EECS 262a
Advanced Topics in Computer Systems
Lecture 25

ExoKernel/seL4 Kernel
November 27th, 2018

John Kubiatowicz
(With slides from Ion Stoica and Ali Ghodsi)
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

Today's Papers

- **Exokernel: An Operating System Architecture for Application-Level Resource Management**, Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Appeared in Proceedings of the ACM Symposium on Operating Systems Principles, 2005
- **seL4: Formal Verification of an OS Kernel**, Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood. Appeared in Proceedings of the ACM Symposium on Operating Systems Principles, 2009

- Thoughts?

The Rise of Microkernels!

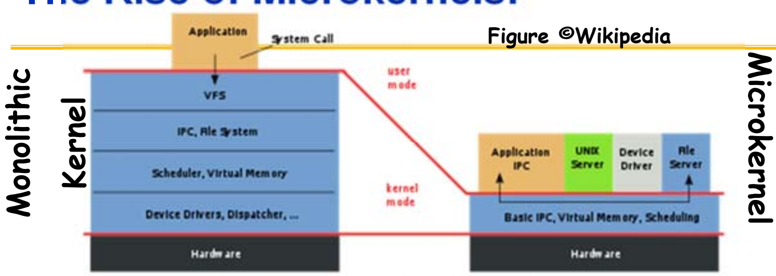


Figure ©Wikipedia

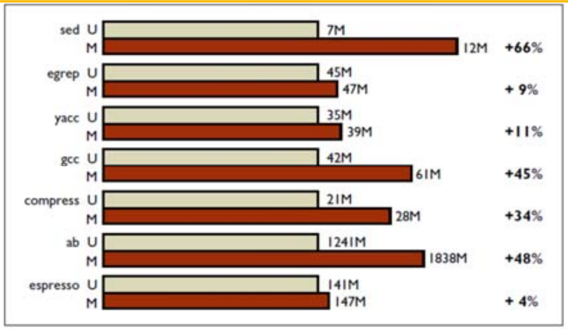
- Moves as much from the kernel into "*user*" space
  – Small core OS running at kernel level
  – OS Services built from many independent user-level processes
  – Communication between modules with message passing
- Benefits:
  – Easier to extend a microkernel
  – Easier to port OS to new architectures
  – More reliable (less code is running in kernel mode)
  – Fault Isolation (parts of kernel protected from other parts)
  – More secure
- Detriments:
  – Performance overhead severe for naïve implementation

Traditional OS services: Management and Protection

- Provides a set of abstractions
  – Processes, Threads, Virtual Memory, Files, IPC
  – APIs, e.g.,: POSIX
- Resource Allocation and Management
- Protection and Security
  – Concurrent execution

4

On the cost of micro-kernels:



- Above: Ultrix (TAN) vs Mach (Reddish-brown)
- Cost of external pagers: page fault of first-generation microkernels took up to 1000µs!
- See optional paper “Toward Real Microkernels” by Leidtke
- After many iterations, L4 became a “best-in-class” microkernel
  - Fast IPC
  - Better resource control than simple pagers (“Address Space Managers”)
  - Other optimizations

Abstractions

- What is an abstraction? Generalization
  - Often an API in CS. Hides implementation details.
- What are the advantages of abstractions?
  - Simpler. Easy to understand and use. Just follow the contract. How we fight complexity.
  - Standardization. Many implementations all satisfy the abstraction. Loose coupling, e.g. Unix’ everything is a file, many implementations and all apps benefit from this standardization.
- What are the disadvantages of abstractions?
  - Contract is a compromise. Least common denominator. Not perfect for each use case
  - Performance often suffers (if you only could tweak an implementation detail of a particular implementation)
  - Can create bloated software.

Context for These Papers (1990s)

- Windows was dominating the market
  - Mac OS downward trend (few percents)
  - Unix market highly fragmented (few percents)

- OS research limited impact
  - Vast majority of OSes proprietary
  - **“Is OS research dead?”**, popular panel topic at systems conferences of the era
- An effort to reboot the OS research, in particular, and OS architecture, in general

Challenge: “Fixed” Interfaces

- Both papers identify “fixed interfaces” provided by existing OSes as main challenge
  - Fixed interfaces provide protection but hurt performance and functionality
- Exokernel:
  - “*Fixed high-level abstractions hurt application performance because there is no single way to abstract physical resources or to implement an abstraction that is best for all applications.*”
  - “*Fixed high-level abstractions limit the functionality of applications, because they are the only available interface between applications and hardware resources*”

