Experience with Processes and Monitors in Mesa[1]

Butler W. Lampson
Xerox Palo Alto Research Center

David D. Redell
Xerox Business Systems

Abstract

The use of monitors for describing concurrency has been much discussed in the literature. When monitors are used in real systems of any size, however, a number of problems arise which have not been adequately dealt with: the semantics of nested monitor calls; the various ways of defining the meaning of WAIT; priority scheduling; handling of timeouts, aborts and other exceptional conditions; interactions with process creation and destruction; monitoring large numbers of small objects. These problems are addressed by the facilities described here for concurrent programming in Mesa. Experience with several substantial applications gives us some confidence in the validity of our solutions.

Key Words and Phrases: concurrency, condition variable, deadlock, module, monitor, operating system, process, synchronization, task

CR Categories: 4.32, 4.35, 5.24

1. Introduction

In early 1977 we began to design the concurrent programming facilities of Pilot, a new operating system for a personal computer [18]. Pilot is a fairly large program itself (24,000 lines of Mesa code). In addition, it must support a variety of quite large application programs, ranging from database management to inter-network message transmission, which are heavy users of concurrency; our experience with some of these applications is discussed later in the paper. We intended the new facilities to be used at least for the following purposes:

> *Local concurrent programming. An* individual application can be implemented as a tightly coupled group of synchronized processes to express the concurrency inherent in the application.

---

[1] This paper appeared in *Communications of the ACM* **23**, 2 (Feb. 1980), pp 105-117. An earlier version was presented at the 7th ACM Symposium on Operating Systems Principles, Pacific Grove, CA, Dec. 1979. This version was created from the published version by scanning and OCR; it may have errors.

Global resource sharing. Independent applications can run together on the same machine, cooperatively sharing the resources; in particular, their processes can share the processor.

Replacing interrupts. A request for software attention to a device can be handled directly by waking up an appropriate process, without going through a separate interrupt mechanism (for example, a forced branch).

Pilot is closely coupled to the Mesa language [17], which is used to write both Pilot itself and the applications programs it supports. Hence it was natural to design these facilities as part of Mesa; this makes them easier to use, and also allows the compiler to detect many kinds of errors in their use. The idea of integrating such facilities into a language is certainly not new; it goes back at least as far as PL/1 [1]. Furthermore, the invention of monitors by Dijkstra, Hoare, and Brinch Hansen [3, 5, 8] provided a very attractive framework for reliable concurrent programming. There followed a number of papers on the integration of concurrency into programming languages, and at least one implementation [4].

We therefore thought that our task would be an easy one: read the literature, compare the alternatives offered there, and pick the one most suitable for our needs. This expectation proved to be naive. Because of the large size and wide variety of our applications, we had to address a number of issues which were not clearly resolved in the published work on monitors. The most notable among these are listed below, with the sections in which they are discussed.

(a) *Program structure*. Mesa has facilities for organizing programs into modules which communicate through well-defined interfaces. Processes must fit into this scheme (see Section 3.1).

(b) *Creating processes*. A set of processes fixed at compile-time is unacceptable in such a general-purpose system (See Section 2). Existing proposals for varying the amount of concurrency were limited to concurrent elaboration of the statements in a block, in the style of Algol 68 (except for the rather complex mechanism in PL/1).

(c) *Creating monitors*. A fixed number of monitors is also unacceptable, since the number of synchronizers should be a function of the amount of data, but many of the details of existing proposals depended on a fixed association of a monitor with a block of the program text (see Section 3.2).

(d) *WAIT in a nested monitor call*. This issue had been (and has continued to be) the source of a considerable amount of confusion, which we had to resolve in an acceptable manner before we could proceed (see Section 3.1).

(e) *Exceptions*. A realistic system must have timeouts, and it must have a way to abort a process (see Section 4.1). Mesa has an UNWIND mechanism for abandoning part of a sequential computation in an orderly way, and this must interact properly with monitors (see Section 3.3).

(f) *Scheduling*. The precise semantics of waiting on a condition variable had been discussed [10] but not agreed upon, and the reasons for making any particular choice had not been articulated (see Section 4). No attention had been paid to the interaction between monitors and priority scheduling of processes (see Section 4.3).

(g) *Input-Output*. The details of fitting I/O devices into the framework of monitors and condition variables had not been fully worked out (see Section 4.2).

Some of these points have also been made by Keedy [12], who discusses the usefulness of monitors in a modern general-purpose mainframe operating system. The Modula language [21] addresses (b) and (g), but in a more limited context than ours.

Before settling on the monitor scheme described below, we considered other possibilities. We felt that our first task was to choose either shared memory (that is, monitors) or message passing as our basic interprocess communication paradigm.

Message passing has been used (without language support) in a number of operating systems; for a recent proposal to embed messages in a language, see [9]. An analysis of the differences between such schemes and those based on monitors was made by Lauer and Needham [14]. They conclude that, given certain mild restrictions on programming style, the two schemes are duals under the transformation

message ↔ process
process ↔ monitor
send/reply ↔ call/return

Since our work is based on a language whose main tool of program structuring is the procedure, it was considerably easier to use a monitor scheme than to devise a message-passing scheme properly integrated with the type system and control structures of the language.

Within the shared memory paradigm, we considered the possibility of adopting a simpler primitive synchronization facility than monitors. Assuming the absence of multiple processors, the simplest form of mutual exclusion appears to be a non-preemptive scheduler; if processes only yield the processor voluntarily, then mutual exclusion is insured between yield points. In its simplest form, this approach tends to produce very delicate programs, since the insertion of a yield in a random place can introduce a subtle bug in a previously correct program. This danger can be alleviated by the addition of a modest amount of "syntactic sugar" to delineate critical sections within which the processor must not be yielded (for example, pseudo monitors). This sugared form of non-preemptive scheduling can provide extremely efficient solutions to simple problems, but was nonetheless rejected for four reasons:

(1) While we were willing to accept an implementation that would not work on multiple processors, we did not want to embed this restriction in our basic semantics.

(2) A separate preemptive mechanism is needed anyway, since the processor must respond to time-critical events (for example, I/O interrupts) for which voluntary process switching is clearly too sluggish. With preemptive process scheduling, interrupts can be treated as ordinary process wakeups, which reduces the total amount of machinery needed and eliminates the awkward situations that tend to occur at the boundary between two scheduling regimes.

(3) The use of non-preemption as mutual exclusion restricts programming generality within critical sections; in particular, a procedure that happens to yield the processor cannot be called. In large systems where modularity is essential, such restrictions are intolerable.

(4) The Mesa concurrency facilities function in a virtual memory environment. The use of non-preemption as mutual exclusion forbids multiprogramming across page faults, since that would effectively insert preemptions at arbitrary points in the program.

For mutual exclusion with a preemptive scheduler, it is necessary to introduce explicit locks, and machinery that makes requesting processes wait when a lock is unavailable. We considered casting our locks as semaphores, but decided that, compared with monitors, they exert too little structuring discipline on concurrent programs. Semaphores do solve several different problems with a single mechanism (for example, mutual exclusion, producer/consumer) but we found similar economies in our implementation of monitors and condition variables (see Section 5.1).

