Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	Dave Redell, Dick Sweet	Location	Palo Alto
Subject	Mesa 4.0 Process Update	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>PROCESS40.BRAVO

Mesa provides language support for concurrent execution of multiple processes. This allows programs that are inherently parallel in nature to be clearly expressed. The language also provides facilities for synchronizing such processes by means of entry to monitors and waiting on condition variables.

The next section discusses the forking and joining of concurrent process. Later sections deal with monitors, how their locks are specified, and how they are entered and exited. Condition variables are discussed, along with their associated operations.

10.1. Concurent execution, FORK and JOIN.

The FORK and JOIN statements allow parallel execution of two procedures. Their use also requires the new data type PROCESS. Since the Mesa process facilities provide considerable flexibility, it is easiest to understand them by first looking at a simple example.

10.1.1. A Process Example

Consider an application with a front-end routine providing interactive composition and editing of input lines:

```
ReadLine: PROCEDURE [s: STRING] RETURNS [CARDINAL] =
    BEGIN
    c: CHARACTER;
    s.length ← 0;
    DO
        c ← ReadChar[];
        IF ControlCharacter[c] THEN DoAction[c]
        ELSE AppendChar[s,c];
        IF c = CR THEN RETURN [s.length];
        ENDLOOP;
    END;
```

The call

```
n ← ReadLine[buffer];
```

will collect a line of user type-in up to a *CR* and put it in some string named *buffer*. Of

course, the caller cannot get anything else accomplished during the type-in of the line. If there is anything else that needs doing, it can be done concurrently with the type-in by *forking* to *ReadLine* instead of calling it:

```
p ← FORK ReadLine[buffer];
. . .
<concurrent computation>
. . .
n ← JOIN p;
```

This allows the statements labeled <concurrent computation> to proceed in parallel with user typing (clearly, the concurrent computation should not reference the string *buffer*). The FORK construct spawns a new process whose result type matches that of *ReadLine*. (*ReadLine* is referred to as the "root procedure" of the new process.)

```
p: PROCESS RETURNS [CARDINAL];
```

Later, the results are retrieved by the JOIN statement, which also deletes the spawned process. Obviously, this must not occur until both processes are ready (i.e. have reached the JOIN and the RETURN, respectively); this rendevous is synchronized automatically by the process facility.

Note that the types of the arguments and results of *ReadLine* are always checked at compile time, whether it is called or forked.

The one major difference between calling a procedure and forking to it is in the handling of signals; see section 10.5.1 for details.

10.1.2. Process Language Constructs

The declaration of a PROCESS is similar to the declaration of a PROCEDURE, except that only the return record is specified. The syntax is formally specified as follows:

```
TypeConstructor  ::= ... | ProcessTC
ProcessTC        ::= PROCESS ReturnsClause
ReturnsClause    ::= empty | RETURNS ResultList   -- from sec. 5.1.
ResultList       ::= FieldList                    -- from sec. 5.1.
```

Suppose that *f* is a procedure and *p* a process. In order to fork *f* and assign the resulting process to *p*, the **ReturnClause** of *f* and that of *p* must be compatible, as described in sec 5.2.

The syntax for the FORK and JOIN statements is straightforward:

```
Statement    ::= ... | JoinCall
Expression   ::= ... | ForkCall | JoinCall
ForkCall     ::= FORK Call
JoinCall     ::= JOIN Call
Call         ::= (see sections 5.4 and 8.2.1)
```

The **ForkCall** always returns a value (of type PROCESS) and thus a FORK cannot stand alone as a statement. Unlike a procedure call, which returns a RECORD, the value of the FORK cannot be discarded by writing an empty extractor. The action specified by the FORK is to spawn a

process parallel to the current one, and to begin it executing the named procedure.

The **JoinCall** appears as either a statement or an expression, depending upon whether or not the process being joined has an empty **ReturnsClause**. It has the following meaning: When the forked procedure has executed a RETURN *and* the JOIN is executed (in either order),

the returning process is deleted, and

the joining process receives the results, and continues execution.

A catchphrase can be attached to either a FORK or JOIN by specifying it in the **Call**. Note, nowever, that such a catchphrase does not catch signals incurred during the execution of the procedure; see section 10.5.1 for further details.

There are several other important similarities with normal procedure calls which are worth noting:

The types of all arguments and results are checked at compile time.

There is no *intrinsic* rule against multiple activations (calls and/or forks) of the same procedure coexisting at once. Of course, it is always possible to write procedures which will work incorrectly if used in this way, but the mechanism itself does not prohibit such use.