Challenge: “Fixed” Interfaces

- Both papers identify “fixed interfaces” provided by existing OSes as main challenge
  - Fixed interfaces provide protection but hurt performance and functionality
- SPIN:
  - *“Existing operating systems provide fixed interfaces and implementations to system services and resources. This makes them inappropriate for applications whose resource demands and usage patterns are poorly matched by the services provided.”*

Problems in existing OSes

- Extensibility
  - Abstractions overly general
  - Apps cannot dictate management
  - Implementations are fixed
- Performance
  - Context switching expensive
  - Generalizations and hiding information affect performance
- Protection and Management offered with loss in Extensibility and Performance

Symptoms

- Very few of innovations making into commercial OSes
  - E.g., scheduler activations, efficient IPC, new virtual memory policies, …
- Applications struggling to get better performances
  - They knew better how to manage resources, and the OS was “standing” in the way

Examples Illustrating need for App Control

- Databases know better than the OS what pages they will access
  - Can prefetch pages, LRU hurts their performance, why?

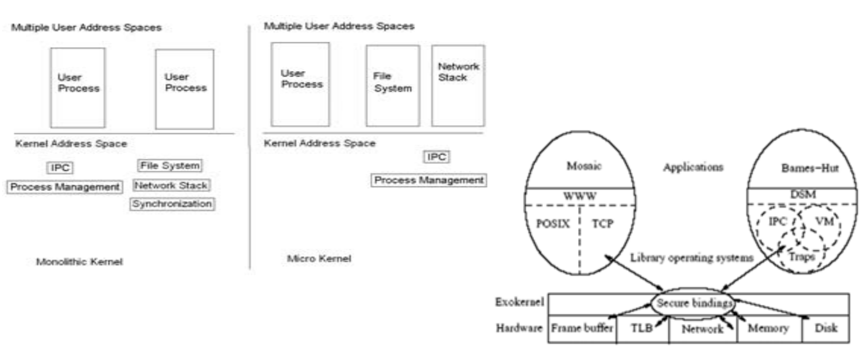
Two Papers, Two Approaches

- Exokernel:
  - Very minimalist kernel, most functionality implemented in user space
  - Assumed many apps have widely different requirements, maximal extensibility
- SPIN:
  - Dynamically link extensions into the kernel
  - Rely on programming language features, e.g. static typechecking
  - Assumed device drivers need flexibility, so focused on how to enable them while staying protected

Exokernel

- A nice illustration of the end-to-end argument:
  - ``general-purpose implementations of abstractions force applications that do not need a given feature to pay substantial overhead costs.''
  - In fact the paper is explicitly invoking it (sec 2.2)!
- Corollary:
  - Kernel just safely exposes resources to apps
  - Apps implement everything else, e.g., interfaces/APIs, resource allocation pollcies

OS Component Layout



Exokernel

Exokernel Main Ideas

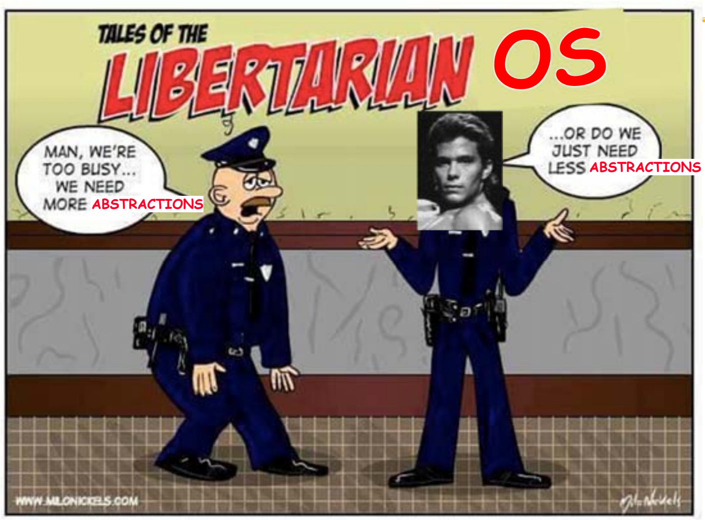
- Kernel: resource sharing, not policies
- Library Operating System: responsible for the abstractions
  - IPC
  - VM
  - Scheduling
  - Networking

Lib OS and the Exokernel

- Lib OS (untrusted) can implement traditional OS abstractions (compatibility)
- Efficient (LibOS in user space)
- Apps link with LibOS of their choice
- Kernel allows LibOS to manage resources, protects LibOSes

Philosophy

- *"An exokernel should avoid resource management. It should only manage resources to the extent required by protection (i.e., management of allocation, revocation, and ownership)."*
- The motivation for this principle is our belief that distributed, application-specific, resource management is the best way to build efficient flexible systems.