We have not associated any protection mechanism with processes in Mesa, except what is implicit in the type system of the language. Since the system supports only one user, we feel that the considerable protection offered by the strong typing of the language is sufficient. This fact contributes substantially to the low cost of process operations.

2. Processes

Mesa casts the creation of a new process as a special procedure activation that executes concurrently with its caller. Mesa allows *any* procedure (except an internal procedure of a monitor; see Section 3.1) to be invoked in this way, at the caller's discretion. It is possible to later retrieve the results returned by the procedure. For example, a keyboard input routine might be invoked as a normal procedure by writing:

buffer ← *ReadLine*[*terminal*]

but since *ReadLine* is likely to wait for input, its caller might wish instead to compute concurrently:

p ← FORK *ReadLine*[*terminal*];
... <concurrent computation> ...
buffer ← JOIN *p*;

Here the types are

ReadLine: PROCEDURE [*Device*] RETURNS [*Line*];
p: PROCESS RETURNS [*Line*];

The rendezvous between the return from *ReadLine* that terminates the new process and the join in the old process is provided automatically. *ReadLine* is the *root* procedure of the new process.

This scheme has a number of important properties.

(h) It treats a process as a first class value in the language, which can be assigned to a variable or an array element, passed as a parameter, and in general treated exactly like any other value. A process value is like a pointer value or a procedure value that refers to a nested procedure, in that it can become a dangling reference if the process to which it refers goes away.

(i) The method for passing parameters to a new process and retrieving its results is exactly the same as the corresponding method for procedures, and is subject to the same strict type

checking. Just as PROCEDURE is a generator for a family of types (depending on the argument and result types), so PROCESS is a similar generator, slightly simpler since it depends only on result types.

(j) No special declaration is needed for a procedure that is invoked as a process. Because of the implementation of procedure calls and other global control transfers in Mesa [13], there is no extra execution cost for this generality.

(k) The cost of creating and destroying a process is moderate, and the cost in storage is only twice the minimum cost of a procedure instance. It is therefore feasible to program with a large number of processes, and to vary the number quite rapidly. As Lauer and Needham [14] point out, there are many synchronization problems that have straightforward solutions using monitors only when obtaining a new process is cheap.

Many patterns of process creation are possible. A common one is to create a *detached* process that never returns a result to its creator, but instead functions quite independently. When the root procedure *p* of a detached process returns, the process is destroyed without any fuss. The fact that no one intends to wait for a result from *p* can be expressed by executing:

Detach[*p*]

From the point of view of the caller, this is similar to freeing a dynamic variable—it is generally an error to make any further use of the current value of *p*, since the process, running asynchronously, may complete its work and be destroyed at any time. Of course the design of the program may be such that this cannot happen, and in this case the value of *p* can still be useful as a parameter to the *Abort* operation (see Section 4.1).

This remark illustrates a general point: Processes offer some new opportunities to create dangling references. A process variable itself is a kind of pointer, and must not be used after the process is destroyed. Furthermore, parameters passed by reference to a process are pointers, and if they happen to be local variables of a procedure, that procedure must not return until the process is destroyed. Like most implementation languages, Mesa does not provide any protection against dangling references, whether connected with processes or not.

The ordinary Mesa facility for exception handling uses the ordering established by procedure calls to control the processing of exceptions. Any block may have an attached exception handler. The block containing the statement that causes the exception is given the first chance to handle it, then its enclosing block, and so forth until a procedure body is reached. Then the caller of the procedure is given a chance in the same way. Since the root procedure of a process has no caller, it must be prepared to handle any exceptions that can be generated in the process, including exceptions generated by the procedure itself. If it fails to do so, the resulting error sends control to the debugger, where the identity of the procedure and the exception can easily be determined by a programmer. This is not much comfort, however, when a system is in operational use. The practical consequence is that while any procedure suitable for forking can also be called sequentially, the converse is not generally true.

3. Monitors

When several processes interact by sharing data, care must be taken to properly synchronize access to the data. The idea behind monitors is that a proper vehicle for this interaction is one that unifies

- the synchronization,
- the shared data,
- the body of code which performs the accesses.

The data is *protected* by a *monitor*, and can only be accessed within the body of a *monitor procedure*. There are two kinds of monitor procedures: *entry procedures*, which can be called from outside the monitor, and *internal procedures*, which can only be called from monitor procedures. Processes can only perform operations on the data by calling entry procedures. The monitor ensures that at most one process is executing a monitor procedure at a time; this process is said to be *in* the monitor. If a process is in the monitor, any other process that calls an entry procedure will be delayed. The monitor procedures are written textually next to each other, and next to the declaration of the protected data, so that a reader can conveniently survey all the references to the data.

As long as any order of calling the entry procedures produces meaningful results, no additional synchronization is needed among the processes sharing the monitor. If a random order is not acceptable, other provisions must be made in the program outside the monitor. For example, an unbounded buffer with *Put* and *Get* procedures imposes no constraints (of course a *Get* may have to wait, but this is taken care of within the monitor, as described in the next section). On the other hand, a tape unit with *Reserve, Read, Write,* and *Release* operations requires that each process execute a *Reserve* first and a *Release* last. A second process executing a *Reserve* will be delayed by the monitor, but another process doing a *Read* without a prior *Reserve* will produce chaos. Thus monitors do not solve all the problems of concurrent programming; they are intended, in part, as primitive building blocks for more complex scheduling policies. A discussion of such policies and how to implement them using monitors is beyond the scope of this paper.

3.1 Monitor modules

In Mesa the simplest monitor is an instance of a *module*, which is the basic unit of global program structuring. A Mesa module consists of a collection of procedures and their global data, and in sequential programming is used to implement a data abstraction. Such a module has PUBLIC procedures that constitute the external interface to the abstraction, and PRIVATE procedures that are internal to the implementation and cannot be called from outside the module; its data is normally entirely private. A MONITOR module differs only slightly. It has three kinds of procedures: *entry, internal* (private), and *external* (non-monitor procedures). The first two are the monitor procedures, and execute with the monitor lock held. For example, consider a simple storage allocator with two entry procedures, *Allocate* and *Free*, and an external procedure *Expand* that increases the size of a block.

```
StorageAllocator: MONITOR = BEGIN
    availableStorage: INTEGER:
    moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
    UNTIL availableStorage ≥ size
        DO WAIT moreAvailable ENDLOOP;
    p ← <remove chunk of size words & update availableStorage>
    END;

Free: ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
    <put back chunk of size words & update availableStorage>;
    NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
    pNew ← Allocate[size];
    <copy contents from old block to new block>;
    Free[pOld] END;

END.
```

A Mesa module is normally used to package a collection of related procedures and protect their private data from external access. In order to avoid introducing a new lexical structuring mechanism, we chose LO make the scope of a monitor identical to a module. Sometimes, however, procedures that belong in an abstraction do not need access to any shared data, and hence need not be entry procedures of the monitor; these must be distinguished somehow.

