Lecture 15: Monitors and Processes in Mesa

Mothy Roscoe and Joe Hellerstein

November 3, 2005

Focus of this paper: building a complex system using light-weight processes (threads in today's terminology) and how they communicate with each other using shared-memory.

Message: You have to build it to understand it.

Influence: Java (via Cedar, Modula 2+, etc.)

Background

- Pilot was the operating system for the Xerox Dandelion computer, sold as the Star 8000 series.
- Occupies mid-point between Alto/Alto-II (programmed in BCPL), and Dorado (using Cedar - Mesa with garbage collection and runtime types)
- Planned was to build a large system using many programmers (business unit partnership with PARC)
- Like all PARC computers from the Alto on, the network and network applications were central $\Rightarrow$ heavy users of concurrency.

Chose to build a single address space system:

- Single user system, so protection not an issue. (Safety was to come from the language.)
- Wanted global resource sharing.
- Mesa language: module-based programming with information hiding.

Since they were starting from scratch, could integrate the hardware, the runtime software, and the language with each other. This was not unusual at the time (and let to a great deal of innovation possibilities, not generally available today in the PC space).

Programming model for inter-process communication: shared memory (monitors) vs. message passing.

- Needham & Lauer claimed the two models are duals of each other.
- Chose shared memory model because they thought they could fit it into Mesa as a language construct more naturally.

How to synchronize processes?

- Non-preemptive scheduler: tends to yield very delicate systems. Why?
  - Have to know whether or not a yield might be called for every procedure you call. Violates information hiding.
  - Prohibits multiprocessor systems.
  - Need a separate preemptive mechanism for I/O anyway.
  - Can't do multiprogramming across page faults.

  Though: there are cases where this argument doesn't hold; where?
- Simple locking (e.g. semaphores): too little structuring discipline, e.g. no guarantee that locks will be released on every code path; wanted something that could be integrated into a Mesa language construct.
- Why not transactions? Aren't these the data-centric concurrency model of choice? Speculative answer:
  - Personal workstations aren't like mainframes, and the typical application demands, window system, email, document editing (Bravo), etc. aren't data-centric but more process-centric.
- Chose preemptive scheduling of light-weight processes and monitors.

Light-weight processes

- easy forking and synchronization
- shared address space
- fast performance for creation, switching, and synchronization; low storage overhead.

Monitors:

- monitor lock (for synchronization)
  - tied to module structure of the language: makes it clear what's being monitored.
  - language automatically acquires and releases the lock.
  - tied to a particular invariant, which helps users think about the program
  - condition variable (for scheduling)
  - Dangling references similar to pointers. There are also language-based solutions that would prohibit these kinds of errors, such as do-across, and more recently the pi-calculus and join-calculus, that eliminate dangling processes because the syntax defines the point of the fork and the join.

Changes made to design and implementation issues encountered

- 3 types of procedures in a monitor module:
  - entry (acquires and releases lock).
  - internal (no locking done): can't be called from outside the module (example of use of module hiding for correctness)
  - external (no locking done): externally callable. Why is this useful?
    - allows grouping of related things into a module.
    - allows doing some of the work outside the monitor lock.
    - allows controlled release and reacquisition of monitor lock.

Options for notify semantics:

- Could cede lock to waking process (directed yield): too many context switches. Q. Why would this approach be desirable? A. Waiting process knows the condition it was waiting on is guaranteed to hold.
- Alternatively, notifier keeps lock, waking process gets put in front of monitor queue. Garantees that the condition holds when the waiting process wakes up, but doesn't work in the presence of priorities.
- Adopted: Notifier keeps lock, wakes process with *no* guarantees $\Rightarrow$ waking process must recheck its condition. Leads to "while" idiom now familiar.

Example from the paper:

```
Allocate: ENTRY PROCEDURE [ size: INTEGER
RETURNS [ p: POINTER ] = BEGIN
        UNTIL availableStorage ≥ size
              DO WAIT moreAvailable ENDLOOP;
        p ← < remove chunk of size words & update availableStorage >
        end;

Free: ENTRY PROCEDURE [ p: POINTER, Size: INTEGER] = BEGIN
        < put back chunk of size words & update availableStorage >;
        NOTIFY moreAvailable END;
```

Q. Where is the bug here?

A. Broadcasts.

- You need BROADCAST.
- Single-waiter NOTIFY is a valuable optimization. C.f. pthreads, Java, etc.

