

Experience with Processes and Monitors in Mesa

Authors: Butler W. Lampson, David D. Redell

Presented by: John McCall

`rjmccall@cs.pdx.edu`

Star and Pilot

The Xerox Star (1981) was a personal workstation designed for office use; it was a fairly visionary device, with many of the features (a graphical display, a mouse, icons, overlapping windows, ethernet) of a modern PC.

The Star's operating system was Pilot:

- pre-emptive multitasking (but not timeslicing)
- virtual memory (but all memory was shared unprotected, including Pilot's)

Mesa

Pilot and the entire Star application suite were written in Mesa, a high-level (hence the name) programming language written specifically for this purpose. Mesa was a successor of Modula and ALGOL and a predecessor of Modula-2 and -3; DoD originally considered Mesa for its institutional PL, but Xerox refused, and so DoD created Ada.

Mesa is a procedural language with rich support for exceptions, modularity, and concurrency; balancing these is a key point of Mesa's design.

Concurrency in Mesa

Processes in Mesa are *lightweight*. All processes share the same memory space, and a process has no explicit stack; instead, a process simply allocates stack frames on the heap.

Processes can safely share data and wait for events using monitors and condition variables. These are slightly different from how they were presented in Hoare.

Alternatives to Monitors

The Mesa designers considered some alternatives to monitors:

- They could use *message-passing*, but it's harder to support in the language, and they were already committed to using procedures.
- They could rely on *non-preemption*, but:
 - they didn't want to rely on having a uniprocessor architecture
 - programmers wouldn't be able to call functions which just coincidentally yield
 - virtual memory and I/O can cause preemptions anyway

Monitors and Modularity

A module in Mesa is a collection of private variables and private procedures (which can't be seen outside the module) and public procedures (which can be called by anyone).

A monitor module is a special kind of module which can also have public *entry procedures* which acquire the monitor's lock on entry and release it on exit.

Mesa also allows monitors to be dynamically created, which isn't possible in Hoare.

Condition Variables, Part 1

- Hoare's semantics: if P_1 is waiting on V , then if P_2 signals V , the system must immediately resume P_1 .
- Mesa's semantics: P_1 resumes whenever it's convenient for the system, potentially even if V hasn't been signalled.

These semantics are much weaker! Hoare's semantics allow for some precise control over scheduling which can't be achieved anymore. Furthermore, Mesa's semantics require the actual condition to be checked after a wakeup; Hoare's semantics provide a guarantee of correctness. Worse, Mesa's semantics allow for starvation: by the time P_1 resumes, some other process might have already reacquired the lock!

Condition Variables, Part 2

It turns out that Mesa's semantics are usually better:

- Hoare's semantics mandate a context-switch which usually isn't required.
- Hoare's semantics make some very stringent demands of schedulers.
- Therefore Mesa's semantics are usually cheaper and easier to implement.
- Anyway, "the low level scheduling mechanism provided by monitor locks should not be used to implement high level scheduling decisions".
- Hoare's semantics encourage programmers to only signal a variable if they're absolutely sure the waiter will be ready to run afterwards; Mesa's semantics make spurious signals much cheaper.

Condition Variables, Part 3

Mesa's semantics also encourage some easy extensions to the condition-variable API.

- `signal()` is now called `notify()`.
- `broadcast()` wakes up every thread that's waiting on the condition.
- A call to `wait()` can time-out, which is frequently useful in practical programs.
- `notify()` can be safely called without holding the lock if the condition variable is turned into a binary semaphore (this eliminates a potential race condition).

Queues

Scheduling in Mesa/Pilot/Star is implemented using a simple, hardware-supported round-robin scheduler. A process is always in one of four types of queues:

- The processor's job is to always run the first job in the system-wide *ready queue*.
- Monitor locks have associated *monitor lock queues*.
- Condition variables have associated *condition variable queues*.
- Faults cause processes to move into a *fault queue*.

When a process is notified, it is moved to the tail of the ready queue. The processor occasionally scans the full process table and notifies any process whose timeout has expired.

Priorities, Part 1

It can be useful in many applications to suggest to the scheduler that some work is more important than others. In Mesa/Pilot, every process is assigned a priority, and a process is only scheduled if no higher-priority process is available; this is implemented by sorting the ready queue in reverse order of priority.

Priorities, Part 2

This approach suffers from a problem (now) called “priority inversion”. Let P_1 , P_2 , and P_3 be processes, where higher-numbered processes have higher priorities.

Suppose that P_2 and P_3 are waiting on something, and P_1 enters a monitor M . P_3 wakes up and tries to enter M , but is blocked. P_2 wakes up and runs for awhile, preventing P_1 from running and exiting its critical section, which would allow P_3 to resume; in effect, P_3 is blocked by P_2 .

This problem does not arise in all applications, and it can be addressed by temporarily raising the priority of any process entering a monitor to the maximum priority of any process which ever enters the monitor.

Deadlock

- Mesa monitors are non-recursive, so a monitor entry procedure can't call itself or any other entry procedure of that monitor. The compiler can check for this.
- If monitor M_1 calls monitor M_2 , and M_2 calls M_1 , there's a possibility for deadlock. This can be solved by always entering monitors in a particular order — i.e. there should never be any mutually-recursive monitors.
- If P_1 is inside monitors M_1 and M_2 , and P_1 waits on a condition variable V inside M_2 , only M_2 is released; P_2 might want to signal V , but might instead be blocked because it needs to enter M_1 first.
- Every process can be waiting on a condition variable, each expecting one of the others to wake it up.