CS 202: Advanced Operating Systems

OS Extensibility: Exo-kernel

Extensibility

- Problem: How?
- Add code to OS
  - how to preserve isolation?
  - … without killing performance?
- What abstractions?
  - General principle: mechanisms in OS, policies through the extensions
  - What mechanisms to expose?

Spin Approach to extensibility

- Co-location of kernel and extension
  - Avoid border crossings
  - But what about protection?
- Language/compiler forced protection
  - Strongly typed language
    - Protection by compiler and run-time
    - Cannot cheat using pointers
  - Logical protection domains
    - No longer rely on hardware address spaces to enforce protection – no boarder crossings
- Dynamic call binding for extensibility

ExoKernel

Motivation for Exokernels

- Traditional centralized resource management cannot be specialized, extended or replaced
- Privileged software must be used by all applications
- Fixed high level abstractions too costly for good efficiency
- Exo-kernel as an end-to-end argument

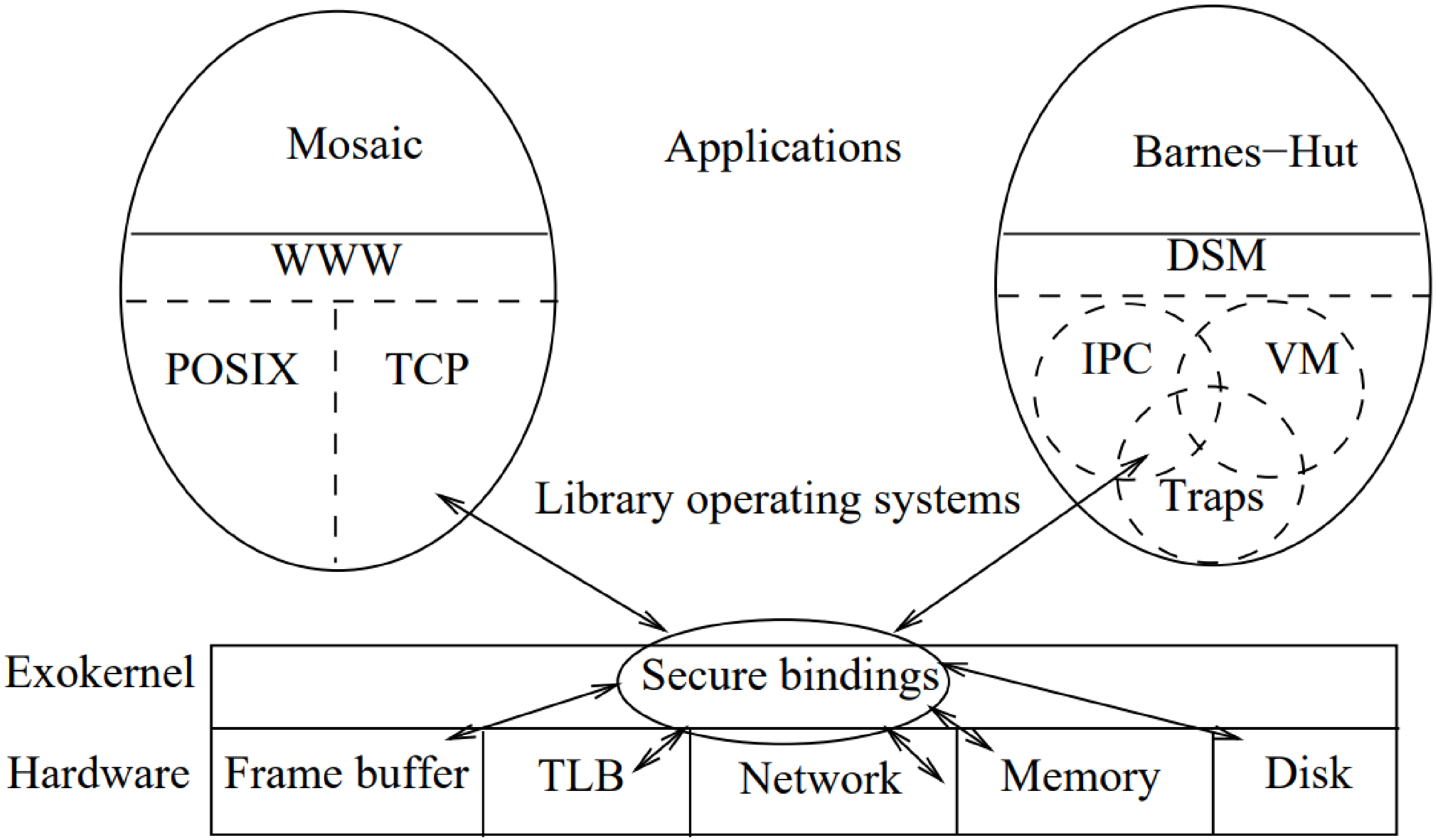
Exokernel Philosophy

- Expose hardware to libraryOS
  - Not even mechanisms are implemented by exo-kernel
    - They argue that mechanism is policy
- Exo-kernel worried only about protection not resource management

Design Principles

- Track resource ownership
- Ensure protection by guarding resource usage
- Revoke access to resources
- Expose hardware, allocation, names and revocation
- Basically validate binding, then let library manage the resource

Exokernel Architecture



Separating Security from Management

- Secure bindings – securely bind machine resources
- Visible revocation – allow libOSes to participate in resource revocation
- Abort protocol – break bindings of uncooperative libOSes

Secure Bindings

- Decouple authorization from use
- Authorization performed at bind time
- Protection checks are simple operations performed by the kernel
- Allows protection without understanding
- Operationally – set of primitives needed for applications to express protection checks

Example resource

- TLB Entry
  - Virtual to physical mapping done by library
  - Binding presented to exo-kernel
  - Exokernel puts it in hardware TLB
  - Process in library OS then uses it without exo-kernel intervention

Implementing Secure Bindings

- Hardware mechanisms: TLB entry, Packet Filters
- Software caching: Software TLB stores
- Downloaded Code: invoked on every resource access or event to determine ownership and kernel actions

Downloaded Code Example: (DPF) Downloaded Packet Filter

- Eliminates kernel crossings
- Can execute when application is not scheduled