Hints vs. Guarantees:

- Notify (or broadcast) is only a hint.
- $\Rightarrow$ don't have to wake up the right process, don't have to change the notifier if we slightly change the wait condition (the two are decoupled).
- $\Rightarrow$ easier to implement, because it's always OK to wake up too many processes. If we get lost, we could even wake up everybody (broadcast)

- Enables timeouts and aborts
- General Principle: use hints for performance that have little or better yet no effect on the correctness.

Q. Having done this, what other kinds of notification does this approach enable?
A. Timeouts, aborts.

Other issues they found:

- Deadlocks: Wait only releases the lock of the current monitor, not any nested calling monitors. This is a general problem with modular systems and synchronization: synchronization requires global knowledge about locks, which violates the information hiding paradigm of modular programming. Why is monitor deadlock less onerous than the yield problem for non-preemptive schedulers?
  - Want to generally insert as many yields as possible to provide increased concurrency; only use locks when you want to synchronize.
  - Yield bugs are difficult to find (symptoms may appear far after the bogus yield)
- Basic deadlock rule: no recursion, direct or mutual
- Lock granularity: introduced monitored records so that the same monitor code could handle multiple instances of something in parallel.
  - You can see how an object-oriented approach here would help ...
- Interrupts: interrupt handler can't afford to wait to acquire a monitor lock.
  - Introduced naked notifies: notifies done without holding the monitor lock.
  - Had to worry about a timing race: the notify could occur between a monitor's condition check and its call on Wait. Added a wakeup-waiting flag to condition variables.
  - What happens with active messages that need to acquire a lock? (move handler to its own thread)
  - Priority Inversion: high-priority processes may block on lower-priority processes

        * a solution: temporarily increase the priority of the holder of the monitor to that of the highest priority blocked process (somewhat tricky – what happens when that high-priority process finishes with the monitor? You have to know the priority of the next highest $\Rightarrow$ keep them sorted or scan the list on exit)

    - Exceptions: must restore monitor invariant as you unwind the stack. What does Java do? (I don't know, but it's probably broken.)

Performance

- Context switch is very fast: 2 procedure calls. Those were the days ...
- Ended up not mattering much for performance: ran only on uniprocessor systems, and concurrency mostly used for clean structuring purposes.
- Procedure calls are slow: 30 instrs (RISC proc. calls are 10x faster). Due to heap-allocated procedure frames. Why did they do this?
    - Didn' want to worry about colliding process stacks.
    - Mental model was any procedure call might be a fork: xfer was basic control transfer primitive.
    - Process creation: 1100 instrs.
    - Good enough most of the time.
    - Fast-fork package implemented later that keeps around a pool of "available" processes (many subsequent thread systems do this)
    - Note that these days, CPU benchmarks discourage fast context switch implementations.

Summary

3 key features about the paper:

- Describes the experiences designers had with designing, building and using a large system that aggressively relies on light-weight processes and monitor facilities for all its software concurrency needs.

- Describes various subtle issues of implementing a threads-with-monitors design in real life for a large system.
- Discusses the performance and overheads of various primitives and three representative applications, but doesn't give a big picture of how important various things turned out to be.

Some flaws:

- Gloss over how hard it is to program with locks and exceptions sometimes. (Not clear if there are better ways, though).
- Performance discussion doesn't give the big picture.

Aside: where did this go?

- Cedar: Mesa with garbage collection and runtime-types
- Modula 2+: the same people but rebuilt at SRC after "the great Exodus"
- SRC Threads: abandoned monitors in favour of Mutexes (why?)
- Modula-3
- Java: Return of monitors

A lesson: The light-weight threads-with-monitors programming paradigm can be used to successfully build large systems, but there are subtle points that have to be correct in the design and implementation in order to do so.

A further lesson: The theory (which had existed for some years) turned out to be only half the story.

> In designing an operating system one needs both theoretical insight and horse sense. Without the former, one designs an ad hoc mess; without the latter one designs an elephant in best Carrara marble (white, perfect, and immobile) *R. M. Needham and D. F. Hartley*.