For example, two asynchronous processes clearly must not execute in the *Allocate* or *Free* procedures at the same time; hence, these must be entry procedures. On the other hand, it is unnecessary to hold the monitor lock during the copy in *Expand,* even though this procedure logically belongs in the storage allocator module; it is thus written as an external procedure. A more complex monitor might also have internal procedures, which are used to structure its computations, but which are inaccessible from outside the monitor. These do not acquire and release the lock on call and return, since they can only be called when the lock is already held.

If no suitable block is available, *Allocate* makes its caller wait on the *condition* variable *moreAvailable. Free* does a NOTIFY to this variable whenever a new block becomes available; this causes some process waiting on the variable to resume execution (see Section 4 for details). The WAIT releases the monitor lock, which is reacquired when the waiting process reenters the monitor. If a WAIT is done in an internal procedure, it still releases the lock. If, however, the monitor calls some other procedure which is outside the monitor module, the lock is not released, even if the other procedure is in (or calls) another monitor and ends up doing a WAIT. The same rule is adopted in Concurrent Pascal [4].

To understand the reasons for this, consider the form of a correctness argument for a program using a monitor. The basic idea is that the monitor maintains an *invariant* that is always true of its data, except when some process is executing in the monitor. Whenever control leaves the monitor, this invariant must be established. In return, whenever control enters the monitor the invariant can be assumed. Thus an entry procedure must establish the invariant before returning, and a monitor procedure must establish it before doing a WAIT. The invariant can be assumed at

the start of an entry procedure, and after each WAIT. Under these conditions, the monitor lock ensures that no one can enter the monitor when the invariant is false. Now, if the lock were to be released on a WAIT done in another monitor which happens to be called from this one, the invariant would have to be established before making the call which leads to the WAIT. Since in general there is no way to know whether a call outside the monitor will lead to a WAIT, the invariant would have to be established before every such call. The result would be to make calling such procedures hopelessly cumbersome.

An alternative solution is to allow an *outside block* to be written inside a monitor, with the following meaning: on entry to the block the lock is released (and hence the invariant must be established); within the block the protected data is inaccessible; on leaving the block the lock is reacquired. This scheme allows the state represented by the execution environment of the monitor to be maintained during the outside call, and imposes a minimal burden on the programmer: to establish the invariant before making the call. This mechanism would be easy to add to Mesa; we have left it out because we have not seen convincing examples in which it significantly simplifies the program.

If an entry procedure generates an exception in the usual way, the result will be a call on the exception handler from within the monitor, so that the lock will not be released. In particular, this means that the exception handler must carefully avoid invoking that same monitor, or a deadlock will result. To avoid this restriction, the entry procedure can restore the invariant and then execute

RETURN WITH ERROR[(arguments)]

which returns from the entry procedure, thus releasing the lock, and then generates the exception.

3.2 Monitors and deadlock

There are three patterns of pairwise deadlock that can occur using monitors. In practice, of course, deadlocks often involve more than two processes, in which case the actual patterns observed tend to be more complicated; conversely, it is also possible for a single process to deadlock with itself (for example, if an entry procedure is recursive).

The simplest form of deadlock takes place inside a single monitor when two processes do a WAIT, each expecting to be awakened by the other. This represents a localized bug in the monitor code and is usually easy to locate and correct.

A more subtle form of deadlock can occur if there is a cyclic calling pattern between two monitors. Thus if monitor *M* calls an entry procedure in *N*, and *N* calls one in M, each will wait for the other to release the monitor lock. This kind of deadlock is made neither more nor less serious by the monitor mechanism. It arises whenever such cyclic dependencies are allowed to occur in a program, and can be avoided in a number of ways. The simplest is to impose a partial ordering on resources such that all the resources simultaneously possessed by any process are totally ordered, and insist that if resource *r* precedes 5 in the ordering, then *r* cannot be acquired later than *5*. When the resources are monitors, this reduces to the simple rule that mutually recursive monitors must be avoided. Concurrent Pascal [4] makes this check at compile time; Mesa cannot do so because it has procedure variables.

A more serious problem arises if *M* calls *N*, and *N* then waits for a condition which can only occur when another process enters *N* through *M* and makes the condition true. In this situation, *N* will be unlocked, since the WAIT occurred there, but *M* will remain locked during the WAIT in *N*. This kind of two level data abstraction must be handled with some care. A straightforward solution using standard monitors is to break *M* into two parts: a monitor *M'* and an ordinary module *0* which implements the abstraction defined by *M*, and calls *M'* for access to the shared data. The call on *N* must be done from *0* rather than from within *M'*.

Monitors, like any other interprocess communication mechanism, are a *tool* for implementing synchronization constraints chosen by the programmer. It is unreasonable to blame the tool when poorly chosen constraints lead to deadlock. What is crucial, however, is that the tool make the program structure as understandable as possible, while not restricting the programmer too much in his choice of constraints (for example, by forcing a monitor lock to be held much longer than necessary). To some extent, these two goals tend to conflict; the Mesa concurrency facilities attempt to strike a reasonable balance and provide an environment in which the conscientious programmer can avoid deadlock reasonably easily. Our experience in this area is reported in Section 6.

3.3 Monitored objects

Often we wish to have a collection of shared data objects, each one representing an instance of some abstract object such as a file, a storage volume, a virtual circuit, or a database view, and we wish to add objects to the collection and delete them dynamically. In a sequential program this is done with standard techniques for allocating and freeing storage. In a concurrent program, however, provision must also be made for serializing access to each object. The straightforward way is to use a single monitor for accessing all instances of the object, and we recommend this approach whenever possible. If the objects function independently of each other for the most part, however, the single monitor drastically reduces the maximum concurrency that can be obtained. In this case, what we want is to give each object its own monitor; all these monitors will share the same code, since all the instances of the abstract object share the same code, but each object will have its own lock.

One way to achieve this result is to make multiple instances of the monitor module. Mesa makes this quite easy, and it is the next recommended approach. However, the data associated with a module instance includes information that the Mesa system uses to support program linking and code swapping, and there is some cost in duplicating this information. Furthermore, module instances are allocated by the system; hence the program cannot exercise the fme control over allocation strategies which is possible for ordinary Mesa data objects. We have therefore introduced a new type constructor called a *monitored record*, which is exactly like an ordinary record, except that it includes a monitor lock and is intended to be used as the protected data of a monitor.

In writing the code for such a monitor, the programmer must specify how to access the monitored record, which might be embedded in some larger data structure passed as a parameter to the entry procedures. This is done with a LOCKS clause which is written at the beginning of the module:

MONITOR LOCKS *file*↑ USING *file*: POINTER TO *FileData*;

if the *FileData* is the protected data. An arbitrary expression can appear in the LOCKS clause; for instance, LOCKS *file.buffers*[*currentPage*] might be appropriate if the protected data is one of the buffers in an array which is part of the *file*. Every entry procedure of this monitor, and every internal procedure that does a WAIT, must have access to a *file*, so that it can acquire and release the lock upon entry or around a WAIT. This can be accomplished in two ways: the *file* may be a global variable of the module, or it may be a parameter to *every* such procedure. In the latter case, we have effectively created a separate monitor for each object, without limiting the program's freedom to arrange access paths and storage allocation as it likes.

