

Experience with Processes and Monitors in Mesa

Arvind Krishnamurthy

Background

- Focus of this paper: light-weight processes (threads) and how they synchronize with each other
- History:
 - Xerox Alto: first personal computer
 - Pilot is the OS for its successor (Xerox Star)
 - Advent of things like server machines and networking introduced applications that are highly concurrent
 - Single user system
 - Safety was to come from language

Background

- 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

- Discuss:
 - what you liked about the paper?
 - what you disliked?
 - what did not make sense or what was not clear?

Programming model

- Two choices for programming concurrency:
 - Shared memory
 - Message passing
- What are their strengths/weaknesses?
- Needham & Lauer claimed the two models are duals
- Mesa uses shared memory model because it fits as a language construct more naturally

Synchronizing Processes

- Goal: mutual exclusion
- An option: non-preemptive scheduler
 - Process owns the processor till it yields
 - What are the downsides of using a non-preemptive scheduler?
- Another option: simple locking (e.g., semaphores)
 - How does it compare to monitors?

Mesa Language Constructs

- Light weight processes
- Monitors
- Condition variables

Light weight Processes

- Easy forking and synchronization
- Shared address space
- Fast performance for creation, switching, and synchronization
 - Low storage overheads
- Mesa is a single user system; what would change if it were to be used in a multi-user system?
- Dangling references similar to those of pointers
 - How can you prevent these dangling references?

Monitors

- Monitor lock for synchronization
 - Tied to module structure of the language; makes it clear what is being monitored
 - Language automatically acquires and releases the lock
- Tied to a particular invariant, which helps users think about the program

Modules and Monitors

- Three 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
- 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

Condition Variables

- Notify semantics options:
 - Cede lock to waking process
 - Notifier keeps lock, waking process gets put in front of monitor queue
 - Notifier keeps lock, wakes process with no guarantees
- What are the strengths/weaknesses of the different options?

Notification in Mesa

- It is a “hint”. Notifying process keeps the lock/control
- Other related aspects of notify:
 - Timeouts
 - Broadcasts: why is this useful?
 - Aborts:
 - 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

Announcements

- Make up class this friday?
- Use discussion board to discuss technical issues

Deadlocks

- Typical deadlock scenarios:
 - Recursion on the same module
 - Enter multiple monitors in different orders
 - Process 1 obtains monitor A followed by B; Process 2 obtains monitor B followed by A
 - Enter multiple monitors in the same order, but wait inside the second monitor does not release the lock of the first monitor
- General problem with modular systems and synchronization
 - Synchronization requires global knowledge about locks, which violates the information hiding paradigm

Other Issues

- Is monitor deadlock as onerous as the yield problem for non-preemptive schedulers?
- 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
 - Introduced naked notifies: notifies done without holding the monitor lock
 - What is the problem with naked notifies?
 - How can this be addressed?

Priority, locks, scheduling

- There are subtle interactions between priorities and scheduling and holding locks
- Mars Pathfinder:
 - Success story for the first few days
 - Landed with fancy airbags, released a “rover”, shot some spectacular photos of the Mars landscape
 - Few days later after it started collecting meteorological data, system started resetting itself periodically

Priority Inversion

- “Information bus” is a shared memory region shared across the following processes:
 - Bus manager (high priority process)
 - Meteorological data gatherer (low priority)
 - Reset if Bus Manager hasn't run for a while
 - Protected by a lock
 - If Bus Manager is scheduled by context-switching out the data gatherer, it will sleep for a bit, let the data gatherer run, which will release the lock in a short while

Priority Inversion

- Another thread: communications task
 - Medium priority, long running task
 - Sometimes the communications task would get scheduled instead of the data gatherer
 - Neither the lower priority data gatherer nor the higher priority bus manager would run
- Works in pairs, but not all three together. Resulted in periodic resets
- How do we fix this problem?

Other Issues

- Exceptions:
 - Must restore monitor invariant as you unwind the stack
 - The idea that you just kill a process and release the locks is naive
 - 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

Performance

- Context switch is very fast
 - Two procedure calls
 - Ended up not mattering much!
 - Ran only on uniprocessor systems
 - Concurrency mostly used for clean structuring purposes
- Procedure calls: 30 instructions
 - Is this a reasonable number?
- Process creation is about 1100 instructions
 - Good enough; “fast fork” implemented later keeps around a pool of available processes

Key Features of the Paper

- Describes the experiences designers had with designing, building, and using a large system that relies on lightweight processes
- Describes various subtle issues of implementing monitors
- Discusses the performance and overheads of various primitives

Discussion

- What about distributed memory systems or clusters?
What is a good programming model for concurrency in such systems?
- What other issues come up for multi-core systems?
Is the Mesa model appropriate for multi-cores?
- What are the key differences between Mesa and its modern counterparts?