Using Threads in Interactive Systems: A Case Study

Franklin Chen

January 28, 1999

The paper being discussed

Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, Mark Weiser, “Using Threads in Interactive Systems: A Case Study”, *14th ACM Symposium on Operating Systems Principles* (SOSP-14), December 1993.

Outline

- Introduction
- Thread model
- Classification of threads
  - Dynamic behavior
  - Paradigm of usage
- Issues discovered from experience
  - Using threads
  - Implementing threads
- Conclusions

Introduction

- Analyze large, real-life, thread-based interactive systems developed at Xerox
  - Cedar
  - GVX
- How large?
  - 2.5 million lines of code
  - 10,000 modules
    - More than 1000 monitors and monitored record types
    - Over 300 condition variables
  - Running: working sets on order of 10 MB

Goals

- Examine and classify common features and uses of threads in the systems

- Focus on threads for the purpose of program structuring, not multiprocessing

- Microscopic view: emphasize properties important for interactive usability

  - Example: update in window systems after user action

Thread model

- Mesa language
- Multiple, lightweight, pre-emptively scheduled threads in shared address space
- FORK, JOIN, DETACH
- Monitor (module, mutex)
- Finer grain: locks for data structures

Thread model (continued)

- Condition variables
  - Condition; WAIT, NOTIFY (exactly one waiter becomes runnable), BROADCAST (all waiters become runnable)
  - Contrast with Hoare model: in Mesa, no guarantee that the condition associated with a condition variable is satisfied when a WAIT completes. Thread must recheck. Common gotcha.
- Priorities
  - Scheduler runs highest priority runnable
  - If several runnable threads of highest priority, then round-robin
  - Runnable higher priority thread preempts currently running thread

Dynamic behavior of threads

Analyze statistics obtained from instrumented version of thread library, and running benchmark programs.

Patterns of usage fall into classes characterizable by thread lifetime.

- Eternal: run forever, waiting on condition variable
  - Example: garbage collection
- Worker: perform some computation
  - Example: document formatting
- Transient: thread with short life, forked off by long-lived thread
  - Example: results of keyboard activity

Dynamic behavior: number of threads

- Cedar
  - Idle system has about 35 eternal threads, forks transient 1/sec
  - Highest fork rate in benchmarks: 5/sec (keyboard input)
  - Maximum number of threads existing concurrently: 41
- GVX
  - Idle system has 22 eternal threads, forks no additional threads
  - Different from Cedar: no additional threads forked for user interface activity

Dynamic behavior: thread-switching rates

Bi-modal distribution of execution intervals in both Cedar and GVX

- Most execution intervals between 0 and 5 milliseconds (eternal, transient)
- Another peak at 45 milliseconds (note time-slice period is 50 milliseconds)

Switching rate:

- Cedar: Between 130/sec and 270/sec
- GVX: Between 33/sec and 60/sec

Dynamic behavior: priorities

- Cedar
  - 7 priority levels
  - One level unused; user interface activity used higher
- GVX
  - 7 priority levels, “standard” are 1 to 4
  - Almost all threads at level 3

Paradigms of thread usage

Examination of code fragments reveals patterns:

- Defer work
  - Most common use of forking; reduce latency
  - Examples: window update, print document, keyboard-and-mouse watcher
- Pumps; slack processes
  - Components of pipelines (input to output, e.g., bounded buffers, external devices)
  - Preprocessing user input
  - Slack process: adds latency, reduces work (merge, replace data). Example: requests to X server.

Paradigms (continued)

- Sleepers and one-shots
  - Sleeper: repeatedly wait for event and execute
  - Examples: blink cursor, check for network connection timeout, garbage collector, cache manager
  - One-shot: go away after execution
  - Example: guarded button (double click)
- Deadlock avoiders
  - Avoid violating lock order constraints
  - Examples: window repainting, forking callback from server to client

Paradigms (continued)

- Task rejuvenation
  - Recover from bad state, e.g., uncaught exception, stack overflow
  - Fork new copy of service, or error reporting thread
  - Example: avoid fork overhead in input event dispatcher of Cedar
  - Controversial: can mask design flaws
- Serializers
  - Queue and thread processing from queue
  - Example: window system with input events from various sources

Paradigms (continued)

- Concurrency exploiters
  - For using multiple processors
- Encapsulated forks
  - Code modularity; not really a thread paradigm
  - Examples: delayed fork, periodical fork, menu-button queue

Issues in thread use

Trade-offs involving simplicity/concurrency and actual efficiency.

Easy uses:

- Sleeper, one-shot, pump without critical timing constraint, work deferrer

- Require little interaction between threads

Hard thread use issue: X

- Slack process, pump with timing constraint
- Performance of feedback for keyboard activity and mouse motion
- X Window system performance requires batching communication from client to server, merging overlapping requests
- Slack process accumulates and merges paint requests in a buffer
- Imaging thread puts paint requests on queue for buffer thread and issues NOTIFY
- Buffer thread must not act on a single paint request, else no merging! How guarantee this?

Attacking the X problem

- Buffer thread must yield. But what if it is higher priority than image threads?

- Attempt: lower priority of buffer thread? Starvation.

- Workaround: modify thread library by adding primitive *YieldButNotToMe*. In practice, most of time, image thread is favored. But no guarantee.

Common usage mistakes

Incorrectness or performance bugs:

- IF instead of WHILE for monitors
- Timeout hacks to compensate for missing NOTIFY (bug), resulting in performance bug, hard to debug
- Timeouts in general that have inappropriate values chosen, based on now-obsolete properties
- Running out of resources, e.g., fork: hard to deal with. Should implementation try to recover? Or user?

X Window system and multithreading

Contrast thread-safe Xlib with alternate client library, XI, designed with multithreading in mind.

- I/O
  - Modified Xlib: any client thread can read, with monitor. Problems:
    - Priority inversion: blocked client thread preempted
    - Also, need timeout for each read
  - XI: new serializing thread. Uses condition variable mechanism. Other differences:
    - Avoid output flushes, since reading thread can block indefinitely
    - Batching with thread makes connection to server asynchronous

Implementation issues

- Spurious lock conflicts. Thread notified, thread awakened, but notifier still has lock.
  - Solution: delay rescheduling, but not notification.

- Priorities. Need to provide some CPU to all runnable threads. Stable priority inversion problem: high priority waits on lock by low priority, not runnable because of middle priority.
  - Solution: for per-monitor metalock, donate cycles from a blocked thread to blocker thread; provide *SystemDaemon* which donates time to a random thread.
  - They are hacks! Makes semantics of priorities unclear, hard to reason about code

Time-slice quantum

Dependency of X server *YieldButNotToMe* solution on quantum: X event buffering. Need to consider needs of keyboard and other user interaction also.

Conclusions

Studies were made of two interactive systems using threads heavily, and taxonomies drawn up.

Interesting difficulties were discovered both in use and implementation.