Unfortunately, the type system of Mesa is not strong enough to make this construction completely safe. If the value of *file* is changed within an entry procedure, for example, chaos will result, since the return from this procedure will release not the lock which was acquired during the call, but some other lock instead. In this example we can insist that *file* be read-only, but with another level of indirection aliasing can occur and such a restriction cannot be enforced. In practice this lack of safety has not been a problem.

3.4 Abandoning a computation

Suppose that a procedure P_1 has called another procedure P_2, which in turn has called P_3 and so forth until the current procedure is P_n. If P_n generates an exception which is eventually handled by P_1 (because P_2 ... P_n do not provide handlers), Mesa allows the exception handler in P_1 to abandon the portion of the computation being done in P_2 ... P_n and continue execution in P_1. When this happens, a distinguished exception called UNWIND is first generated, and each of P_2 ... P_n is given a chance to handle it and do any necessary cleanup before its activation is destroyed.

This feature of Mesa is not part of the concurrency facilities, but it does interact with those facilities in the following way. If one of the procedures being abandoned, say P_i, is an entry procedure, then the invariant must be restored and the monitor lock released before P_i is destroyed. Thus if the logic of the program allows an UNWIND, the programmer must supply a suitable handler in P_i to restore the invariant; Mesa will automatically supply the code to release the lock. If the programmer fails to supply an UNWIND handler for an entry procedure, the lock is *not* automatically released, but remains set; the cause of the resulting deadlock is not hard to find.

4. Condition variables

In this section we discuss the precise semantics of WAIT and other details associated with condition variables. Hoare's definition of monitors [8] requires that a process waiting on a condition variable must run immediately when another process *signals* that variable, and that the signaling process in turn runs as soon as the waiter leaves the monitor. This definition allows the waiter to assume the truth of some predicate stronger than the monitor invariant (which the signaler must of course establish), but it requires several additional process switches whenever a process continues after a WAIT. It also requires that the signaling mechanism be perfectly reliable.

Mesa takes a different view: When one process establishes a condition for which some other process may be waiting, it *notifies* the corresponding condition variable. A NOTIFY is regarded as a *hint* to a waiting process; it causes execution of some process waiting on the condition to resume at some convenient future time. When the waiting process resumes, it will reacquire the

monitor lock. There is no guarantee that some other process will not enter the monitor before the waiting process. Hence nothing more than the monitor invariant may be assumed after a WAIT, and the waiter must reevaluate the situation each time it resumes. The proper pattern of code for waiting is therefore:

WHILE NOT <OK to proceed> DO WAIT *c* ENDLOOP.

This arrangement results in an extra evaluation of the <OK to proceed> predicate after a wait, compared to Hoare's monitors, in which the code is:

IF NOT <OK to proceed> THEN WAIT *c*.

In return, however, there are no extra process switches, and indeed no constraints at all on when the waiting process must run after a NOTIFY. In fact, it is perfectly all right to run the waiting process even if there is no NOTIFY, although this is presumably pointless if a NOTIFY is done whenever an interesting change is made to the protected data.

It is possible that such a laissez-faire attitude to scheduling monitor accesses will lead to unfairness and even starvation. We do not think this is a legitimate cause for concern, since in a properly designed system there should typically be no processes waiting for a monitor lock. As Hoare, Brinch Hansen, Keedy, and others have pointed out, the low level scheduling mechanism provided by monitor locks should not be used to implement high level scheduling decisions within a system (for example, about which process should get a printer next). High level scheduling should be done by taking account of the specific characteristics of the resource being scheduled (for example, whether the right kind of paper is in the printer). Such a scheduler will delay its client processes on condition variables after recording information about their requirements, make its decisions based on this information, and notify the proper conditions. In such a design the data protected by a monitor is never a bottleneck.

The verification rules for Mesa monitors are thus extremely simple: The monitor invariant must be established just before a return from an entry procedure or a WAIT, and it may be assumed at the start of an entry procedure and just after a WAIT. Since awakened waiters do not run immediately, the predicate established before a NOTIFY cannot be assumed after the corresponding WAIT, but since the waiter tests explicitly for <OK to proceed>, verification is actually made simpler and more localized.

Another consequence of Mesa's treatment of NOTIFY as a hint is that many applications do not trouble to determine whether the exact condition needed by a waiter has been established. Instead, they choose a very cheap predicate which implies the exact condition (for example, some change has occurred), and NOTIFY a *covering* condition variable. Any waiting process is then responsible for determining whether the exact condition holds; if not, it simply waits again. For example, a process may need to wait until a particular object in a set changes state. A single condition covers the entire set, and a process changing any of the objects broadcasts to this condition (see Section 4.1). The information about exactly which objects are currently of interest is implicit in the states of the waiting processes, rather than having to be represented explicitly in a shared data structure. This is an attractive way to decouple the detailed design of two processes: it is feasible because the cost of waking up a process is small.

4.1 Alternatives to NOTIFY

With this rule it is easy to add three additional ways to resume a waiting process:

Timeout. Associated with a condition variable is a timeout interval *t.* A process which has been waiting for time *t* will resume regardless of whether the condition has been notified. Presumably in most cases it will check the time and take some recovery action before waiting again. The original design for timeouts raised an exception if the timeout occurred; it was changed because many users simply wanted to retry on a timeout, and objected to the cost and coding complexity of handling the exception. This decision could certainly go either way.

Abort. A process may be aborted at any time by executing *Abort*[*p*]. The effect is that the next time the process waits, or if it is waiting now, it will resume immediately and the ***A****borted* exception will occur. This mechanism allows one process to gently prod another, generally to suggest that it should clean up and terminate. The aborted process is, however, free to do arbitrary computations, or indeed to ignore the abort entirely.

Broadcast. Instead of doing a NOTIFY to a condition, a process may do a BROADCAST, which causes ***all*** the processes waiting on the condition to resume, instead of simply one of them. Since a NOTIFY is just a hint, it is always correct to use BROADCAST. It is better to use NOTIFY if there will typically be several processes waiting on the condition, and it is known that any waiting process can respond properly. On the other hand, there are times when a BROADCAST is correct and a NOTIFY is not; the alert reader may have noticed a problem with the example program in Section 3.1, which can be solved by replacing the NOTIFY with a BROADCAST.

None of these mechanisms affects the proof rule for monitors at all. Each provides a way to attract the attention of a waiting process at an appropriate time.

Note that there is no way to stop a runaway process. This reflects the fact that Mesa processes are cooperative. Many aspects of the design would not be appropriate in a competitive environment such as a general-purpose timesharing system.

4.2 Naked NOTIFY

Communication with input/output devices is handled by monitors and condition variables much like communication among processes. There is typically a shared data structure, whose details are determined by the hardware, for passing commands to the device and returning status information. Since it is not possible for the device to wait on a monitor lock, the update operations on this structure must be designed so that the single word atomic read and write operations provided by the memory are sufficient to make them atomic. When the device needs attention, it can NOTIFY a condition variable to wake up a waiting process (that is, the interrupt handler); since the device does not actually acquire the monitor lock, its NOTIFY is called a ***naked*** NOTIFY. The device finds the address of the condition variable in a ftxed memory location.

