. L. 1971. Concurrent control with readers and writers. *Commun. ACM 14*, 10 (Oct.), 667–668.

DAHL, O.-J., MYHRHAUG, B., AND NYGAARD, K. 1970. *Simula67 Common Base Language*. Norwegian Computing Center, Oslo Norway.

DIJKSTRA, E. W. 1968. The structure of the “THE”–multiprogramming system. *Commun. ACM 11*, 5 (May), 341–346.

FORTIER, M. 1989. Study of monitors. Master’s thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1989.

GENTLEMAN, W. M. 1985. Using the harmony operating system. Tech. Rep. 24685, National Research Council of Canada, Ottawa, Canada, May.

GOSLING, J., ROSENTHAL, D. S. H., AND ARDEN, R. J. 1989. *The NeWS Book*. Springer-Verlag.

GRIES, D. 1981. *The Science of Computer Programming*. Springer-Verlag, 175 Fifth Ave., New York, New York 10010.

HANSEN, P. B. 1981. The design of edison. *Software Practice and Experience 11*, 4 (Apr.), 363–396.

HANSEN, P. B. 1981. Edison—a multiprocessor language. *Software Practice and Experience 11*, 4 (Apr.), 325–361.

HANSEN, P. B. 1981. Edison programs. *Software Practice and Experience 11*, 4 (Apr.), 397–414.

HOARE, C. A. R. 1974. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct.), 549–557.

HOLT, R. C., AND CORDY, J. R. 1988. The turing programming language. *Commun. ACM 31*, 12 (Dec.), 1410–1423.

HOLT, R. C. 1992. *Turing Reference Manual*, third ed. Holt Software Associates Inc.

HOWARD, J. H. 1976. Proving monitors. *Commun. ACM 19*, 5 (May), 273–279.

HOWARD, J. H. 1976. Signalling in monitors. In *Proceedings Second International Conference Software Engineering* (San Francisco, U.S.A, Oct.), pp. 47–52.

HOWARD, J. H. 1982. Reply to “on proof rules for monitors”. *Operating Systems Review 16*, 5 (Oct.), 8–9.

KERNIGHAN, B. W., AND RITCHIE, D. M. 1988. *The C Programming Language*, second ed. Prentice Hall Software Series. Prentice Hall.

KESSELS, J. L. W. 1977. An alternative to event queues for synchronization in monitors. *Commun. ACM 20*, 7 (July), 500–503.

LAMPSON, B. W., AND REDELL, D. D. 1980. Experience with processes and monitors in mesa. *Commun. ACM 23*, 2 (Feb.), 105–117.

MITCHELL, J. G., MAYBURY, W., AND SWEET, R. 1979. Mesa language manual. Tech. Rep. CSL–79–3, Xerox Palo Alto Research Center, Apr.

NELSON, G., Ed. 1991. *Systems Programming with Modula-3*. Prentice-Hall, Inc.

RAJ, R. K., TEMPERO, E., LEVY, H. M., BLACK, A. P., HUTCHINSON, N. C., AND JUL, E. 1991. Emerald: A general-purpose programming language. *Software—Practice and Experience 21*, 1 (Jan.), 91–118.

WIRTH, N. 1985. *Programming in Modula-2*, third, corrected ed. Texts and Monographs in Computer Science. Springer-Verlag.