One expected pattern of usage of the above mechanism is to place a matching FORK/JOIN pair at the beginning and end of a single textual unit (i.e. procedure, compound statement, etc.) so that the computation within the textual unit occurs in parallel with that of the spawned process. This style is encouraged, but is *not* mandatory; in fact, the matching FORK and JOIN need not even be done by the same process. Care must be taken, of course, to insure that each spawned process is joined only once, since the result of joining an already deleted process is undefined. Note that the spawned process always begins and ends its life in the same textual unit (i.e. the target procedure of the FORK).

While many processes will tend to follow the FORK/JOIN paradigm, there will be others whose role is better cast as continuing provision of services, rather than one-time calculation of results. Such a "detached" process is never joined. If its lifetime is bounded at all, its deletion is a private matter, since it involves neither synchronization nor delivery of results. No language features are required for this operation; see the runtime documentation for the description of the system procedure provided for detaching a process.

10.2. Monitors

Generally, when two or more processes are cooperating, they need to interact in more complicated ways than simply forking and joining. Some more general mechanism is needed to allow orderly, synchronized interaction among processes. The interprocess synchronization mechanism provided in Mesa is a variant of *monitors* adapted from the work of Hoare, Brinch Hansen, and Dijkstra. The underlying view is that interaction among processes always reduces to carefully synchronized access to shared data, and that a proper vehicle for this interaction is one which unifies:

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

The Mesa monitor facility allows considerable flexibility in its use. Before getting into the details, let us first look at a slightly over-simplified description of the mechanism and a simple example. The remainder of this section deals with the basics of monitors (more complex uses are described in section 10.4); WAIT and NOTIFY are described in section 10.3.

10.2.1. An overview of Monitors

A *monitor* is a module instance. It thus has its own data in its global frame, and its own procedures for accessing that data. Some of the procedures are public, allowing calls into the monitor from outside. Obviously, conflicts could arise if two processes were executing in the same monitor at the same time. To prevent this, a *monitor lock* is used for mutual exclusion (i.e. to insure that only one process may be in each monitor at any one time). A call into a monitor (to an *entry procedure*) implicitly acquires its lock (waiting if necessary), and returning from the monitor releases it. The monitor lock serves to guarantee the integrity of the global data, which is expressed as the *monitor invariant* -- i.e an assertion defining what constitutes a "good state" of the data for that particular monitor. It is the responsibility of *every* entry procedure to restore the monitor invariant before returning, for the benefit of the next process entering the monitor.

Things are complicated slightly by the possibility that one process may enter the monitor and find that the monitor data, while in a good state, nevertheless indicates that that process cannot continue until some other process enters the monitor and improves the situation. The WAIT operation allows the first process to release the monitor lock and await the desired condition. The WAIT is performed on a *condition variable*, which is associated by agreement with the actual condition needed. When another process makes that condition true, it will perform a NOTIFY on the condition variable, and the waiting process will continue from where it left off (after reacquiring the lock, of course.)

For example, consider a fixed block storage allocator providing two entry procedures: *Allocate* and *Free*. A caller of *Allocate* may find the free storage exhausted and be obliged to wait until some caller of *Free* returns a block of storage.

```
StorageAllocator: MONITOR =
    BEGIN
    StorageAvailable: CONDITION;
    FreeList: POINTER;

    Allocate: ENTRY PROCEDURE RETURNS [p: POINTER] =
        BEGIN
        WHILE FreeList = NIL DO
            WAIT StorageAvailable
            ENDLOOP;
        p ← FreeList; FreeList ← p.next;
        END;

    Free: ENTRY PROCEDURE [p: POINTER] =
        BEGIN
        p.next ← FreeList; FreeList ← p;
        NOTIFY StorageAvailable
        END;
    END.
```

Note that it is clearly undesirable for two asynchonous processes to be executing in the *StorageAllocator* at the same time. The use of entry procedures for *Allocate* and *Free* assures mutual exclusion. The monitor lock is released while WAITing in *Allocate* in order to allow *Free* to be called (this also allows other processes to call *Allocate* as well, leading to several processes waiting on the queue for *StorageAvailable*).