There is one complication associated with a naked NOTIFY: Since the notification is not protected by a monitor lock, there can be a race. It is possible for a process to be in the monitor, find the <OK to proceed> predicate to be FALSE (that is, the device does not need attention), and be about to do a WAIT, when the device updates the shared data and does its NOTIFY. The WAIT will then be done and the NOTIFY from the device will be lost. With ordinary processes, this cannot happen,

since the monitor lock ensures that one process cannot be testing the predicate and preparing to WAIT, while another is changing the value of <OK to proceed> and doing the NOTIFY. The problem is avoided by providing the familiar wakeup-waiting switch [19] in a condition variable, thus turning it into a binary semaphore [8]. This switch is needed only for condition variables that are notified by devices.

We briefly considered a design in which devices would wait on and acquire the monitor lock, exactly like ordinary Mesa processes; this design is attractive because it avoids both the anomalies just discussed. However, there is a serious problem with any kind of mutual exclusion between two processes which run on processors of substantially different speeds: The faster process may have to wait for the slower one. The worst-case response time of the faster process therefore cannot be less than the time the slower one needs to finish its critical section. Although one can get higher throughput from the faster processor than from the slower one, one cannot get better worst-case real time performance. We consider this a fundamental deficiency.

It therefore seemed best to avoid any mutual exclusion (except for that provided by the atomic memory read and write operations) between Mesa code and device hardware and microcode. Their relationship is easily cast into a producer-consumer form, and this can be implemented, using linked lists or arrays, with only the memory's mutual exclusion. Only a small amount of Mesa code must handle device data structures without the protection of a monitor. Clearly a change of models must occur at some point between a disk head and an application program; we see no good reason why it should not happen within Mesa code, although it should certainly be tightly encapsulated.

4. Priorities

In some applications it is desirable to use a priority scheduling discipline for allocating the processor(s) to processes which are not waiting. Unless care is taken, the ordering implied by the assignment of priorities can be subverted by monitors. Suppose there are three priority levels (3 highest, 1 lowest), and three processes P_1, P_2, and P_3, one running at each level. Let P_1 and P_3 communicate using a monitor *M*. Now consider the following sequence of events:

1. P_1 enters *M*.
2. P_1 is preempted by P_2.
3. P_2 is preempted by P_3.
4. P_3 tries to enter the monitor, and waits for the lock.
5. P_2 runs again, and can effectively prevent P_3 from running, contrary to the purpose of the priorities.

A simple way to avoid this situation is to associate with each monitor the priority of the highest priority process which ever enters that monitor. Then whenever a process enters a monitor, its priority is temporarily increased to the monitor's priority. Modula solves the problem in an even simpler way—interrupts are disabled on entry to *M*, thus effectively giving the process the highest possible priority, as well as supplying the monitor lock for *M*. This approach fails if a page fault can occur while executing in *M*.

The mechanism is not free, and whether or not it is needed depends on the application. For instance, if only processes with adjacent priorities share a monitor, the problem described above

cannot occur. Even if this is not the case, the problem may occur rarely, and absolute enforcement of the priority scheduling may not be important.

5. Implementation

The implementation of processes and monitors is split more or less equally among the Mesa compiler, the runtime package, and the underlying machine. The compiler recognizes the various syntactic constructs and generates appropriate code, including implicit calls on built-in (that is, known to the compiler) support procedures. The runtime implements the less heavily used operations, such as process creation and destruction. The machine directly implements the more heavily used features, such as process scheduling and monitor entry/exit.

Note that it was primarily frequency of use, rather than cleanliness of abstraction, that motivated our division of labor between processor and software. Nonetheless, the split did turn out to be a fairly clean layering, in which the birth and death of processes are implemented on top of monitors and process scheduling.

5.1 The processor

The existence of a process is normally represented only by its stack of procedure activation records or *frames*, plus a small (10-byte) description called a *ProcessState.* Frames are allocated from a *frame heap* by a microcoded allocator. They come in a range of sizes that differ by 20 percent to 30 percent; there is a separate free list for each size up to a few hundred bytes (about 15 sizes). Allocating and freeing frames are thus very fast, except when more frames of a given size are needed. Because all frames come from the heap, there is no need to preplan the stack space needed by a process. When a frame of a given size is needed but not available, there is a *frame fault*, and the fault handler allocates more frames in virtual memory. Resident procedures have a private frame heap that is replenished by seizing real memory from the virtual memory manager.

The *ProcessState*s are kept in a fixed table known to the processor; the size of this table determines the maximum number of processes. At any given time, a *ProcessState* is on exactly one *queue.* There are four kinds of queues:

Ready queue. There is one ready queue, containing all processes that are ready to run.

Monitor lock queue. When a process attempts to enter a locked monitor, it is moved from the ready queue to a queue associated with the monitor lock.

Condition variable queue. When a process executes a WAIT, it is moved from the ready queue to a queue associated with the condition variable.

Fault queue. A fault can make a process temporarily unable to run; such a process is moved from the ready queue to a fault queue, and a fault handling process is notified.

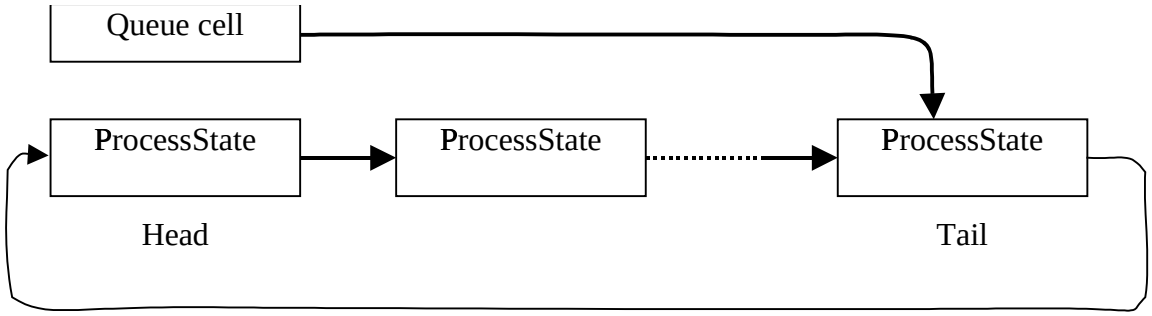

Figure 1: A process queue

Queues are kept sorted by process priority. The implementation of queues is a simple one way circular list, with the queue cell pointing to the *tail* of the queue (see Figure 1). This compact structure allows rapid access to both the head and the tail of the queue. Insertion at the tail and removal at the head are quick and easy; more general insertion and deletion involve scanning some fraction of the queue. The queues are usually short enough that this is not a problem. Only the ready queue grows to a substantial size during normal operation, and its patterns of insertions and deletions are such that queue scanning overhead is small.

The queue cell of the ready queue is kept in a fixed location known to the processor, whose fundamental task is to always execute the next instruction of the highest priority ready process. To this end, a check is made before each instruction, and a process switch is done if necessary. In particular, this is the mechanism by which interrupts are serviced. The machine thus implements a simple priority scheduler, which is preemptive between priorities and FIFO within a given priority.

