OS Extensibility: Spin, Exo-kernel and L4
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
On Microkernel construction (L3/4)
L4 microkernel family

- Successful OS with different offshoot distributions
  - Commercially successful
    - OKLabs OKL4 shipped over 1.5 billion installations by 2012
      - Mostly qualcomm wireless modems
      - But also player in automotive and airborne entertainment systems
    - Used in the secure enclave processor on Apple’s A7 chips
      - All iOS devices have it! 100s of millions
Big picture overview

- Conventional wisdom at the time was:
  - Microkernels offer nice abstractions and should be flexible
  - …but are inherently low performance due to high cost of border crossings and IPC
  - …because they are inefficient they are inflexible

- This paper refutes the performance argument
  - Main takeaway: its an implementation issue
    - Identifies reasons for low performance and shows by construction that they are not inherent to microkernels
    - 10-20x improvement in performance over Mach

- Several insights on how microkernels should (and shouldn’t) be built
  - E.g., Microkernels should not be portable
Paper argues for the following

- Only put in anything that if moved out prohibits functionality
- Assumes:
  - We require security/protection
  - We require a page-based VM
  - Subsystems should be isolated from one another
  - Two subsystems should be able to communicate without involving a third
Abstractions provided by L3

- Address spaces (to support protection/separation)
  - Grant, Map, Flush
  - Handling I/O

- Threads and IPC
  - Threads: represent the address space
  - End point for IPC (messages)
  - Interrupts are IPC messages from kernel
    - Microkernel turns hardware interrupts to thread events

- Unique ids (to be able to identify address spaces, threads, IPC end points etc..)
Debunking performance issues

What are the performance issues?

1. Switching overhead
   - Kernel user switches
   - Address space switches
   - Threads switches and IPC

2. Memory locality loss
   - TLB
   - Caches
Mode switches

- System calls (mode switches) should not be expensive
  - Called context switches in the paper
- Show that 90% of system call time on Mach is “overhead”
  - What? Paper doesn’t really say
    - Could be parameter checking, parameter passing, inefficiencies in saving state…
  - L3 does not have this overhead
Thread/address space switches

- If TLBs are not tagged, they must be flushed
  - Today? x86 introduced tags but they are not utilized
- If caches are physically indexed, no loss of locality
  - No need to flush caches when address space changes
- Customize switch code to HW
- Empirically demonstrate that IPC is fast
Review: End-to-end Core i7 Address Translation

CPU

Virtual address (VA)

VPN VPO

TLBT TLBI

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

CR3

Page tables

PTE PTE PTE PTE

TLB hit

L1 hit

L2, L3, and main memory

32/64

Result

L1 miss

L1 d-cache (64 sets, 8 lines/set)

Physical address (PA)

CT CI CO

Physical address (PA)

VPN1 VPN2 VPN3 VPN4

PPN PPO

L1, L2, and main memory

VPN1 VPN2 VPN3 VPN4
Tricks to reduce the effect

- TLB flushes due to AS switch could be very expensive
  - Since microkernel increases AS switches, this is a problem
  - Tagged TLB? If you have them
  - Tricks with segments to provide isolation between small address spaces
    - Remap them as segments within one address space
    - Avoid TLB flushes
Memory effects

- Chen and Bershad showed memory behavior on microkernels worse than monolithic
- Paper shows this is all due to more cache misses
- Are they capacity or conflict misses?
  - Conflict: could be structure
  - Capacity: could be size of code
- Chen and Bershad also showed that self-interference more of a problem than user-kernel interference
- Ratio of conflict to capacity much lower in Mach
  - too much code, most of it in Mach
Conclusion

- It's an implementation issue in Mach
- It's mostly due to Mach trying to be portable
- Microkernel should not be portable
  - It's the hardware compatibility layer
  - Example: implementation decisions even between 486 and Pentium are different if you want high performance
  - Think of microkernel as microcode
Extra SLIDES-- FYI
Aegis and ExOS

- Aegis exports the processor, physical memory, TLB, exceptions, interrupts and a packet filter system
- ExOS implements processes, virtual memory, user-level exceptions, interprocess abstractions and some network protocols
- Only used for experimentation
Aegis Implementation Overview

- Multiplexes the processor
- Dispatches Exceptions
- Translates addresses
- Transfers control between address spaces
- Multiplexes the network
Processor Time Slices

