Advanced Topics in Computer Systems, CS262A

Prof. Eric Brewer

Experience With Processes and Monitors in Mesa

I. Experience With Processes and Monitors in Mesa

Focus of this paper: light-weight processes (threads in today's terminology) and how they synchronize with each other.

History:

- 2nd system; followed the Alto.
- planned to build a large system using many programmers. (Some thoughts about commercializing.)
- advent of things like server machines and networking introduced applications that are 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.

Large system, many programmers, many applications:

- Module-based programming with information hiding.

Since they were starting "from scratch", they could integrate the hardware, the runtime software, and the language with each other.

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.
- 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.

Lecture 19

- 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 those of pointers. There are also language-based solutions that would prohibit these kinds of errors, such as do-across, which is just a parallel control structure. It eliminates dangling processes because the syntax defines the point of the fork and the join.
- Monitors (and Mesa in particular) led to several aspects of Java. Java's synchronized

Monitors	Java Synchronized Objects
external	public
internal	private synchronized
entry	public synchronized

objects are the object-oriented programming version on monitors, and they are a better solution than monitored records. (Each instance has its own lock rather than just each element of an array.)

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.
  - 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.

- Notify semantics:
  - Cede lock to waking process: too many context switches. Why would this approach be desirable? (Waiting process knows the condition it was waiting on is guaranteed to hold.)

- Notifier keeps lock, waking process gets put a front of monitor queue. Doesn't work in the presence of priorities.
- Notifier keeps lock, wakes process with no guarantees => waking process must recheck its condition.

What other kinds of notification does this approach enable?

Timeouts, broadcasts, aborts.

- Abort: a nice request to abort -- allows the target process to reach a wait or monitor exit, and then it voluntarily aborts. No need to re-establish the invariant -- as compared to just killing the process outright!
- 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.
- Interrupts: interrupt handler can't block waiting 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. This a "time of check to time of use" (toctou, pronounced "tock-too") bug -- the condition of the test no longer applies at the time of use. In particular, the sequence: 1) check for waiters == false, 2) naked notify, 3) go to sleep is a toctou bug. Added a *wakeup-waiting* flag to condition variables; naked notify sets the wakeup waiting flag if the target is awake, and before that process goes to sleep it checks this bit, and if set, resets it, and stays awake to check for notifies again.
  - What happens in general with a message handlers that needs to acquire a lock? (use the interrupt to queue the task, and then acquire locks in a process context instead)
- 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 ⇒ keep them sorted or scan the list on exit)
  - The Mars rover stalled due to this kind of bug and had to be debugged and fixed from earth!
- Exceptions: must restore monitor invariant as you unwind the stack. What does Java do? (you must use a sequence of try-finally blocks)
  - The idea that you can just kill a process and release the locks is naive -- each lock protects some invariant that really needs to be restored before you can release the lock.
  - Entry procedures that have an exception, but no exception handler DO NOT release the monitor lock. This ensures deadlock and a trip into the debugger, but at least it maintains the invariant.

Lecture 19

Hints vs. Guarantees:

- Notify is only a hint.
- ⇒ 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).
- ⇒ 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. Inktomi uses hints for fault tolerance: if the hint is wrong, things we'll timeout and we'll use a backup strategy ⇒ performance hit for incorrect hint, but no errors.

Performance:

- Context switch is very fast: 2 procedure calls.
  - Ended up not mattering much for performance:

  - ran only on uniprocessor systems.

  - 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't 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 or "available" processes.

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).
- Performance discussion doesn't give the big picture.

Lecture 19

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.