10.2.2. Monitor Locks

The most basic component of a monitor is its *monitor lock*. A monitor lock is a predefined type, which can be thought of as a small record:

```
MONITORLOCK: TYPE = PRIVATE RECORD [locked: BOOLEAN, queue: Queue];
```

The monitor lock is private; its fields are never accessed explicitly by the Mesa programmer. Instead, it is used implicitly to synchronize entry into the monitor code, thereby authorizing access to the monitor data (and in some cases, other resources, such as I/O devices, etc.) The next section describes several kinds of monitors which can be constructed from this basic mechanism. In all of these, the idea is the same: during entry to a monitor, it is necessary to acquire the monitor lock by:

1. waiting (in the queue) until: *locked* = FALSE,
2. setting: *locked* ← TRUE.

10.2.3. Declaring monitor modules, ENTRY *and* INTERNAL *procedures*

In addition to a collection of data and an associated lock, a monitor contains a set of procedure that do operations on the data. *Monitor modules* are declared much like program or definitions modules; for example:

```
M: MONITOR [arguments] =
BEGIN
. . .
END.
```

The procedures in a monitor module are of three kinds:

Entry procedures

Internal procedures

External procedures

Every monitor has one or more *entry* procedures; these acquire the monitor lock when called, and are declared as:

```
P: ENTRY PROCEDURE [arguments] = . . .
```

The entry procedures will usually comprise the set of public procedures visible to clients of the monitor module. (There are some situations in which this is not the case; see external procedures, below). The usual Mesa default rules for PUBLIC and PRIVATE procedures apply.

Many monitors will also have *internal* procedures: common routines shared among the

several entry procedures. These execute with the monitor lock held, and may thus freely access the monitor data (including condition variables) as necessary. Internal procedures should be private, since direct calls to them from outside the monitor would bypass the acquisition of the lock (for monitors implemented as multiple modules, this is not quite right; see section 10.4, below). internal procedures can be called only from an entry procedure or another internal procedure. They are declared as follows:

```
Q: INTERNAL PROCEDURE [arguments] = . . .
```

The attributes ENTRY or INTERNAL may be specified on a procedure only in a monitor module. Section 10.2.4 describes how one declares an interface for a monitor.

Some monitor modules may wish to have *external* procedures. These are declared as normal non-monitor procedures:

```
R: INTERNAL PROCEDURE [arguments] = . . .
```

Such procedures are logically outside the monitor, but are declared within the same module for reasons of logical packaging. For example, a public external procedure might do some preliminary processing and then make repeated calls into the monitor proper (via a private entry procedure) before returning to its client. Being outside the monitor, an external procedure must *not* reference any monitor data (including condition variables), nor call any internal procedures. The compiler checks for calls to internal procedures and usage of the condition variable operations (WAIT, NOTIFY, etc.) within external procedures, but does not check for accesses to monitor data.

A fine point:

> Actually, unchanging read-only global variables *may* be accessed by external procedures; it is changeable monitor data that is strictly off-limits.

Generally speaking, a chain of procedure calls involving a monitor module has the general form:

Client procedure -- outside module
↓
External procedure(s) -- inside module but outside monitor
↓
Entry procedure -- inside monitor
↓
Internal procedure(s) -- inside monitor

Any deviation from this pattern is likely to be a mistake. A useful technique to avoid bugs and increase the readibility of a monitor module is to structure the source text in the corresponding order:

```
M: MONITOR =
  BEGIN
  <External procedures>
  <Entry procedures>
  <Internal procedures>
  <Initialization (main-body) code>
  END.
```

10.2.4. Interfaces to monitors

In Mesa, the attributes ENTRY and INTERNAL are associated with a procedure's body, not with its type. Thus they cannot be specified in a DEFINITIONS module. Typically, internal procedures are not exported anyway, although they may be for a multi-module monitor (see section 10.4.4). In fact, the compiler will issue a warning when the combination PUBLIC INTERNAL occurs.

From the client side of an interface, a monitor appears to be a normal program module, hence the keywords MONITOR and ENTRY do not appear. For example, a monitor *M* with entry procedures *P* and *Q* might appear as:

```
MDefs: DEFINITIONS =
    BEGIN
    M: PROGRAM [arguments];
    P, Q: PROCEDURE [arguments] RETURNS [results];
    .
    .
    END.
```

10.2.5. Interactions of processes and monitors

One interaction should be noted between the process spawning and monitor mechanisms as defined so far. If a process executing within a monitor forked to an internal procedure of the same monitor, the result would be two processes inside the monitor at the same time, which is the exact situation that monitors are supposed to avoid. The following rule is therefore enforced:

> A FORK may have as its target any procedure *except an internal procedure of a monitor.*

A fine point:

> In the case of a multi-module monitor (see section 10.4.4) calls to other monitor procedures through an interface cannot be checked for the INTERNAL attribute, since this information is not available in the interface (see section 10.2.4).

