Synthesis

CS380L: Mike Dahlin

September 18, 2002

Qua. Henry Massalin, many, many times.

1 Preliminaries

1.1 Review

• Hydra
  
  – Microkernel → need extensibility → need non-hierarchical security
  – Capabilities
  – Lessons:
    * Technical: Capability-based security (“friend”, Java2, .net, ...
    * Research: Dangerous to build framework (“so we can do lots of optimizations and build lots of neat services”) before knowing what you want to use it for. If you have 1 or 2 optimizations in mind, build them first and build a clean framework once you have concrete experience. If you have 10 in mind, build 1 or 2 first, then the framework, then the other 8. If you have 0 in mind, stop.
  Unix (according to paper) took 2 work-years to build. Hydra team (according to paper) had been working for a year since core Hydra built to build environment on framework. Build framework + extensions v. build simple system?
  * Research: Learn the classic literature...Dan Wallach example.
1.2 Outline

- Synthesis
  - Extreme research
  - Minimalism
  - I/O oriented OS

- Projects

1.3 Preview

- Structure/extensibility: Synthesis, Exokernel, Disco
- Concurrency: Mesa, threads
- Communication: RPC, Active messages, duality of memory and communication

2 Extreme research

- Build an I/O machine: restructure kernel + Lots of low-level optimizations
  - minimize kernel: super-efficient context switch, ..
  - synthesize code for special case (constant folding, inlining, ...)
  - executable data structures
  - minimize synchronization
  - clean I/O processing structure: composable pipeline of filters (+ other optimizations to make the clean abstraction fast).

- Oh, and by the way
  - write a C compiler
    * that does run-time compilation
    * and compiles at 10,000 lines/second (v. 200 lines per second on contemporary compilers)
    * and novel “super-optimizer”
  - build a custom machine

- Lots of ideas
– But if you are not a genius, how do you use them?
– No clean “3 lessons” in paper...need to (retrospectively) extract key principles...
– My opinion on “lessons” and subsequent echos:
  * Dynamic code generation – shocking then, now routine! (Java)
  * Minimalism: Exokernel – specialized code, dynamic code generation, minimalist context switches + how do I do this securely for arbitrary users?
  * I/O optimized kernel/pipeline structure: Scout, Click Router, SEDA, StageServer, IXP?

3 Minimalism

• How to get absolute maximum speedup?
  – By doing absolute minimum work

• Code synthesis
  – Generate tailor-made code on the fly

• “Executable data structure”
  – Embed code in data structures to avoid data structure traversals and to specialize code for each object (e.g., put context switch code inside of thread control block.)

• Code specialization
  – Special-casing and optimizing for common case

• Minimize synchronization
  – Don’t try this at home!
  – Dangerous to trade robustness of known solutions for a little performance. Only acceptable if (a) you can prove your answer is right and (b) you have measured the performance gain and it is dramatic.
  – Example
Q_put(data){
    h = Q_head;
    if(next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
    Q_head = next(h);
}

Hydra Single-writer, Single-reader queue

– Idea: memory write acts as “atomic commit” of update
– Works on uniprocessor with compiler that guarantees not to reorder instructions
– Do you want these things scattered throughout your code?

• How much gain from each individual optimization? Which ones were most important?
3.1 Example: Context switch
• Per-thread interrupt vector
  – On context thread, change processor interrupt base register to point to the vector of the newly executing thread
  – Per-thread system calls, error traps
    * E.g., optimize “read” for a process that has one open descriptor and is reading from a pipe to avoid layers of demultiplexing, etc.
    * Interrupts – mostly the same across all threads (why?)
    * Timer interrupt points to context switch code (sw_out)

• Per-thread context save code
  – Save or not save floating point registers

• “Executable data structure”
  – Timer interrupt vectored directly to current thread’s sw_out; sw_out calls (directly) next thread’s sw_in or sw_in_mmu
    * interrupt vectored to sw_out
    * sw_out saves registers
    * sw_out jumps to next sw_in_mmu
    * sw_in_mmu updates MMU
    * sw_in_mmu updates CPU interrupt vector base
    * sw_in_mmu restores CPU registers (including putting user-PC into the user-PC register)
    * sw_in_mmu does return from exception (replacing PC with user-PC and changing mode back to user mode)

3.2 Lessons/issues from Synthesis minimalism

• Does this approach generalize to a big real OS?
  – Assembly language as “fast prototyping language”
  – Self-changing code
  – Limited functionality

• How much does this matter for real applications?
  – Kernel primitives have impressive performance
– How much does this translate to app performance?
– System was never complete enough to run real benchmarks

• Implications
  – Bounding performance
  – The minimalism principle revisited many more times
  – Clean abstraction/framework is the key to make it usable

• Prelude to Exokernel
  – Specialized code
  – Dynamic code generation
  – Minimalist context switches
  – much cleaner abstractions
  – How do I do all this securely for arbitrary users?

• What remains (one thought..)
  – Exokernel is cleaner abstraction but still “build from scratch”
  – → Came much closer to completion (real benchmarks) but still never complete enough for day to day use, let alone commercial adoption
  – Can fast, specialized adaptations live next to existing kernel structures (so can optimize the stuff that matters but leave the rest there for backwards compatibility?)

4 I/O Oriented OS

• Traditional OS: Schedule CPU and isolate processes
  – Issues: CPU scheduling (hence many papers on this)
  – Context switch at timer interrupt → make context switch faster than quantum and you’re done
  – Communication is uncommon case (hence unix wins, Multics and Hydra lose.)

• New OS: Process I/O and filter data???
– Examples: DSP, Media Processor (Sony Playstation 2, Microsoft/Nvidia Xbox, MicroUnity), programmable router (Cisco, Intel, ...)
– Non-issue: CPU scheduling – everything is interrupt driven; timer interrupt at quantum expiration is uncommon case
– Computation driven by data arrival → need fast context switches
  * Skeptic: no, hardware just needs to buffer a set of arrivals to amortize interrupt overhead.
– Core abstraction: Pipeline of stages process requests
  * Scout (arizona), Click (MIT), SEDA (Berkeley), StageServer (Microsoft), Morphware (UT)

5 Admin

project selections