Hidden in Plain Sight
Bryan Cantrill
February 23, 2006
ACM Queue

Summarized by
Yunyong Teng-amnuay
November 17, 2009
For
2110522 UNIX/Linux for Enterprise Environments

Background

- 1997 Sun built 64-processor SMP server with upto 64 GB and thousands of I/O devices → a flagship
- Performance measure → benchmarking
- Server paused benchmark for several minutes but still very busy.
- Symptom → high lock contention in network stack

Hard to Fix

- Hypothesis → data gathering → custom kernel modules → very delicate
- Each cycle took several hours
- A reboot took 90 minutes
- Root cause → server configured as a router
- Router failure in the lab → server became a router
- A rather simple problem

The Problem

- Lack of software observability
- “We have built mind-bogglingly complicated systems that we cannot see, allowing glaring performance problems to hide in broad daylight in our systems.”

Roots of the Problem

- Duality of software → information / machine
- s/w has no physical manifestation → do not exhibit any physical properties when running
- In the old days → panel of running lights → cute → not any more
- s/w constructs added to make s/w observable → ใส่ print เข้าไปเยอะๆ

Old School Debugging

- Additional constructs → system ทำงานช้าลง
- Stripped out in ***production system***
- 2 versions of software → observable in development & testing vs. unobservable production version
- Performance problem occurs in production → must be duplicated in development/testing environment

Old School Debugging (2)

- In sufficiently complicated systems → Two causes can have the same symptom.
- "Good news: We were able to reproduce it in dev, and we think that we found the problem. Dev has a fix, and we're scheduling downtime for tonight to put it into prod. Let's keep our fingers crossed..."
- The next day: "Hey, yeah, about that fix...well, the good news is that we are faster this morning; the bad news is that we're only about 3 percent faster. So it's back to the war room..."

Problem Getting Worse

- The dark side of software layering/abstraction
- At higher layer of abstraction → less code can do more work → ก้าวพลาดก็พังเป็นแถบๆ
- Mistakes at high level → problems at low level
  - High CPU utilization
  - Memory pressure
  - Abnormal I/O activities
  - Excessive network traffic

Myth on Performance

- Performance problems are introduced at highest level of abstraction but appear at lowest level of abstraction
- Fixes
  - Faster hardware
  - Faster os / database / better compilers
- Hunting vermin to feed the family: squirrel vs. cow

Constraints on a Solution

- Observability infrastructure
  - Development → Production
  - Programs → Systems
- Implications:
  - Zero disable probe effects
  - Entire stack instrument-able

Constraints on a Solution (2)

- Observation on production version
  - No source
  - No compile-time options
  - No restarts
- Infra → absolutely safe – absolute non-negotiable constraint on architecture

One Solution: DTrace

- Production system
- Safety constraint
- Dynamic instrumentation – s/w trap
- Delicate contexts: interrupt handling / context switching / synchronization
- Instrument separated from data consumer
- Instrument → providers → safety concern

One Solution: Dtrace (2)

- Consumer enables only published probes → compromise for safety
- Actions (and data) wanted at probes → programmable
- Virtual machine → execute user codes in kernel without side effect
- Limited instruction set → safety assured
- D → C-like language with extensions: no loops / no user-defined functions, etc.

One Solution: Dtrace (3)

- Powerful predicate-based filtering
- Data aggregation with n-tuple at the source (in situ) → layer cut-through
- Scalable on complex systems
- Abstract probe names → semantic based
- Providers for: Java, Perl, PHP, Python, Ruby, Tcl, and APL

Example Trace

- Page 5-6 /usr/bin/ps problem