Queues other than the ready list are passed to the processor by software as operands of instructions, or through a trap vector in the case of fault queues. The queue cells are passed by reference, since in general they must be updated (that is, the identity of the tail may change.) Monitor locks and condition variables are implemented as small records containing their associated queue cells plus a small amount of extra information: in a monitor lock, the actual lock; in a condition variable, the timeout interval and the wakeup-waiting switch.

At a fixed interval (about 20 times per second) the processor scans the table of ***ProcessState***s and notifies any waiting processes whose timeout intervals have expired. This special NOTIFY is tricky because the processor does not know the location of the condition variables on which such processes are waiting, and hence cannot update the queue cells. This problem is solved by leaving the queue cells out of date, but marking the processes in such a way that the next normal usage of the queue cells will notice the situation and update them appropriately.

There is no provision for time-slicing in the current implementation, but it could easily be added, since it has no effect on the semantics of processes.

5.2 The runtime support package

The *Process* module of the Mesa runtime package does creation and deletion of processes. This module is written (in Mesa) as a monitor, using the underlying synchronization machinery of the processor to coordinate the implementation of FORK and JOIN as the built-in entry procedures *Process.Fork* and *Process.Join*, respectively. The unused *ProcessState*s are treated as essentially normal processes which are all waiting on a condition variable called *rebirth*. A call of *Process.Fork* performs appropriate "brain surgery" on the first process in the queue and then notifies *rebirth* to bring the process to life: *Process.Join* synchronizes with the dying process and retrieves the results. The (implicitly invoked) procedure *Process.End* synchronizes the dying process with the joining process and then commits suicide by waiting on *rebirth.* An explicit call on *Process.Detach* marks the process so that when it later calls *Process.End,* it will simply destroy itself immediately.

The operations *Process.Abort* and *Process.Yield* are provided to allow special handling of processes that wait too long and compute too long, respectively. Both adjust the states of the appropriate queues, using the machine's standard queueing mechanisms. Utility routines are also provided by the runtime for such operations as setting a condition variable timeout and setting a process priority.

5.3 The compiler

The compiler recognizes the syntactic constructs for processes and monitors and emits the appropriate code (for example, a MONITORENTRY instruction at the start of each entry procedure, an implicit call of *Process.Fork* for each FORK). The compiler also performs special static checks to help avoid certain frequently encountered errors. For example, use of WAIT in an external procedure is flagged as an error, as is a direct call from an external procedure to an internal one. Because of the power of the underlying Mesa control structure primitives, and the care with which concurrency was integrated into the language, the introduction of processes and monitors into Mesa resulted in remarkably little upheaval inside the compiler.

5.4 Performance

Mesa's concurrent programming facilities allow the intrinsic parallelism of application programs to be represented naturally; the hope is that well structured programs with high global efficiency will result. At the same time, these facilities have nontrivial local costs in storage and/or execution time when compared with similar sequential constructs; it is important to minimize these costs, so that the facilities can be applied to a finer grain of concurrency. This section summarizes the costs of processes and monitors relative to other basic Mesa constructs, such as simple statements, procedures, and modules. Of course, the relative efficiency of an arbitrary concurrent program and an equivalent sequential one cannot be determined from these numbers alone; the intent is simply to provide an indication of the relative costs of various local constructs.

Storage costs fall naturally into data and program storage (both of which reside in swappable virtual memory unless otherwise indicated). The minimum cost for the existence of a Mesa module is 8 bytes of data and 2 bytes of code. Changing the module to a monitor adds 2 bytes of data and 2 bytes of code. The prime component of a module is a set of procedures, each of which

requires a minimum of an 8-byte activation record and 2 bytes of code. Changing a normal procedure to a monitor entry procedure leaves the size of the activation record unchanged, and adds 8 bytes of code. All of these costs are small compared with the program and data storage actually needed by typical modules and procedures. The other cost specific to monitors is space for condition variables; each condition variable occupies 4 bytes of data storage, while WAIT and NOTIFY require 12 bytes and 3 bytes of code, respectively.

The data storage overhead for a process is 10 bytes of resident storage for its *ProcessState*, plus the swappable storage for its stack of procedure activation records. The process itself contains no extra code, but the code for the FORK and JOIN which create and delete it together occupy 13 bytes, as compared with 3 bytes for a normal procedure call and return. The FORK/JOIN sequence also uses 2 data bytes to store the process value. In summary:

	Space (*bytes*)	
Construct	*data*	*code*
module	8	2
procedure	8	2
call + return	-	3
monitor	10	4
entry procedure	8	10
FORK+JOIN	2	13
process	10	0
condition variable	4	
WAIT	-	12
NOTIFY	-	3

For measuring execution times we define a unit called a *tick*: the time required to execute a simple instruction (for example, on a "one MIP" machine, one tick would be one microsecond). A tick is arbitrarily set at one-fourth of the time needed to execute the simple statement "$a \leftarrow b + c$" (that is, two loads, an add, and a store). One interesting number against which to compare the concurrency facilities is the cost of a normal procedure call (and its associated return), which takes 30 ticks if there are no arguments or results.

The cost of calling and returning from a monitor entry procedure is 50 ticks, about 70 percent more than an ordinary call and return. In practice, the percentage increase is somewhat lower, since typical procedures pass arguments and return results, at a cost of 24 ticks per item. A process switch takes 60 ticks; this includes the queue manipulations and all the state saving and restoring. The speed of WAIT and NOTIFY depends somewhat on the number and priorities of the processes involved, but representative figures are 15 ticks for a WAIT and 6 ticks for a NOTIFY. Finally, the minimum cost of a FORK/ JOIN pair is 1,100 ticks, or about 38 times that of a procedure call. To summarize:

Construct	*Time* (*ticks*)
simple instruction	1
call + return	30
monitor call + return	50
process switch	60
WAIT	15
NOTIFY, no one waiting	4
NOTIFY, process waiting	9
FORK+JOIN	1,100

On the basis of these performance figures, we feel that our implementation has met our efficiency goals, with the possible exception of FORK and JOIN. The decision to implement these two language constructs in software rather than in the underlying machine is the main reason for their somewhat lackluster performance. Nevertheless, we still regard this decision as a sound one, since these two facilities are considerably more complex than the basic synchronization mechanism, and are used much less frequently (especially JOIN, since the detached processes discussed in Section 2 have turned out to be quite popular).

6. Applications

In this section we describe the way in which processes and monitors are used by three substantial Mesa programs: an operating system, a calendar system using replicated databases, and an internetwork gateway.

6.1 Pilot: A general-purpose operating system

Pilot is a Mesa-based operating system [18] which runs on a large personal computer. It was designed jointly with the new language features and makes heavy use of them. Pilot has several autonomous processes of its own, and can be called by any number of client processes of any priority, in a fully asynchronous manner. Exploiting this potential concurrency requires extensive use of monitors within Pilot; the roughly 75 program modules contain nearly 40 separate monitors.

The Pilot implementation includes about 15 dedicated processes (the exact number depends on the hardware configuration); most of these are event handlers for three classes of events:

I/O interrupts. Naked notifies as discussed in Section 4.2.