10.3. Condition Variables

Condition variables are declared as:

```
c: CONDITION;
```

The content of a condition variable is private to the process mechanism; condition variables may be accessed only via the operations defined below. It is important to note that it is the condition *variable* which is the basic construct; a condition (i.e. the contents of a condition variable) should *not* itself be thought of as a meaningful object; it may *not* be assigned to a condition variable, passed as a parameter, etc.

10.3.1. Wait, Notify, and Broadcast

A process executing in a monitor may find some condition of the monitor data which forces it to wait until another process enters the monitor and improves the situation. This can be accomplished using a condition variable, and the three basic operations: WAIT, NOTIFY, and

BROADCAST, defined by the following syntax:

```
Statement    ::= ... | WaitStmt | NotifyStmt
WaitStmt     ::= WAIT Variable OptCatchPhrase
NotifyStmt   ::= NOTIFY Variable | BROADCAST Variable
```

A condition variable *c* is always associated with some Boolean expression describing a desired state of the monitor data, yielding the general pattern:

Process waiting for condition:

```
WHILE ~BooleanExpression DO
   WAIT c
   ENDLOOP;
```

Process making condition true:

```
make BooleanExpression true; -- i.e. as side effect of modifying global data
NOTIFY c;
```

Consider the storage allocator example from section 10.2.1. In this case, the desired **BooleanExpression** is "*FreeList* # NIL". There are several important points regarding WAIT and NOTIFY, some of which are illustrated by that example:

WAIT always releases the lock while waiting, in order to allow entry by other processes, including the process which will do the NOTIFY (e.g. *Allocate* must not lock out the caller of *Free* while waiting, or a deadlock will result). Thus, the programmer is always obliged to restore the monitor invariant (return the monitor data to a "good state") before doing a WAIT.

NOTIFY, on the other hand, retains the lock, and may thus be invoked *without* restoring the invariant; the monitor data may be left in in an arbitrary state, so long as the invariant is restored before the next time the lock is released (by exiting an entry procedure, for example).

A NOTIFY directed to a condition variable on which no one is waiting is simply discarded. Moreover, the built-in test for this case is more efficient than any explicit test that the programmer could make to avoid doing the extra NOTIFY. (Thus, in the example above, *Free* always does a NOTIFY, without attempting to determine if it was actually needed.)

Each WAIT *must* be embedded in a loop checking the corresponding condition. (E.g. *Allocate*, upon being notified of the *StorageAvailable* condition, still loops back and tests again to insure that the freelist is actually non-empty.) This rechecking is necessary because the condition, even if true when the NOTIFY is done, may become false again by the time the awakened process gets to run. (Even though the freelist is always non-empty when *Free* does its NOTIFY, a third process could have called *Allocate* and emptied the freelist before the waiting process got a chance to inspect it.)

Given that a process awakening from a WAIT must be careful to recheck its desired condition, the process doing the NOTIFY can be somewhat more casual about insuring that the condition is actually true when it does the NOTIFY. This leads to the notion

of a *covering condition variable*, which is notified whenever the condition desired by the waiting process is *likely* to be true; this approach is useful if the expected cost of false alarms (i.e. extra wakeups that test the condition and wait again) is lower than the cost of having the notifier always know precisely what the waiter is waiting for.

The last two points are somewhat subtle, but quite important; condition variables in Mesa act as *suggestions* that their associated Boolean expressions are likely to be true and should therefore be rechecked. They do *not* guarantee that a process, upon awakening from a WAIT, will necessarily find the condition it expects. The programmer should never write code which implicitly assumes the truth of some condition simply because a NOTIFY has occurred.

It is often the case that the user will wish to notify *all* processes waiting on a condition variable. This can be done using:

```
    :
BROADCAST c;
```

This operation can be used when several of the waiting processes should run, or when *some* waiting process should run, but not necessarily the head of the queue.

Consider a variation of the *StorageAllocator* example:

```
StorageAllocator: MONITOR =
    BEGIN
    StorageAvailable: CONDITION;
    . . .

    Allocate: ENTRY PROCEDURE [size: CARDINAL] RETURNS [p: POINTER] =
        BEGIN
        UNTIL <storage chunk of size words is available> DO
            WAIT StorageAvailable
            ENDLOOP;
        p ← <remove chunk of size words>;
        END;

    Free: ENTRY PROCEDURE [p: POINTER, size: CARDINAL] =
        BEGIN
        . . .
        <put back storage chunk of size words>
        . . .
        BROADCAST StorageAvailable
        END;
    END.
```

In this example, there may be several processes waiting on the queue of *StorageAvailable*, each with a different *size* requirement. It is not sufficient to simply NOTIFY the head of the queue, since that process may not be satisfied with the newly available storage while another waiting process might be. This is a case in which BROADCAST is needed instead of NOTIFY.

