Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	Dave Redell, John Wick	Location	Palo Alto
Subject	Mesa 4.0 Change Summary	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>SUMMARY40.BRAVO

This memo outlines changes made in Mesa since the last release (October 17, 1977).

References

The following documents can be found on [IRIS]<MESA>DOC>; all files are in Bravo format. Hardcopy is available through your support group; in addition, the PRESS files MESA40A, MESA40B, and MESA40C are a compilation of this material (about 75 pages).

Mesa 4.0 Change Summary. SUMMARY40.BRAVO

Mesa 4.0 Compiler Update. COMPILER40.BRAVO, ARITHMETIC40.BRAVO

Mesa 4.0 Process Update. PROCESS40.BRAVO

Mesa 4.0 Binder Update. BINDER40.BRAVO

Mesa 4.0 System Update. SYSTEM40.BRAVO

Mesa 4.0 Microcode Update. MICROCODE40.BRAVO

Mesa 4.0 Debugger Update. DEBUGGER40.BRAVO

The section on processes is a preliminary draft of a new chapter of the *Mesa Language Manual* (which will be sent to the printer shortly); thanks are due to Dave Redell and the *Pilot Functional Specification* for contributing much of this material.

The MESA>DOC directory also includes new versions of the *Mesa System Documentation* and the *Mesa Debugger Documentation* (the relevant PRESS files are SYSTEM1, SYSTEM2, and DEBUGGER).

Highlights

The primary emphasis in this release has been on three areas: implementation of features required by Pilot and Dstar applications for effective use of the new machine architecture (processes, monitors, long pointers, etc.), reduction of overhead in the basic system structures and improved performance of the Mesa runtime environment (faster microcode, smaller global frames, more efficient memory management), and extension of the debugger's capabilities (primarily an interpreter for a subset of the Mesa language).

The primary impact of Mesa 4.0 on existing systems is in the area of concurrent programming. A brief intoduction to the new process mechanism appears below. It is intended to present enough information to enable programmers to experiment with the new features of the language and the runtime system. However, before attempting to revise or redesign existing systems to use these facilities, programmers are urged to carefully examine the material in the *Mesa 4.0 Process Update* and the *Mesa System Documentation.*

Warning: Because Pilot will be available soon, the Alto/Mesa operating system software has not been revised and redesigned to fully exploit the capabilities of the new process mechanism. In particular, arbitrary preemptive processes are not supported, and the restrictions of Mesa 3.0 on processes running at interrupt level still apply.

A Brief Introduction to Processes in Mesa

Mesa 4.0 introduces three new facilities for concurrent programming:

Processes, which provide the basic framework for concurrent programming.

Monitors, which provide the fundamental interprocess synchronization facility.

Condition variables, which build upon monitors to provide more flexible forms of interprocess synchronization.

As compared with the mechanisms provided in earlier releases of Mesa, the new concurrency facilities are more extensive, and are much more thoroughly integrated into the language. The purpose of the new facilities is to allow easy use of concurrency as a basic control structure in Mesa programs. Concurrency can be an important consideration in progam design, especially when input/output or user interactions may cause unpredictable delays.

Processes

For 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;
```

Thus, the call:

```
n ← ReadLine[s];
```

would collect a line of user typing up to a CR and return it to the caller. Of course, the caller cannot get anything else accomplished during the type-in of the line. If there was anything else that needed doing, it could be done concurrently with the type-in by *forking* to ReadLine instead of calling it:

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

This would allow the statements labeled <concurrent computation> to proceed in parallel with user typing. The FORK statement 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.

Monitors

Further investigation of ReadLine reveals another example of interprocess interaction; the ReadChar routine it uses inspects an input character buffer, which is filled by an independent dedicated keyboard process. (Such dedicated processes replace the "hard processes" of earlier releases of Mesa.) To avoid conflict over the buffer, appropriate synchronization is needed. A *monitor* can be used to insure that neither process will ever access the buffer while the other has it in a "bad state" (e.g. inconsistent pointers, etc.). The keyboard monitor might look like:

```
Keyboard: MONITOR =
BEGIN
buffer: STRING;
ReadChar: PUBLIC ENTRY PROCEDURE RETURNS [c: CHARACTER] =
  BEGIN
  c ← <get character from buffer>
  END;
PutChar: PUBLIC ENTRY PROCEDURE [c: CHARACTER] =
  BEGIN
  <put c in buffer>
  END;
END.
```

The keyword MONITOR confers upon the Keyboard module some special properties. The most fundamental is the presence of *entry* procedures, identified by the keyword ENTRY. These procedures have the property that calls on them are *mutually exclusive*; that is, a new call cannot commence while any previous call is in progress. In effect, the monitor module is made temporarily private to a single process, and any other processes wishing to use it are delayed until the first process is finished. In this example, the client's call to ReadChar and the keyboard process' call to PutChar are guaranteed mutually exclusive access to the buffer.

Condition variables

As long as it finds some characters in the buffer, ReadChar as shown above will work correctly without conflict over the buffer. If it finds the buffer empty, however, it cannot

simply loop in the monitor waiting for a character to arrive; not only would this be inefficient, but it would lock out the keyboard process from ever delivering the desired next character! What is needed is some way for ReadChar to pause and release the mutual exclusion temporarily until PutChar has delivered the next character. This facility is provided by *condition variables.* Condition variables serve as the basic building blocks out of which the programmer can fashion whatever generalized synchronization machinery proves necessary in a given situation. For example, the Keyboard monitor can be modified to use the WAIT and NOTIFY operations on condition variables as follows:

```
Keyboard: MONITOR =
BEGIN
buffer: STRING;
bufferNonEmpty: CONDITION;
ReadChar: PUBLIC ENTRY PROCEDURE RETURNS [c: CHARACTER] =
  BEGIN
  WHILE <buffer empty> DO
    WAIT bufferNonEmpty
    ENDLOOP;
  c ← <get character from buffer>
  END;
PutChar: PUBLIC ENTRY PROCEDURE [c: CHARACTER] =
  BEGIN
  <put c in buffer>
  NOTIFY bufferNonEmpty;
  END;
END.
```

Note that the WAIT statement is embedded in a WHILE-loop which repeatedly tests for the desired condition. *This is the only recommended usage pattern for the WAIT statement.* In particular, it would have been incorrect to replace the loop above by:

```
IF <buffer empty> THEN WAIT bufferNonEmpty;
c ← <get character from buffer>
```

This rule exemplifies a fundamental property of condition variables in Mesa: a condition variable always corresponds to some Boolean expression describing a desired state of the monitor data, and suggests that any interested process(es) might do well to reevaluate it. *It does not guarantee that the Boolean expression has become true,* hence programmers should *never* write programs (such as the fragment above) that implicitly assume the truth of the desired condition upon awakening from a WAIT.

Priorities

The set of existing processes grows and shrinks dynamically as FORKs and JOINs occur. At any given time, some of the processes are *ready* and compete for use of the processor. The choice of which one to run is done on the basis of priority. A process starts life with the priority of its parent (who executed the FORK), and may change its own priority by calling SetPriority.

CAUTION: *Use of multiple priorities in the Alto/Mesa implementation is severely restricted.* Any process running at other than the default priority (currently, 1) is forbidden to use many of the standard runtime support features of the Mesa environment. In practice, this means that non-standard priorities should be used only for interrupt handling, while all

"normal" processing takes place concurrently at the default priority level.

More general features

More complex situations will sometimes require more flexible use of the concurrency facilities. Such use involves more complicated rules and syntactic constructs, which are described in the *Mesa 4.0 Process Update.*

Distribution:
Mesa Users
Mesa Group