Threads: A Case Study

15-712 #3

To understand systems, it is not enough to describe how things should be;

one also needs to know how they are.

- Hauser, Jacobi, Theimer, Welsh, and Weiser

"Using Threads in Interactive Systems: A Case Study"

Rare Observation & Experience Paper

- 10-20 years of advanced hands-on coding
- 2.5Mloc, 10K files, 1K monitors, 300 cond. vars
  - Asserted to be largest/longest lived thread-based interactive system
- Developed for Altos & Dorados (last time)
  - Data from Sparcstations using SunOS + Portable Common Runtime
- Threading while the rest of world was Process-based
  - MIMD shared memory, uniprocessor
  - i.e. threads were for structure, not parallel procs
- Cedar - original research system (less efficient?)
- GVX - product split from Cedar in '83

PARC & Mesa again

- Insert wow-ness about PARC again
  - Huge, successful, innovative systems research lab
- Mesa:
  - Native monitors (think "java synchronized methods")
  - Preemptive, strict priorities. 50ms scheduling quantum.
    - Weaker than Hoare (exactly one wakes up, immediately takes over monitor)
    - But notify gives strong perf - exactly one wakens

Design space

- Multi-tasking models: Who runs when?
  - Preemptive
  - Cooperative
- Implementation: where do threads exist?
  - Kernel
  - Userlevel
- Tradeoffs: Single-proc performance, efficiency, thread switch time, really non-blocking-ness, multiprocessor capable, processor details (affinity, IPIs, etc.)

Method

- Instrumented runtime to collect data
  - Good: Very powerful; lots of data from real code
  - Potential bad: Affects system behavior?
- Recorded
  - #threads
  - lifetime
  - run-length distribution
  - lock/wait rates
- Workload:
  - "representative" benchmarks: compile, format, view

Thread Types

- Eternal, Worker, Transient
- Eternal:
  - "Managers" wait on external events & trigger workers
  - Execution mostly short: < 5ms; but ~45ms dominate cycle use
  - Goal: Minimize latency to event processing
    - e.g., to provide good iterative perf
- Workers: forked to handle job or wait on job notification
- Cedar: 35-40 (-120 in real use); GVX: 22

Transient threads

- To handle specific task
- (Note: Some workers may be eternal; some may be transient. Not entirely orthogonal definition.)
  - Transients ran briefly
- How does this compare?
- My mac has 223 threads running w/Keynote/etc, 93 processes. Apps have 4-20 threads. Daemons 20+
- Their sys: 41 in benchmarks; 2-3x this in "real use"
  - Not too far off...

Synchronization

- Monitors: Protect specific bit of code (or specific data structure)
  - Kind of like "java synchronized"
  - Heavy use as mutexes on shared data
    - < 0.4% contention. (Remember: single proc!)
  - Waits mostly timeout (sleepers, polling, blinking, $ mgmt)
- Cond vars: wait for specific condition (more abstract notion - programmer can define)
  - Not surprisingly: more monitors than CVs
  - Example monitor: protect data structure in shared lib
  - Example CV: checking for items in a work queue
- Cedar uses 5-10x synchronization

Impl notes on synch

- Note that underlying operations must be atomic
  - Depends on single vs. multi-proc; memory ordering semantics; cache sharing semantics; etc.
  - Processors provide primitives such as atomic test/set to use to implement

Scheduling

- Nice tidbit:
  - "Most execution intervals are short; longer execution intervals account for most of the total execution time in our systems"
  - This trend shows up often (most web objects are small; large objects account for most data volume)
- Helps for scheduling -- still a modern area
  - e.g., old BSD priority: 1 / recent CPU use
  - BSD ULE: "interactivity score" (time run vs. time voluntarily slept)
  - Can often provide both good interactive performance *and* efficient resource use. Schedule interactive stuff first.

Thread use

- Birrell91 (intro to programming w/threads)
  - Exploit concurrency w/multiple CPUs
  - Processor sharing to make progress on multiple tasks
    - Network clients, multiple humans, etc.
  - Defer low-priority work while busy
- Hauser93: more patterns
  - Defer work, pumps, slac procs, sleepers & 1-shots, deadlock avoidance, task rejuv, serializers, concurrency exploiters, encapsulated forks
  - Tradeoff: forking takes cycles & mem => programming ease and parallelism

Defer work

- Get output back to user sooner (most important)
  - Email send, document print, window update, etc.
  - Enduring importance: ensure user interactivity!
  - Most common use of forking in Cedar
- Delay until less busy time
  - Priority of forked thread determines delay
  - Really, same thing: do important work first...