An important rule of thumb: *it is always correct to use a* BROADCAST. NOTIFY should be used instead of BROADCAST if *both* of the following conditions hold:

It is expected that there will typically be several processes waiting in the condition variable queue (making it expensive to notify all of them with a BROADCAST), and

It is known that the process at the head of the condition variable queue will always be the right one to respond to the situation (making the multiple notification unnecessary);

If both of these conditions are met, a NOTIFY is sufficient, and may represent a significant efficiency improvement over a BROADCAST. The allocator example in section 10.2.1 is a situation in which NOTIFY is preferrable to BROADCAST.

As described above, the condition variable mechanism, and the programs using it, are intended to be robust in the face of "extra" NOTIFYs. The next section explores the opposite problem: "missing" NOTIFYs.

10.3.2. Timeouts

One potential problem with waiting on a condition variable is the possibility that one may wait "too long." There are several ways this could happen, including:

- Hardware error (e.g. "lost interrupt")
- Software error (e.g. failure to do a NOTIFY)
- Communication error (e.g. lost packet)

To handle such situations, waits on condition variables are allowed to *time out*. This is done by associating a *timeout interval* with each condition variable, which limits the delay that a process can experience on a given WAIT operation. If no NOTIFY has arrived within this time interval, one will be generated automatically. The Mesa language does not currently have a facility for setting the timeout field of a CONDITION variable. See the runtime documentation for the description of the system procedure provided for this operation.

The waiting process will perceive this event as a normal NOTIFY. (Some programs may wish to distinguish timeouts from normal NOTIFYs; this requires checking the time as well as the desired condition on each iteration of the loop.)

No facility is provided to time out waits for monitor locks. This is because there would be, in general, no way to recover from such a timeout.

10.4. More about Monitors

The next few sections deal with the full generality of monitor locks and monitors.

10.4.1. The LOCKS Clause

Normally, a monitor's data comprises its global variables, protected by the special global variable *LOCK*:

LOCK: MONITORLOCK;

This implicit variable is declared automatically in the global frame of any module whose heading is of the form:

M: MONITOR [*arguments*] IMPORTS . . . EXPORTS . . . =

In such a monitor it is generally not necessary to mention *LOCK* explicitly at all. For more general use of the monitor mechanism, it is necessary to declare at the beginning of the monitor module exactly which MONITORLOCK is to be acquired by entry procedures. This declaration appears as part of the program type constructor that is at the head of the module. The syntax is as follows:

```
ProgramTC     ::= ... | MONITOR ParameterList ReturnsClause LocksClause
LocksClause   ::= empty | LOCKS Expression |
                  LOCKS Expression USING identifier : TypeSpecification
```

If the **LocksClause** is empty, entry to the monitor is controlled by the distinguished variable *LOCK* (automatically supplied by the compiler). Otherwise, the **LocksClause** must designate a variable of type MONITORLOCK, a record containing a distinguished lock field (see section 10.4.2), or a pointer that can be dereferenced (perhaps several times) to yield one of the preceding. If a **LocksClause** is present, the compiler does not generate the variable *LOCK*.

If the USING clause is absent, the lock is located by evaluating the LOCKS expression in the context of the monitor's main body; i.e., the monitor's parameters, imports, and global variables are visible, as are any identifiers made accessible by a global OPEN. Evaluation occurs upon entry to, and again upon exit from, the entry procedures (and for any WAITs in entry or internal procedures). The location of the designated lock can thus be affected by assignments within the procedure to variables in the LOCKS expression. To avoid disaster, it is essential that each reevaluation yield a designator of the same MONITORLOCK. This case is described further in section 10.4.4.

If the USING clause is present, the lock is located in the following way: every entry or internal procedure must have a parameter with the same identifier and a compatible type as that specified in the USING clause. The occurrences of that identifier in the LOCKS clause are bound to that procedure parameter in every entry procedure (and internal procedure doing a WAIT). The same care is necessary with respect to reevaluation; to emphasize this, the distinguished argument is treated as a read-only value within the body of the procedure. See section 10.4.5 for further details.