Exokernel design

- Securely expose hardware
  - Decouple authorization from use of resources
  - Authorization at bind time (i.e., granting access to resource)
  - Only access checks when using resource
  - E.g., LibOS loads TLB on TLB fault, and then uses it multiple times
- Expose allocation
  - Allow LibOSes to request specific physical resources
  - Not implicit allocation; LibOS should participate in every allocation decision

Exokernel design

- Expose names (CS trick #1^{-1})
  - Remove one level of indirection and expose useful attributes
    - » E.g., index in direct mapped caches identify physical pages conflicting
  - Additionally, expose bookkeeping data structures
    - » E.g., freelist, disk arm position (?), TLB entries
- Expose revocation
  - "Polite" and then forcibly abort

Example: Memory

Self-authenticated capability

- Guard TLB loads and DMA
  - Secure binding: using self-authenticating capabilities
    - » For each page Exokernel creates a random value, check
    - » Exokernel records: {Page, Read/Write Rights, MAC(check, Rights)}
  - When accessing page, owner need to present capability
  - Page owner can change capabilities associated and deallocate it
- Large Software TLB (why?)
  - TLB of that time small, LibOS can manage a much bigger TLB in software
  - Expensive checks during page fault can be reduced with a larger TLB

Example: Processor Sharing

- Process time represented as linear vector of time slices
  - Round robin allocation of slices
- Secure binding: allocate slices to LibOSes
  - Simple, powerful technique: donate time slice to a particular process
  - A LibOS can donate unused time slices to its process of choice
- If process takes excessive time, it is killed (revocation)

Example: Network

- Downloadable filters
- Application-specific Safe Handlers (ASHes)
  - Can reply directly to traffic, e.g., can implement new transport protocols; dramatically reduce
- Secure biding happens at download time

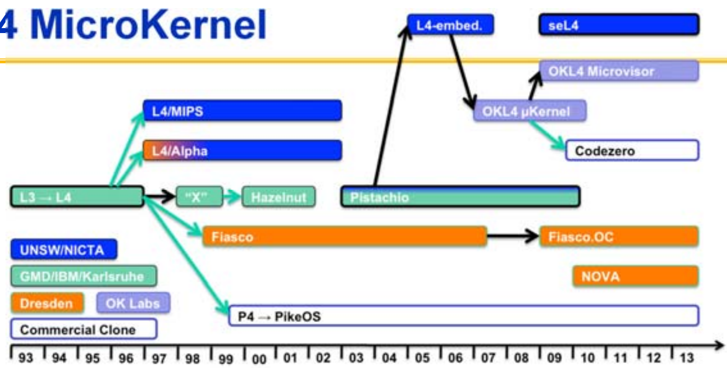
Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Final Project Timing

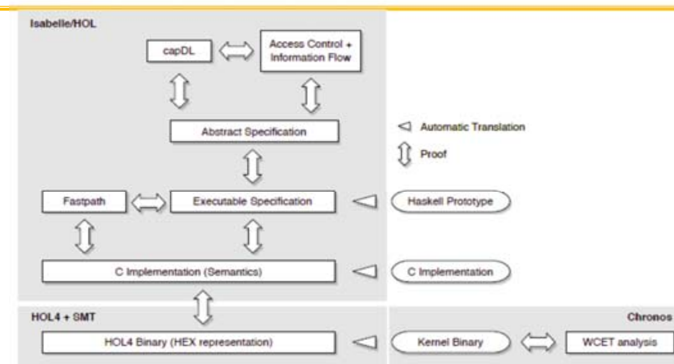
- Final abstract/project proposal on the WEBSITE
  - Please update your project description and proposal before next week
- Poster Session:
  - Next Friday (12/7) from 9:00-12:00 in 5th-floor atrium
  - Everyone must be setup by 9:30 – if you are late, you may not get a chance to have your poster reviewed
  - Plan on staying whole time, but might be shorter
  - Who needs posters printed???
- Final paper:
  - Due Monday 12/10 @ AOE (by 5am)
  - 10 pages, 2- column, conference format
  - Must have a related work section!
  - Also, plan on a future work and/or discussion section
  - Make sure that your METRICs of success are clear and available

