Why this paper?

- Nested monitor calls
- Handling of time-outs, aborts and exceptional conditions
- Meaning of WAIT and process scheduling
- Interactions with process creation and destruction
- Monitoring a dynamic number of objects

Motivation of the Work

- Pilot, written in Mesa, supporting a large range of applications
  - Local concurrent programming, global resource sharing, and interrupts
- Decided on shared-memory paradigm
- Decided against non-preemptive scheduling and semaphores
- Problems with monitors

Process Creation in Mesa

- A special procedure activation

```
p <- FORK  readline[terminal];
  ….
buffer <-  JOIN p;
```

- a process in this context means a thread
  - any procedure can be invoked as a thread
- a process (thread) is a first-class value
- a process value can be a dangling reference
- Exception handling

Monitors

Unifies

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

Ensures that at most one process is executing a monitor procedure at a time

- impose no constraint on the order of calling entry procedures

Monitor Modules

- procedures:
  - entry
  - internal
  - external
- “wait” and releasing locks:
  - what if wait is called inside an internal procedure?
  - what if wait is called in some other monitor’s procedure?
  - what about exceptions?

Deadlocks

- three patterns:
  - conditional variables only
  - cyclic calling among monitors
  - involving both monitors and conditional variables

An Example

```
Procedure Get() =
    enter monitor A {
        from monitor A enter monitor B {
                while (not ready) do wait( c ) ;
                ready = false;
        }
    }
Procedure Give () =
    enter monitor A {
        from monitor A enter monitor B {
                ready = true; signal ( c );
        }
}
```

Monitoring Dynamic Objects

- Monitoring many similar objects:
  - one monitor
  - multiple copies of a monitor module
  - monitored object
    - they are really using semaphores
- Abandoning computations upon exception:
  - P1->P2->…->Pn: Pn, …, P2 need to restore the invariant through unwind before Mesa releases their locks

Conditional Variables

- wait/signal in Mesa:
  - signal wakes up all waiting processes
  - pattern of code for waiting:
    while (condition not satisfied) { wait c;}
- Implication on proof rules
- Variants:
  - time out
  - one process sending abort signal to another process

Dealing with Input and Output

- What about I/O devices
  - How to encapsulate I/O device interrupts in the monitor paradigm
  - Solution: “naked” notify
  - Such solution is not always necessary...

Priority Inversion

- Scenario:
  - P1 enters M;
  - P1 preempted by P2 (P2’s priority > P1’s)
  - P2 preempted by P3 (P3’s priority > P2’s)
  - P3 tries to lock M, blocks, gives CPU to P2
  - P2 continue its very long computation…
- Solutions

Implementation

- Implementation
  - machine
  - run-time system (built as a monitor module on top of the machine layer)
  - compiler

Processor Layer

- “frames”: simple, small context for threads
  - as a consequence, cheap creation of threads
- queues of threads
  - monitor lock queues
  - conditional variable queues
  - ready queue
  - fault queue
- scheduling: priority and FIFO within a priority
- thread switching mechanism