10.4.2. Monitored Records

For situations in which the monitor data cannot simply be the global variables of the monitor module, a *monitored record* can be used:

```
r: MONITORED RECORD [x: INTEGER, . . . ];
```

A monitored record is a normal Mesa record, except that it contains an automatically declared field of type MONITORLOCK. As usual, the monitor lock is used implicitly to synchronize entry into the monitor code, which may then access the other fields in the monitored record. The fields of the monitored record must *not* be accessed except from within a monitor which first acquires its lock. In analogy with the global variable case, the monitor lock field in a monitored record is given the special name *LOCK*; generally, it need not be referred to explicitly (except during initialization; see section 10.6).

A fine point:

A more general form of monitor lock declaration is discussed in section 10.4.6

CAUTION: If a monitored record is to be passed around (e.g. as an argument to a procedure) this should always be done by reference using a POINTER TO MONITORED RECORD. Copying a

monitored record (e.g. passing it by value) will generally lead to chaos.

10.4.3. Monitors and module instances

Even when all the procedures of a monitor are in one module, it is not quite correct to think of the module and the monitor as identical. For one thing, a monitor module, like an ordinary program module, may have several instances. In the most straightforward case, each instance constitutes a separate monitor. More generally, through the use of monitored records, the number of monitors may be larger or smaller than the number of instances of the corresponding module(s). The crucial observation is that in all cases:

> *There is a one-to-one correspondence between monitors and monitor locks.*

The generalization of monitors through the use of monitored records tends to follow one of two patterns:

> *Multi-module monitors*, in which several module instances implement a single monitor.
>
> *Object monitors*, in which a single module instance implements several monitors.

A fine point:

> These two patterns are *not* mutually exclusive; multi-module object monitors are possible, and may occasionally prove necessary.

10.4.4. Multi-module monitors

In implementing a monitor, the most obvious approach is to package all the data and procedures of the monitor within a single module instance (if there are multiple instances of such a module, they constitute separate monitors and share nothing except code.) While this will doubtless be the most common technique, the monitor may grow too large to be treated as a single module.

Typically, this leads to multiple modules. In this case the mechanics of constructing the monitor are changed somewhat. There must be a central location that contains the monitor lock for the monitor implemented by the multiple modules. This can be done either by using a MONITORED RECORD or by choosing one of the modules to be the "root" of the monitor. Consider the following example:

```
BigMonRoot: MONITOR IMPORTS . . . EXPORTS . . . =
    BEGIN
    monitorDatum1: . . .
    monitorDatum2: . . .
    . . .
    p1: PUBLIC ENTRY PROCEDURE . . .
    . . .
    END.

BigMonA: MONITOR
    LOCKS root      -- could equivalently say root.LOCK
    IMPORTS root: BigMonRoot . . . EXPORTS . . . =
    BEGIN
    . . .
```

```
    p2: PUBLIC ENTRY PROCEDURE . . .
        x ← root.monitorDatum1;  -- access the protected data of the monitor
    . . .
    END.

BigMonB: MONITOR
    LOCKS root
    IMPORTS root: BigMonRoot . . . EXPORTS . . . =
    BEGIN OPEN root;
    . . .
    p3: PUBLIC ENTRY PROCEDURE . . .
        monitorDatum2 ← . . .;  -- access the protected data via an OPEN
    . . .
    END.
```

The monitor *BigMon* is implemented by three modules. The modules *BigMonA* and *BigMonB* have a LOCKS clause to specify the location of the monitor lock: in this case, the distinguished variable *LOCK* in *BigMonRoot*. When any of the entry procedures *p*1, *p*2, or *p*3 is called, this lock is acquired (waiting if necessary), and is released upon returning. The reader can verify that no two independent processes can be in the monitor at the same time.

Another means of implementing multi-module monitors is by means of a MONITORED RECORD. Use of OPEN allows the fields of the record to be referenced without qualification. Such a monitor is written as:

```
MonitorData: TYPE = MONITORED RECORD [x: INTEGER, . . . ];

MonA: MONITOR [pm: POINTER TO MonitorData]
LOCKS pm
IMPORTS . . .
EXPORTS . . . =
BEGIN OPEN pm;
P: ENTRY PROCEDURE [. . .] =
    BEGIN
    . . .
    x ← x+1; -- access to a monitor variable
    . . .
    END;
. . .
END.
```

The LOCKS clause in the heading of this module (and each other module of this monitor) leads to a MONITORED RECORD. Of course, in all such multi-module monitors, the LOCKS clause will involve one or more levels of indirection (POINTER TO MONITORED RECORD, etc.) since passing a monitor lock by value is not meaningful. As usual, Mesa will provide one or more levels of automatic dereferencing as needed.