L4 MicroKernel



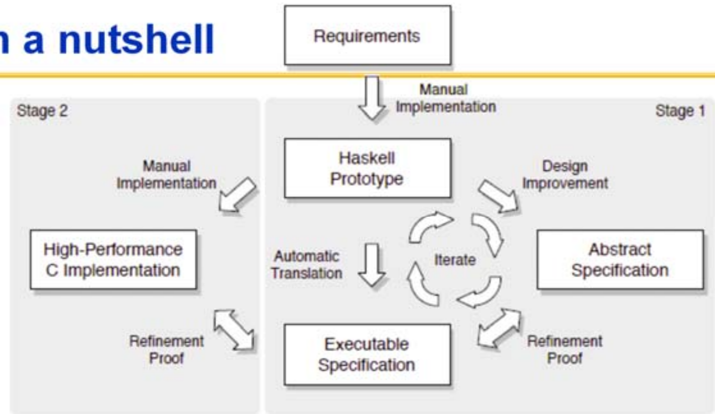
- Original L4 kernel: Jochen Liedtke
  - Derived from L3 kernel
  - Developed at GMD (Germany)
  - Designed to have fast IPC, better memory management,
- Picked up in several commercial usages

Layers of Proof Techniques



- Results (including follow-on results)
  - Functional correctness verification in Isabelle/HOL framework
  - Functional correctness for hand-optimized IPC
  - Correctness of access-control enforcement
  - Automatic poof of refinement between compiled binary and C implementation
  - Automatic static analysis of worst-case execution time for sys calls

In a nutshell



- Apply automatic techniques for "proof" of correctness
  - Series of *refinements* from Abstract Specification ⇒ Executable Specification ⇒ C implementation
  - Utilized Isabelle/HOL theorem prover
- Three different implementations
  - Spanning "obvious to humans" and "efficient"

Three Implementations

- Abstract Specification: Isabelle/HOL code
  - Describes *what* system does without saying *how* it is done
  - Written as series of assertions about what should be true
- Executable Specification: Haskel
  - Fill in details left open at abstract level
  - Specify *how* the kernel works (as opposed to what I does)
  - Deterministic execution (except for underlying machine)
  - Data structures have explicit data types
  - Controlled subset of Haskel
- The Detailed Implementation: C
  - Translation from C into Isabelle requires controlled subset of C99
    - » No address-of operator & on local (stack) variables
    - » Only 1 function call in expressions or need proof of side-effect freedom
    - » No function calls through pointers
    - » No goto, switch statements with fall-through
- Machine model: need model of how hardware behaves

Example: Scheduling

```
schedule ≡ do
  threads ← all_active_tcbs;
  thread ← select threads;
  switch_to_thread thread
od OR switch_to_idle_thread
```

- Figure 3: Isabell/High-order logic (HOL) code for scheduler at abstract level

```
schedule = do
  action <- getSchedulerAction
  case action of
    ChooseNewThread -> do
      chooseThread
      setSchedulerAction ResumeCurrentThread
      ...
chooseThread = do
    r <- findM chooseThread' (reverse [minBound .. maxBound])
    when (r == Nothing) $ switchToIdleThread
chooseThread' prio = do
    q <- getQueue prio
    liftM isJust $ findM chooseThread'' q
chooseThread'' thread = do
    runnable <- isRunnable thread
    if not runnable then do
            tcbSchedDequeue thread
            return False
    else do
            switchToThread thread
            return True
```

- Figure 4: Haskell Code for Scheduler

Challenges

- Smaller Kernel ⇒ Increased Interdependence of components
  - Radical size reduction in microkernel leads to high degree of dependence of remaining components
  - Proof techniques still possible
- Design special versions of each of the languages
- Design multiple implementations each in different languages
- Huge number of invariants is moving from Abstract specification to Executable specification
  - Low-level memory invariants
  - Typing invariants
  - Data structure Invariants
  - Algorithmic Invariants

Experience and Lessons Learned

- Performance
  - Optimized IPC path is well performing (via micro-benchmarks)
  - Other performance not characterized in this paper
- Verification effort: extremely high!
  - Three phases, each with non-trivial effort
  - Overall size of proof (including automatically generated proofs): 200,000 lines of Isabelle script
  - Abstract Spec: 4 person months (4 pm)
  - Haskell prototype: 2 person years (2 py)
  - Executable spec translated from Haskel prototype: 3 pm
  - Initial C translation+implementation: 2pm
  - Cost of the proof: 20py (seL4 specific, 11py)
- What about future: 8py for new kernel from scratch!
- Cost of change
  - Simple or independent small features not huge, say 1 person week
  - Cost to add new features can be very large: up to 2py!

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?