Threads programming

CS380L: Mike Dahlin

September 30, 2002

> But, the main reason for advocating the use of this pattern is to make your program more obiously, and more robustly, correct...[T]his programming convention allows you to verify correctness by local inspection, which is always preferable to global inspection."
>
> *Andrew Birrell "An introduction to programming with threads"*

> One final warning: don't emphasize efficiency at the expense of correctness. It is much easier to start with a correct program and ...[make] it efficient, than to start with an efficient program and ...[make] it correct.
>
> *Andrew Birrell "An introduction to programming with threads"*

1 Preliminaries

1.1 Review

1.2 Outline

1.3 Preview

2 Rules to live by

Mostly from Lamspon and Redell "Experience with processors and monitors in Mesa" CACM v23 n2 1980.

2.0.1 History

- Mesa: programming language for Pilot personal computer at Xerox PARC
  - Personal computer for PARC Computer scientists: write software in one language (Mesa), each user has own machine → protection is to guard against bugs not malicious attacks; resource mgmt is minor concern

- Extreme research: light weight threads
  - "Second system"
    - First system was Alto (pre-PC PC)
    - Second system: support muti-programming
  - Debate of concurrent programming model
    - Message passing, or
    - Shared memory
    - Debate settled on the ground of ease of programming
  - Debate of synchronization primitives
    - Non-preemptive scheduler, or
    - Semaphores, or
    - Monitors (locks + condition variables)
      - Hoare semantics, or
      - Hansen semantics
    - Debate settled on the ground of structures
  - Terminology and modern languages
    - Lampson/Redell "process" = modern "thread"
      - Mesa: no HW protection – process == thread
    - LR "Monitor" v. "Monitored record"
      - LR Default "monitor" associates synchronization variables with code (e.g., class variables/static variables)
      - LR "Monitored record" associates synchronization variables with object (e.g., member variables)
      - Modern: Member variables are default → when I say "monitor" I mean "montored record"
    - Monitor the language construct v. the programming construct
      - My definition: monitor = programming style that uses locks + condition varaibles as synchronization primitive
      - Some programming languages (Mesa, Java) add syntactic sugar to automatically grab locks at start of procedure, release at end
      - Some programming languages (C++, C) require programmer to follow coding conventions and manually acquire/release locks
      - Both use "monitors" in my terminology (not everyone would agree.)

2.0.2 Invariants are key

- Sequential object-oriented programming
    * Each object has invariants assumed on public method entry and established before method exit
- Monitor: Establish invariants
    * Object initialization
    * Before return from public method
    * Before wait
    * Before throw (or pass through) exception from public method
- Monitor: Safe to assume invariants
    * Public entry
    * Return from wait
        · Note: do not assume *more* than invariant on return!
- Signal is *hint*
    * *Not* safe to assume anything stronger than invariants on return from wait → must check condition
        · **Always:** `while(...){wait(&lock);}`
    * Hansen semantics are *much better* than Hoare semantics
        1. Modularity: LR: "*Verification is made simpler and more localized.*" Look at code around the `wait` to see when the thread will proceed v. look at all code that can call `signal`
            · NB: Textbooks often assert that Hoare variables are preferable for "proving correctness" of programs.
            · This is bunk (IMHO). Much easier to prove safety under Hansen semantics. Placement of signals/broadcasts irrelevant to safety.
            · Hoare semantics convenient for proving liveness/progress/fairness.
        2. Simpler signalling: LR "many applications do not trouble to determine whether the exact condition needed by a waiter has been established. Instead, they choose a very cheap [MDD: or simple] predicate which implies the exact condition...and NOTIFY a *covering* condition variable."
        3. Simpler run-time support
            · Fewer constraints on scheduler
            · Supports timeout, abort, broadcast
        4. Portability: Are correct Hansen programs still guaranteed correct under Hoare threads library? Vice versa?

            · Hansen program need not re-establish invariants before signalling, but Hoare threads library schedules waiter immediatly after signal. Correct Hansen programs can break under Hoare.
            · Hoare programs can assume something stronger than monitor invariants when rescheduled after wait. Correct Hoare programs can break under Hansen.
            · Practical answer: essentially all real systems are Hansen semantics. *Always use while(...)wait*
            · Not a bad idea to always signal/broadcast at the end of a method just before releasing lock (e.g., after re-establishing invariants.)
            · If you do both of these things, your program will always work regardless of underlying semantics.