Process faults. Page faults and other such events, signaled via fault queues as discussed in Section 5.1. Both client code and the higher levels of Pilot, including some of the dedicated processes, can cause such faults.

Internal exceptions. Missing entries in resident databases, for example, cause an appropriate high level “helper” process to wake up and retrieve the needed data from secondary storage.

There are also a few “daemon” processes, which awaken periodically and perform housekeeping chores (for example, swap out unreferenced pages). Essentially all of Pilot’s internal processes

and monitors are created at system initialization time (in particular, a suitable complement of interrupt handler processes is created to match the actual hardware configuration, which is determined by interrogating the hardware). The running system makes no use of dynamic process and monitor creation, largely because much of Pilot is involved in implementing facilities such as virtual memory which are themselves used by the dynamic creation software.

The internal structure of Pilot is fairly complicated, but careful placement of monitors and dedicated processes succeeded in limiting the number of bugs which caused deadlock; over the life of the system, somewhere between one and two dozen distinct deadlocks have been discovered, all of which have been fixed relatively easily without any global disruption of the system's structure.

At least two areas have caused annoying problems in the development of Pilot:

1. *The lack of mutual exclusion in the handling of interrupts*. As in more conventional interrupt systems, subtle bugs have occurred due to timing races between I/O devices and their handlers. To some extent, the illusion of mutual exclusion provided by the casting of interrupt code as a monitor may have contributed to this, although we feel that the resultant economy of mechanism still justifies this choice.

2. *The interaction of the concurrency and exception facilities*. Aside from the general problems of exception handling in a concurrent environment, we have experienced some difficulties due to the specific interactions of Mesa signals with processes and monitors (see Sections 3.1 and 3.4). In particular, the reasonable and consistent handling of signals (including UNWINDS) in entry procedures represents a considerable increase in the mental overhead involved in designing a new monitor or understanding an existing one.

6.2 Violet: A distributed calendar system

The Violet system [6, 7] is a distributed database manager which supports replicated data files, and provides a display interface to a distributed calendar system. It is constructed according to the hierarchy of abstractions shown in Figure 2. Each level builds on the next lower one by calling procedures supplied by it. In addition, two of the levels explicitly deal with more than one process. Of course, as any level with multiple processes calls lower levels, it is possible for multiple processes to be executing procedures in those levels as well.

The user interface level has three processes: *Display*, *Keyboard*, and *DataChanges*. The *Display* process is responsible for keeping the display of the database consistent with the views specified by the user and with changes occurring in the database itself. The other processes notify it when changes occur, and it calls on lower levels to read information for updating the display. *Display* never calls update operations in any lower level. The other two processes respond to changes initiated either by the user (*Keyboard*) or by the database (*DataChanges*). The latter process is FORKed from the *Transactions* module when data being looked at by Violet changes, and disappears when it has reported the changes to *Display*.

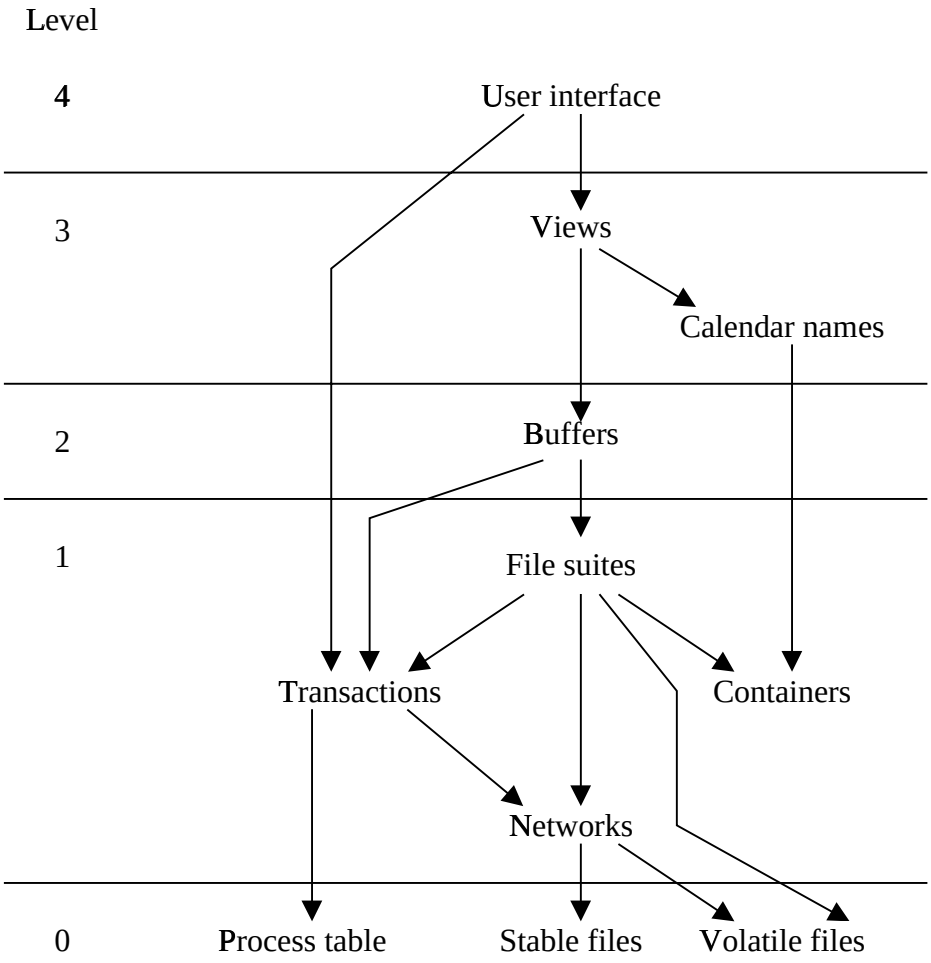

Figure 2: The internal structure of Violet

A more complex constellation of processes exists in ***FileSuites***, which constructs a single *replicated file* from a set of *representative* files, each containing data from some version of the replicated file. The representatives are stored in a transactional file system [11], so that each one is updated atomically, and each carries a version number. For each *FileSuite* being accessed, there is a monitor that keeps track of the known representatives and their version numbers. The replicated file is considered to be updated when all the representatives in a *write quorum* have been updated; the latest version can be found by examining a *read quorum*. Provided the sum of the read quorum and the write quorum is as large as the total set of representatives, the replicated file behaves like a conventional file.

When the file suite is created, it FORKs and detaches an *inquiry* process for each representative. This process tries to read the representative's version number, and if successful, reports the number to the monitor associated with the file suite and notifies the condition *CrowdLarger*. Any process trying to read from the suite must collect a read quorum. If there are not enough representatives present yet, it waits on *CrowdLarger*. The inquiry processes expire after their work is done.

When the client wants to update the *FileSuite*, it must collect a write quorum of representatives containing the current version, again waiting on *CrowdLarger* if one is not yet present. It then FORKS an *update* process for each representative in the quorum, and each tries to write its file. After FORKing the update processes, the client JOINS each one in turn, and hence does not proceed until all have completed. Because all processes run within the same transaction, the underlying transactional file system guarantees that either all the representatives in the quorum will be written, or none of them.