Pipelines, pumps, & slack

- Pipelines of multiple threads
  - Birrell91 intended pipelines for multiple CPUs
    - Amusingly, that era is just starting now
    - Software will take a while to catch up
    - (But look at # threads in CPU-heavy Apple programs...)
  - Hauser93 saw programmer convenience
    - "Pumps" as components of a pipeline
    - e.g., input filters
    - analog: cat foo | grep -v bar | gzip > foo.txt.gz
-

Slack procs: batching

- Coalesce work
  - Deliberately add latency
  - Batch for greater efficiency
  - Complex: Bound added latency w/efficiency...
  - Excess switches to higher-priority slack
    - YieldButNotToMe
    - Batching effect limited/forced by length of quanta
      - Could easily be too much or too little

Sleepers & Oneshots

- Sleepers "time" system behavior
  - Blink cursor in M ms, test input in T secs
  - Service work batched by a slack proc (polling)
    - Garbage collection, file state change callbacks, etc.
- Sleepers that exit: one-shots
  - Test that a condition persists long enough to be real (double click example)
- Errors: programming by "time"
  - Timeout values often wrong (how long to wait)
  - Machine quanta forces min. resolution
  - Mistakes create bad performance & response time (hard to debug...)

Deadlock Avoiders

- Best example: A lock hierarchy
  - Must hold locks A, B, and C to do operation
  - Grab locks A & B. Do some work. Then want C to do the rest.
    - Complex to keep all in mind and ensure no deadlocks
    - Must know all held locks at some call depth, release exactly the right set
    - This is a programming nightmare; source of many bugs
  - Simplify w/deadlock avoider:
    - Fork new thread that directly goes for A, B, and C
    - Unroll main proc all the way (release all locks)

Task Rejuvenation

- "Crash and Reboot" error handling
  - Today: "microreboot"; like a database abort and retry
  - May be a crash *response*, or could even do it preemptively as a "system cleaning" technique. :)
- Ask for fresh start, exit confused code
- But this raises serious design/religious issue...

Failure masking vs. fixing

- Robustness techniques can mask bugs (or make possible to blithely ignore)
  - These become performance problems
  - And perhaps lurking correctness problems -- there's buggy code running!
- What's more important?
- Pushing the idea hard: failure-oblivious code
  - Access invalid memory? Feed program junk data
  - ... it often keeps on running.
  - ... it often keeps on running *correctly*. Freakish, no?
  - Correctness, bug identification, or robustness? No clear answer - depends a lot on system.
  - Consider goals of a busy web server

Serializeers & Encap. forks

- Serializers
  - Single thread processing events from queue; queue filled by multiple threads.
  - This abstraction can *really* help simplify system
    - Allows the components to safely operate asynchronously
    - Stronger modularity between components
- Encapsulated forks
  - Library code that can be run sync or async
  - e.g., callbacks have code not understood by routine calling them, so explicitly indicate sync/async
  - Protect the server's thread of control

What use threads?

- Most common: defer work
- Sleepers (incl. queue watchers, timeouts, etc)
- General pumps
- Deadlock avoidance in Cedar

Priorities

- Hard to program!
  - Priority inversion is common
    - High prio thread waits on resource X
    - Low prio thread holds lock on resource X, but
    - Low prio thread can't run b/c of med prio thread CPU hog
    - mars pathfinder...
    - Suggest: trickling CPU to threads (breaking strict prio)
      - e.g. proportional fair share by prio
  - Again, a system robustness trade-off
    - Masking incorrect behavior!
    - Results in delays until locker thread *eventually* gets some CPU

Running out of Resources

- Mostly unrelated to threads. :)
- But very hard to deal with!
  - Memory failures are a *pain*.
  - Even in modern systems!
  - Many, many routines implicitly allocate memory
  - Forces programmers to really plan mem usage

Threads & Closures

- Closure: data structure holding all state needed to complete some work
  - i.e. buffer control block & I/O completion
  - Interrupt forks a "soft" interrupt handler w/pointer to buffer. Worker finishes I/O handling & wakes reader
  - Worker gets prior state from buffer header.
- Threads use stack state
  - 100KB in this paper. (Large - 32MB-64MB in $$ sys!)
  - Closures use "only enough" memory; more flexible
  - Threads visible to OS and debugger, often conceptually easier (debugging a closure-based system can *hurt*.)
  - Ex: 1000s of concurrent conns in web server

About this paper

- The good:
  - Loads of significant data; rare experience/introspection
  - Not enough empirical work in CS. Hard to evaluate abstractions, particularly programming abstractions.
    - And papers w/this much real data are *very* rare
- Hmm:
  - Not much comparison. Are these abstractions useful? Correct? The best? Why? Inter-system (Cedar/VVX) comparison?
  - Reader has to interpret most of the data.
    - This wasn't a "see, my idea wins" paper
  - What's good/bad/surprising in the #s?