        Code that assumes Hansen is correct (safe) under Hoare but not vice versa.
            · Note: Hoare must re-establish invariants before signalling
        · What about performance?
        · Hansen "while": Extra check of condition (usually cheap)
        · Hoare: an extra context switch if signaller has more work to do before leaving monitor

2.0.3 Other issues

Deadlock

- Biggest problem for monitors
    * Deadlock breaks modularity
    * Fundamental problem. No known general solution. Instead: coding standards, defensive programming, etc.
- Pairwise deadlock case 1: Mutual waiting within monitor
    * LR: "Localized bug in the monitor code ... usually easy to locate and correct
- Pairwise deadlock case 2: Lock cycle across monitors
    * Simplest solution: partial ordering across resources (avoid mutually recursive monitors)
    * *What about callbacks?*
- Pairwise deadlock case 3: Nested monitors + wait
    * Solution: "Break [monitor] M into two parts: a monitor M' and an ordinary module O which implements the abstraction defined by M, and calls M' for access to shared data. The call on [nested monitor] N must now be done from O rather than from within M'."

- Solutions to cases 2 and 3 break modularity and are not general
  * They require knowledge of internals of other modules: *Can this module call me? Can this method wait?*
    · Target of call: no lock → OK. Caller can continue to hold lock
    · Target of call: locks but never waits → caller can continue to hold lock if partial ordering exists (e.g., callee never calls back to caller or higher)
    · Target of call: locks and may wait → dangerous to call while holding a lock
- Proposed rule: Manually release lock when calling another module
  * Why not automatically release lock on any call out?
    · Unacceptable semantic burden for reasoning about procedure calls – need to establish invariant before each call!
  * Manual requires lots of careful thought
    · Birrell: "You should generally avoid holding a mutex while making an up-call (but this is easier said than done)."
  * Still follow rule: release lock only at beginning/end of procedure
    · → wrapper procedures?
    · → continuation style of programming?
    · Be careful not to assume anything stronger than invariant upon re-entry (danger: "implicit" reasoning based on "program counter").
  * Exceptions to rule
    · **Callee uses no locks OR Callee uses no condition variables and partial order exists**
    · Inner classes only?
    · Manually verify and hope invariant continues to hold?
    · Syntactic sugar? (e.g., similar to `const`)
    · Static/run-time verficiation?
    · Project idea: Extend Lin and Guyers (or Engler's) static checker to detect possible deadlocks where "no locks" rule is violated? (I think the partial order rule has already been done)
    · **Other exeptions? (When is it safe to call a method that might wait?)**
  * Consider major restructuring
    · Birrell: Upcalls for message reception (e.g., link → IP → tcp) can be made into top-down reads of data
    · Birrell: For message processing this gives "uniformly bad performance" → use upcalls instead
    · Dahlin: Beware premature optimization. You give up a lot of simplicity when you use upcalls; make sure it is worth the performance (measure it!)

Priority inversion

- Example
  * Threads 1, 2, 3 have low, medium, and high priority
  * Thread 1 acquires lock
  * Thread 3 waits for lock
  * Thread 2 chews up CPU
  * → Thread 3 (high priority) is waiting for thread 2 (low priority)
  * Example: Mars pathfinder bug (and many others)
- LR propose 3 solutions
  1. Don't worry about it
  2. Thread holding lock is given highest priority of any thread waiting for lock
  3. Thread holding lock is given highest priority in system
- Pros/cons in Mesa? In more general threads system?

I/O, interrupts

- Hardware = globally shared data structure
  * Concurrent access: hardware events + one or more software threads (to control, poll, etc the hardware)
  * Hardware cannot be forced to acquire monitor lock before accessing state
- Solution
  * Software access via monitor (to limit access by multiple software threads)
  * Software/HW synchronization is ad-hoc (typically uses atomic load/store to hardware registers; hardware defines semantics)
  * Mesa: *Naked notify* Software thread wants to wait for hardware event
    · Problem: Event can happen between software "while()" and software "wait()" → software waits forever
    · Solution: add state to condition variable (b̃inary semaphore)
  * General event solution: (first try)
    · LR: "We briefly considered a design in which devices would wait on and acquire the monitor lock, exactly like ordinary Mesa processes"
    · LR: "This design is attractive because it avoids both .. anomolies just discussed"
    · LR: "The worst case response time of the faster process therefore cannot be less than the time the slower one needs to finish its critical section....We consider this a fundamental deficiency"

- **Interrupt based world... (LR can consider modifying all hardware, we can't)**
- Interrupt t̃hread – have it enter monitor and deposit new data
- *Problem: Do not acquire lock when servicing interrupt (roughly for reasons above; also b/c timer interrupt often turned off..)*
- Have interrupt fork a new thread whose job it is to deposit work
- *Overhead*
- *Lose FIFO order*
- *Same "fundamental problem": LR: "Although one can get higher throughput from the faster processor than the slower one, one cannot get better worst-case real-time performance."*
- *How did we pass work to thread in thread pool? (Same problem, we've just pushed it down a level...)*

* General event solution: Queue of events
* Producer/consumer queue of events
* HW synchronization between producer and consumer via semaphore
  - HW/SW boundary
* History: Dijkstra invented semaphores to solve the I/O problem! No surprise they are a good match.
* Recent history: Many high-performance I/O systems use queue of pending events as primary abstraction – Exokernel interrupts, Active Messages, TRIPS ... (think about this as we read the Active Messages paper in a few weeks)

High-performance implementation Lots of work to make 'extreme concurrency" work. I won't cover all points. 2 points worth noting.