It is possible that a write quorum is not currently accessible, but a read quorum is. In this case the writing client FORKs a *copy* process for each representative which is accessible but is not up to date. This process copies the current file suite contents (obtained from the read quorum) into the representative, which is now eligible to join the write quorum.

Thus as many as three processes may be created for each representative in each replicated file. In the normal situation when the state of enough representatives is known, however, all these processes have done their work and vanished; only one monitor call is required to collect a quorum. This potentially complex structure is held together by a single monitor containing an array of representative states and a single condition variable.

6.3 Gateway: An internetwork forwarder

Another substantial application program that has been implemented in Mesa using the process and monitor facilities is an internetwork gateway for packet networks [2]. The gateway is attached to two or more networks and serves as the connection point between them, passing packets across network boundaries as required. To perform this task efficiently requires rather heavy use of concurrency.

At the lowest level, the gateway contains a set of device drivers, one per device, typically consisting of a high priority interrupt process, and a monitor for synchronizing with the device and with non-interrupt-level software. Aside from the drivers for standard devices (disk, keyboard, etc.) a gateway contains two or more drivers for Ethernet local broadcast networks [16] and/or common carrier lines. Each Ethernet driver has two processes, an interrupt process and a background process for autonomous handling of timeouts and other infrequent events. The driver for common carrier lines is similar, but has a third process which makes a collection of lines resemble a single Ethernet by iteratively simulating a broadcast. The other network drivers have much the same structure; all drivers provide the same standard network interface to higher level software.

The next level of software provides packet routing and dispatching functions. The *dispatcher* consists of a monitor and a dedicated process. The monitor synchronizes interactions between the drivers and the dispatcher process. The dispatcher process is normally waiting for the completion of a packet transfer (input or output); when one occurs, the interrupt process handles the interrupt, notifies the dispatcher, and immediately returns to await the next interrupt. For example, on input the interrupt process notifies the dispatcher, which dispatches the newly arrived packet to the appropriate *socket* for further processing by invoking a procedure associated with the socket.

The *router* contains a monitor that keeps a *routing* table mapping network names to addresses of other gateway machines. This defines the next "hop" in the path to each accessible remote

network. The router also contains a dedicated housekeeping process that maintains the table by exchanging special packets with other gateways. A packet is transmitted rather differently than it is received. The process wishing to transmit to a remote socket calls into the router monitor to consult the routing table, and then the same process calls directly into the appropriate network driver monitor to initiate the output operation. Such asymmetry between input and output is particularly characteristic of packet communication, but is also typical of much other I/O software.

The primary operation of the gateway is now easy to describe: When the arrival of a packet has been processed up through the level of the dispatcher, and it is discovered that the packet is addressed to a remote socket, the dispatcher forwards it by doing a normal transmission; that is, consulting the routing table and calling back down to the driver to initiate output. Thus, although the gateway contains a substantial number of asynchronous processes, the most critical path (forwarding a message) involves only a single switch between a pair of processes.

Conclusion

The integration of processes and monitors into the Mesa language was a somewhat more substantial task than one might have anticipated, given the flexibility of Mesa's control structures and the amount of published work on monitors. This was largely because Mesa is designed for the construction of large, serious programs, and processes and monitors had to be refined sufficiently to fit into this context. The task has been accomplished, however, yielding a set of language features of sufficient power that they serve as the only software concurrency mechanism on our personal computer, handling situations ranging from input/output interrupts to cooperative resource sharing among unrelated application programs.

Received June 1979; accepted September 1979: revised November 1979

References

1. *American National Standard Programming Language PL/1*. X3.53, American Nat. Standards Inst., New York, 1976.

2. Boggs, D.R. et al. Pup: An internetwork architecture. *IEEE Trans. on Communications* **28**, 4 (April 1980).

3. Brinch Hansen, P. *Operating System Principles*. Prentice-Hall, July 1973.

4. Brinch Hansen. P. The programming language Concurrent Pascal. *IEEE Trans. on Software Engineering* **1**,2 (June 1975), 199-207.

5. Dijkstra, E.W. Hierarchical ordering of sequential processes. In *Operating Systems Techniques,* Academic Press, 1972.

6. Gifford, D.K. Weighted voting for replicated data. *Operating Systems Review* **13**, 5 (Dec.1979), l50-l62.

7. Gifford. D.K. Violet, an experimental decentralized system. Integrated Office Systems **W**orkshop, IRIA, Rocquencourt, France, Nov. 1979 (also available as CSL report 79-12, **X**erox Research Center, Palo Alto, Calif.).

8. Hoare, C.A.R. Monitors: An operating system structuring concept. *Comm. ACM* **17**, 10 (Oct.1974), 549-557.

9. Hoare, C.A.R. Communicating sequential processes. *Comm. ACM* **21**, 8 (Aug.1978), 666-677.

10. **H**oward, J.H. Signaling in monitors. ***S****econd Int. Conf. on Software Engineering*, San Francisco, Oct.1976, 47-52.

11. **I**srael, J.E., Mitchell, J.G., and Sturgis, H.E. Separating data from function in a distributed file system. ***S****econd Int. Symposium on Operating Systems*, IRIA, Rocquencourt, France, Oct. 1978.

12. Keedy, J.J. On structuring operating systems with monitors. ***A****ustralian Computer J*. **10**, 1 (Feb.1978), 23-27 (reprinted in *Operating Systems Review* **13**, 1 (Jan.1979), 5-9).

13. **L**ampson, B.W., Mitchell, J.G., and Satterthwaite, E.H. On the transfer of control between contexts. *Lecture Notes in Computer Science* **19**, Springer, 1974, 181-203.

14. Lauer. H.E., and Needham. R.M. On the duality of operating system structures. ***S****econd Int. **S**ymposium on Operating Systems*, IRIA, Rocquencourt, France, Oct. 1978 (reprinted in *Operating Systems Review* **13,***2* (April 1979), 3-19).

15. **L**ister, AM., and Maynard. K.J. An implementation of monitors. ***S****oftware—Practice and Experience* **6**,3 (July 1976), 377-386.

16. Metcalfe. R.M., and Boggs, D.G. Ethernet: Packet switching for local computer networks. *Comm. ACM* **19,** *7* (July 1976), 395-403.

17. Mitchell. J.G., Maybury. W., and Sweet, R. *Mesa Language Manual*. Xerox Research Center, **P**alo Alto, Calif., 1979.

18. Redell, D., et al. Pilot: An operating system for a personal computer. *Comm. ACM* **23,***2* (Feb.1980).

19. Saltzer, J.H. *Traffic Control in a Multiplexed Computer System*. MAC-TR-30, MIT, July 1966.

20. Saxena, A.R., and Bredt, T.H. A structured specification of a hierarchical operating system. ***S****IGPLAN Notices* **10**, 6 (June 1975), 310-318.

21. **W**irth, N. Modula: A language for modular multi-programming. ***S****oftware—Practice and Experience* **7**, 1 (Jan.1977), 3-36.