- Written in a type safe language and compiled at runtime for security
- Uses Application-specific Safe Handlers which can initiate a message to reduce round trip latency

Visible Resource Revocation

- Traditionally resources revoked invisibly
- Allows libOSes to guide de-allocation and have knowledge of available resources – ie: can choose own ‘victim page’
- Places workload on the libOS to organize resource lists

Abort Protocol

- Forced resource revocation
- Uses ‘repossession vector’
- Raises a repossession exception
- Possible relocation depending on state of resource

Managing core services

- Virtual memory:
  - Page fault generates an upcall to the library OS via a registered handler
  - LibOS handles the allocation, then presents a mapping to be installed into the TLB providing a capability
  - Exo-kernel installs the mapping
  - Software TLBs

Managing CPU

- A time vector that gets allocated to the different library operating systems
  - Allows allocation of CPU time to fit the application
- Revokes the CPU from the OS using an upcall
  - The libOS is expected to save what it needs and give up the CPU
  - If not, things escalate
  - Can install revocation handler in exo-kernel

Putting it all together

- Lets consider an exo-kernel with downloaded code into the exo-kernel
- When normal processing occurs, Exo-kernel is a sleeping beauty
- When a discontinuity occurs (traps, faults, external interrupts), exokernel fields them
  - Passes them to the right OS (requires book-keeping) – compare to SPIN?
  - Application specific handlers

Evaluation

- Again, a full implementation
- How to make sense from the quantitative results?
  - Absolute numbers are typically meaningless given that we are part of a bigger system
    - Trends are what matter
- Again, emphasis is on space and time
  - Key takeaway→ at least as good as a monolithic kernel

Questions and conclusions

- Downloaded code – security?
  - Some mention of SFI and little languages
  - SPIN is better here?
- SPIN vs. Exokernel
  - Spin—extend mechanisms; some abstractions still exist
  - Exo-kernel: securely expose low-level primitives (primitive vs. mechanism?)
- Microkernel vs. exo-kernel
  - Much lower interfaces exported
  - Argue they lead to better performance
  - Of course, less border crossing due to downloadable code

Conclusions

- Simplicity and limited exokernel primitives can be implemented efficiently
- Hardware multiplexing can be fast and efficient
- Traditional abstractions can be implemented at the application level
- Applications can create special purpose implementations by modifying libraries