More generally, the target of the LOCKS clause can evaluate to a MONITORLOCK (i.e. the example above is equivalent to writing "LOCKS *pm.LOCK*").

CAUTION: The meaning of the target expression of the LOCKS clause *must not change* between the call to the entry procedure and the subsequent return (i.e. in the above example, changing *pm* would invariably be an error) since this would lead to a different monitor lock being released than was acquired, resulting in total chaos.

There are a few other issues regarding multi-module monitors which arise any time a tightly coupled piece of Mesa code must be split into multiple module instances and then spliced back together. For example:

> If the lock is in a MONITORED RECORD, the monitor data will probably need to be in the record also. While the global variables of such a multi-module monitor are covered by the monitor lock, they do *not* constitute monitor data in the normal sense of the term, since they are not uniformly visible to all the module instances.
>
> Making the internal procedures of a multi-instance monitor PRIVATE will not work if one instance wishes to call an internal procedure in another instance. (Such a call is perfectly acceptable so long as the caller already holds the monitor lock). Instead, a second interface (hidden from the clients) is needed as part of the "glue" holding the monitor together. Note however, that Mesa cannot currently check that the procedure being called through the interface is an internal one (see section 10.2.4).
>
> A fine point:
>
> > The compiler will complain about the PUBLIC INTERNAL procedures, but this is just a warning.

10.4.5. Object monitors

Some applications deal with *objects,* implemented, say, as records named by pointers. Often it is necessary to insure that operations on these objects are *atomic*, i.e., once the operation has begun, the object will not be otherwise referenced until the operation is finished. If a module instance provides operations on some class of objects, the simplest way of guaranteeing such atomicity is to make the module instance a monitor. This is logically correct, but if a high degree of concurrency is expected, it may create a bottleneck; it will serialize the operations on *all* objects in the class, rather than on *each* of them individually. If this problem is deemed serious, it can be solved by implementing the objects as monitored records, thus effectively creating a separate monitor for each object. A single module instance can implement the operations on all the objects as entry procedures, each taking as a parameter the object to be locked. The locking of the parameter is specified in the module heading via a **LocksClause** with a USING clause. For example:

```
ObjectRecord: TYPE = MONITORED RECORD [ . . . ];

ObjectHandle: TYPE = POINTER TO ObjectRecord;

ObjectManager: MONITOR [arguments]
    LOCKS object USING object: ObjectHandle
    IMPORTS . . .
    EXPORTS . . . =
    BEGIN
    Operation: PUBLIC ENTRY PROCEDURE [object: ObjectHandle, . . . ] =
        BEGIN
        . . .
        END;
    . . .
    END.
```

Note that the argument of USING is evaluated in the scope of the arguments to the entry procedures, rather than the global scope of the module. In order for this to make sense, each entry procedure, and each internal procedure that does a WAIT, must have an argument which

matches exactly the name and type specified in the USING subclause. All other components of the argument of LOCKS are evaluated in the global scope, as usual.

As with the simpler form of LOCKS clause, the target may be a more complicated expression and/or may evaluate to a monitor lock rather than a monitored record. For example:

LOCKS *p.q.LOCK* USING *p*: POINTER TO *ComplexRecord* . . .

CAUTION: Again, the meaning of the target expression of the LOCKS clause *must not change* between the call to the entry procedure and the subsequent return. (I.e. in the above example, changing *p* or *p.q* would almost surely be an error.)

CAUTION: It is important to note that global variables of object monitors are very dangerous; they are *not* covered by a monitor lock, and thus do *not* constitute monitor data. If used at all, they must be set only at module initialization time and must be read-only thereafter.

10.4.6. Explicit declaration of monitor locks

It is possible to declare monitor locks explicitly:

myLock: MONITORLOCK;

The normal cases of monitors and monitored records are essentially stylized uses of this facility via the automatic declaration of *LOCK*, and should cover all but the most obscure situations. For example, explicit delarations are useful in defining MACHINE DEPENDENT monitored records. (Note that the LOCKS clause becomes mandatory when an explicitly declared monitor lock is used.) More generally, explicit declarations allow the programmer to declare records with several monitor locks, declare locks in local frames, and so on; this flexibility can lead to a wide variety of subtle bugs, hence use of the standard constructs whenever possible is strongly advised.