1. Memory overhead
   - C/C++ stack, stack frame allocation
     - Stack frame allocation: bump stack pointer
     - Single thread: Fault (and grow stack) if stack crosses page boundary to unallocated memory
     - Multi thread option 1: allocate each stack on separate page; use unallocated pages between to detect and deal with growth → **space overhead 1 page per thread**
       * OS/2 anecdote
     - Multi thread option 2: allocate stacks with "enough" room between them. Hope they don't grow too big. → **stack smash bugs**

       * Stack smash bugs also an option for single-threaded programs in C/C++ world...
   - Mesa stack, stack frame allocation
     - Requirement: type safety – no stack smashing allowed
     - Requirement: cheap threads – don't allocate excess stack space per thread
     - Solution: Allocate call stacks as linked structures in heap.
       * Mesa: 10 bytes overhead per thread
2. Context switch overhead
   - Goal: want to make context switches as fast as procedure calls
   - Two ways of accomplishing this:
     - Make context switches fast
     - Make procedure calls slow
     - They did both in Mesa
   - Context switch: 2x procedure call
   - Procedure calls: 30 instructions, 10x RISC cost
   - (Typical) Unix context switch: 1000-2000 instructions (100-1000x)
   - Reason
     - Heap-allocated procedure frames
     - (Implementation artifact or fundamental to making threads cheap?)

2.1 Basic threads programming

Dahlin "Basic Threads Programming: Standards and Strategy" CS372 (Undergrad OS) handout

- These rules keep you from making most common mistakes
- Fall 2001 midterm/final exam:
  - Every incorrect program violated at least one rule
  - 90% of programs that violated at least one rule were semantically incorrect
    * All that violate one rule are *wrong* – they are harder to read, understand, maintain.
- Depart from these rules only with careful consideration and only after discussion of alternatives with at least 2 other good programmers

3 Admin

4 Other views

Note: these "other views" would probably agree with the vast majority of what is said above. One of them appears to be *slightly* more liberal about programming rules. One of them appears to be *slighlty* more conservative about when threads should be used at all. I believe Birrell or Ousterhout would agree that if you use threads, you should follow the simple rules above the vast majority of the time and unless you have a very good reason.

4.1 Advance threads programming

Andrew Birrell "An introduction to programming with threads" Digital SRC Tech report, Jan 1989.

- Same basic message as Lampson/Redell (and me)
  - Not too surprising...
- More tolerant than I am about exceptions to the rules
  - He "discourages" rather than "forbids"
    - Birrell "strongly discourage doing X, but if you do X make sure you consider Y and Z..."
    - Dahlin "Don't do X. If you really think you need to do X, first think about how to restructure your code. If you still think you need to do X, talk to at least 2 other competent engineers first and/or code it the other way and measure it first."
  - "Double checked locking pattern for lazy initialization"
    - e.g.

```
IF NOT initDone THEN
   LOCK m DO
     IF NOT initDone THEN
       Initialize();
       initDone := TRUE
     END
   END
END
```

    - Broken. Don't do this.
    - Reason 1: assumes a lot about the memory model, allowed compiler optimizations, etc.
      - See Bacon, et al 'The "Double-Checked Locking is Broken" Declaration' http://www.cs.umd.edu
      - Memory model: processor may see initialized object but not see initialized fields within object
    - Reason 2: How long until some bonehead behind you deletes the second check as redundant?
    - Reason 3: Do things the same way every time
  - Releasing the mutex within a LOCK clause
    - Birrell strongly discourages "SRC discourages this paradigm, to reduce the tendency to introduce quite subtle bugs"
    - But if you still want to do this...Birrell "You must exercise extra care"
      - Operations must be correctly bracketed even in presence of exceptions
      - Object state may have changed while you had the mutex unlocked...your embedded control flow might depend decisions based on on previous state that is no longer valid
    - But if you still want to do this...Dahlin "Don't do this!"
      - Restructure code to acquire/release lock at beginning/end of procedures (e.g., wrapper procedure, explicit continuations)
      - $\rightarrow$ force you to think about re-establishing invariants, etc.
      - I predict: 9 times out of 10, in the process of re-writing you will end up with much cleaner code
  - Some useful tricks
    - Version stamps to support caching
    - Thread pools ("work crews")

4.2 Case against threads

John Ousterhout "Why Threads Are A Bad Idea (for most purposes)." This talk was presented as an Invited Talk at the 1996 USENIX Technical Conference (January 25, 1996).

The talk compares the threads style of programming to an alternative approach, events, that use only a single thread of control. Although each approach has its weaknesses, events result in simpler, more manageable code than threads, with efficiency that is generally as good as or better than threads. Most of the applications for which threading is currently recommended (including nearly all user-interface applications) would be better off with an event-based implementation.