LAMPSON80
Lampson, B.W. and Redell D.D.; Experience with Processes and Monitors in Mesa; CACM 23, 2 (February 1980) pp. 105-117.

The Mesa programming language was designed in the mid-1970s at the Xerox Palo Alto Research Center for the "construction of large, serious programs" in a wide range of application areas. To facilitate concurrent programming, processes and monitors were added later to the language. The authors present an introduction to the concurrency facilities in Mesa and reflect on their design decisions in light of outstanding research issues concerning processes and interprocess communication.

The major issues addressed include: (1) program structure (processes must fit into the module-based scheme); (2) process creation (dynamic vs. static); (3) monitor creation (dynamic vs. static); (4) WAIT statement semantics in a nested monitor call; (5) alternative methods for resuming waiting processes (timeouts, aborts, broadcasts); (6) the precise semantics of waiting on a condition variable; (7) the interaction between monitors and priority scheduling of processes; and (8) fitting input/output devices into the framework of monitors and condition variables.

Mesa considers the creation of a new process as a special procedure activation which executes concurrently with its caller. *Any* procedure (except an internal procedure of a monitor) may be called in this way. The caller issues an explicit FORK command to create the process and a JOIN command to allow the caller and callee to merge, with the callee returning results. The DETACH command frees the created process from its creator, so that the DETACHed process dies on its own, without synchronizing with the caller. A process in Mesa is treated as a value in the language, which can be assigned, passed as a parameter, and in general treated like any other value. This raises the problem of dangling references to pointers; no protection is provided. No special declaration of a procedure to be involved as a process is needed (in contrast to Concurrent Euclid [Holt 1983], for example). The authors claim that process creation and destruction is "moderate" in cost, and the cost in storage is reported to be twice the minimum cost of a procedure instance.

Monitors in Mesa are, as the usual notion suggests, tools for implementing synchronization constraints on processes for access to shared data. Mesa monitors are usually cast as modules, having three kinds of procedures: *entry* (the externally visible entry points into the monitor); *internal* (private to the monitor); and *external* (publicly visible but not required to execute in a manner mutually exclusive from the entry procedures).

The notion of acquiring and releasing the monitor lock is discussed in the light of exceptions and nested WAIT statements. A type constructor called a *monitored record* is introduced; it is exactly like an ordinary record, except that it has a monitor lock and is intended for use as the protected data of a monitor. This is to allow for control over multiple independent data objects within a single monitor (instead of requiring the programmer to create multiple instances of the monitor module).

The Mesa design does not include signalling of condition variables as in Hoare's definition of monitors [Hoare 1974], in which a process waiting on a condition must run immediately when another process signals that variable, and the signalling process in turn runs as soon as the waiter leaves the monitor. In Mesa, 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; some process waiting on the condition will resume at some convenient future time. When the waiting process resumes, it will reacquire the monitor lock but does *not* have the guarantee that some other process has not entered the monitor after the waiting process was notified; the waiting process must therefore reevaluate the waiting condition each time it resumes. This laissez-faire approach provides the waiter with weaker guarantees (only the monitor invariant) but saves process switches. This approach leads to three other process wakeup methods: (1) *timeout*: a process which has been waiting for some timeout interval on a condition variable will resume regardless of whether the condition has been notified. (2) *abort*: a process may be aborted at any time by executing Abort[process], which raises the Aborted exception when the process next waits. The aborted process is, however, free to ignore the abort entirely; the abort is a gentle suggestion mechanism. (3) *broadcast*: a BROADCAST on a condition causes all the processes waiting on the condition to resume.

Communication with input/output devices is handled by monitors and condition variables. When a device needs attention, it NOTIFYs a condition variable to wake up the waiting handler process; since the device does not actually acquire the monitor lock, the NOTIFY is called a *naked NOTIFY*. Wakeup-waiting switches are provided in certain condition variables (making them binary semaphores) to ensure device NOTIFYs are not lost due to timing races.

The authors discuss the implementation, describing the processor and its data structures, the run time support package, the compiler, and the performance of the Mesa concurrency facilities. As well, three applications and the Mesa problems discovered through the applications are described: (1) Pilot, a general purpose operating system to run a large, personal computer; (2) Violet, distributed calendar system; and (3) Gateway, an internetwork forwarder for packet networks.

By discussing some general issues in the design and use of monitors and processes, the authors provide insight into the reasons for decisions taken in the Mesa language design. There are few examples, but the paper does not suffer much from this. It does become somewhat obscure when discussing the Mesa divergence from Hoare's monitors, process priorities, and non-module monitors. The design of the Mesa concurrency facilities incorporates some unique features which are worth examining.

[Hoare 1974] Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." *Comm. ACM* **17**, 10 (October 1974), pp. 549-557.

[Holt 1983] Holt, R. C. *Concurrent Euclid, Unix, and Tunis*. Toronto: Addison-Wesley, 1983.