10.5. Signals

10.5.1. Signals and Processes

Each process has its own call stack, down which signals propagate. If the signaller scans to the bottom of the stack and finds no catch phrase, the signal is propagated to the debugger. The important point to note is that forking to a procedure is different from calling it, in that the forking creates a gap across which signals cannot propagate. This implies that in practice, one cannot casually fork to any arbitrary procedure. The only suitable targets for forks are procedures which catch any signals they incur, and which never generate any signals of their own.

10.5.2. Signals and Monitors

Signals require special attention within the body of an entry procedure. A signal raised with the monitor lock held will propagate without releasing the lock and possibly invoke arbitrary computations. For errors, this can be avoided by using the RETURN WITH ERROR construct.

RETURN WITH ERROR *NoSuchObject*;

Recall from Chapter 8 that this statement has the effect of removing the currently executing frame from the call chain before issuing the ERROR. If the statement appears within an entry procedure, the monitor lock is released before the error is started as well. Naturally, the monitor invariant must be restored before this operation is performed.

For example, consider the following program segment:

```
Failure: ERROR [kind: CARDINAL] = CODE;

Proc: ENTRY PROCEDURE [. . .] RETURNS [c1, c2: CHARACTER] =
    BEGIN
    ENABLE UNWIND => . . .
    . . .
    IF cond1 THEN ERROR Failure[1];
    IF cond2 THEN RETURN WITH ERROR Failure[2];
    . . .
    END;
```

Execution of the construct ERROR *Failure*[1] raises a signal that propagates until some catch phrase specifies an exit. At that time, unwinding begins; the catch phrase for UNWIND in *Proc* is executed and then *Proc*'s frame is destroyed. Within an entry procedure such as *Proc*, the lock is held until the unwind (and thus through unpredictable computation performed by catch phrases).

Execution of the construct RETURN WITH ERROR *Failure*[2] releases the monitor lock and destroys the frame of *Proc* before propagation of the signal begins. Note that the argument list in this construct is determined by the declaration of *Failure* (not by *Proc*'s RETURNS clause). The catch phrase for UNWIND is not executed in this case. The signal *Failure* is actually raised by the system, after which *Failure* propagates as an ordinary error (beginning with *Proc*'s caller).

When the RETURN WITH ERROR construct is used from within an internal procedure, the monitor lock is *not* released; RETURN WITH ERROR will release the monitor lock in precisely those cases that RETURN will.

Another important issue regarding signals is the handling of UNWINDS; any entry procedure that may experience an UNWIND must catch it and clean up the monitor data (restore the monitor invariant):

```
P: ENTRY PROCEDURE [ ... ] =
    BEGIN ENABLE UNWIND => BEGIN <restore invariant> END;
    .
    .
    .
    END;
```

At the end of the UNWIND catchphrase, the compiler will append code to release the monitor lock before the frame is unwound. It is important to note that a monitor always has at least one cleanup task to perform when catching an UNWIND signal: the monitor lock must be released. To this end, the programmer should be sure to place an enable-clause on the body of every entry procedure that might evoke an UNWIND (directly or indirectly). If the monitor invariant is already satisfied, no further cleanup need be specified, but *the null catch-phrase must be written* so that the compiler will generate the code to unlock the monitor:

```
BEGIN ENABLE UNWIND => NULL;
```

This should be omitted *only* when it is certain that no UNWINDs can occur.

Another point is that signals caught by the **OptCatchPhrase** of a WAIT operation should be thought of as occurring after reacquisition of the monitor lock. Thus, like all other monitor code, catch phrases within a monitor are always executed with the monitor lock held.

10.6. Initialization

When a new monitor comes into existence, its monitor data will generally need to be set to some appropriate initial values; in particular, the monitor lock and any condition variables must be initialized. As usual, Mesa takes responsibility for initializing the simple common cases; for the cases not handled automatically, it is the responsibility of the programmer to provide appropriate initialization code, and to arrange that it be executed at the proper time. The two types of initialization apply in the following situations:

Monitor data in global variables can be initialized using the normal Mesa initial value constructs in declarations. Monitor locks and condition variables in the global frame will also be initialized automatically (although in this case, the programmer does not write any explicit initial value in the declaration).

Monitor data in records must be initialized by the programmer. System procedures must be used to initialize the monitor lock and condition variables. See the runtime documentation for the descriptions of appropriate procedures.

A fine point:

If a variable containing a record is declared in a frame, it is normally possible to initialize it in the declaration (i.e. using a constructor as the initial value); however, this does *not* apply if the record contains monitor locks or condition variables, which must be initialized via calls to system procedures.

Since initialization code modifies the monitor data, it must have exclusive access to it. The programmer should insure this by arranging that the monitor not be called by its client processes until it is ready for use.