- CPU represented as a linear vector of time slices
- Round robin scheduling
- Position in the vector
- Timer interrupts denote beginning and end of time slices and is handled like an exception
Null Procedure and System Call Costs

<table>
<thead>
<tr>
<th>Machine</th>
<th>OS</th>
<th>Procedure call</th>
<th>Syscall (getpid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC2100</td>
<td>Ultrix</td>
<td>0.57</td>
<td>32.2</td>
</tr>
<tr>
<td>DEC2100</td>
<td>Aegis</td>
<td>0.56</td>
<td>3.2 / 4.7</td>
</tr>
<tr>
<td>DEC3100</td>
<td>Ultrix</td>
<td>0.42</td>
<td>33.7</td>
</tr>
<tr>
<td>DEC3100</td>
<td>Aegis</td>
<td>0.42</td>
<td>2.9 / 3.5</td>
</tr>
<tr>
<td>DEC5000</td>
<td>Ultrix</td>
<td>0.28</td>
<td>21.3</td>
</tr>
<tr>
<td>DEC5000</td>
<td>Aegis</td>
<td>0.28</td>
<td>1.6 / 2.3</td>
</tr>
</tbody>
</table>
Aegis Exceptions

- All hardware exceptions passed to applications
- Save scratch registers into ‘save area’ using physical addresses
- Load exception program counter, last virtual address where translation failed and the cause of the exception
- Jumps to application specified program counter where execution resumes
## Aegis vs. Ultrix Exception Handling Times

<table>
<thead>
<tr>
<th>Machine</th>
<th>OS</th>
<th>unalign</th>
<th>overflow</th>
<th>coproc</th>
<th>prot</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC2100</td>
<td>Ultrix</td>
<td>n/a</td>
<td>208.0</td>
<td>n/a</td>
<td>238.0</td>
</tr>
<tr>
<td>DEC2100</td>
<td>Aegis</td>
<td>2.8</td>
<td>2.8</td>
<td>2.8</td>
<td>3.0</td>
</tr>
<tr>
<td>DEC3100</td>
<td>Ultrix</td>
<td>n/a</td>
<td>151.0</td>
<td>n/a</td>
<td>177.0</td>
</tr>
<tr>
<td>DEC3100</td>
<td>Aegis</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>2.3</td>
</tr>
<tr>
<td>DEC5000</td>
<td>Ultrix</td>
<td>n/a</td>
<td>130.0</td>
<td>n/a</td>
<td>154.0</td>
</tr>
<tr>
<td>DEC5000</td>
<td>Aegis</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Address Translation

- Bootstrapping through ‘guaranteed mapping’
- Virtual addresses separated into two segments:
  - Normal data and code tables and exception code
Protected Control Transfer

- Changes program counter to value in the callee
- Asynchronous calling process donates remainder of time slice to callee’s process environment – Synchronous calls donate all remaining time slices
- Installs callee’s processor context (address-context identifier, address-space tag, processor status word)
- Transfer is atomic to processes
- Aegis will not overwrite application visible registers
## Protected Control Transfer Times Compared with L3

<table>
<thead>
<tr>
<th>OS</th>
<th>Machine</th>
<th>MHz</th>
<th>Transfer cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aegis</td>
<td>DEC2100</td>
<td>12.5MHz</td>
<td>2.9</td>
</tr>
<tr>
<td>Aegis</td>
<td>DEC3100</td>
<td>16.67MHz</td>
<td>2.2</td>
</tr>
<tr>
<td>Aegis</td>
<td>DEC5000</td>
<td>25MHz</td>
<td>1.4</td>
</tr>
<tr>
<td>L3</td>
<td>486</td>
<td>50MHz</td>
<td>9.3 (normalized)</td>
</tr>
</tbody>
</table>
Dynamic Packet Filter (DPF)

- Message demultiplexing determines which application a message should be delivered to
- Dynamic code generation is performed by VCODE
- Generates one executable instruction in 10 instructions

<table>
<thead>
<tr>
<th>Filter</th>
<th>Classification Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPF</td>
<td>35.0</td>
</tr>
<tr>
<td>PATHFINDER</td>
<td>19.0</td>
</tr>
<tr>
<td>DPF</td>
<td>1.5</td>
</tr>
</tbody>
</table>
ExOS: A Library Operating System

- Manages operating system abstractions at the application level within the address space of the application using it
- System calls can perform as fast as procedure calls
IPC Abstractions

- Pipes in ExOS use a shared memory circular buffer
- Pipe uses inline read and write calls
- Shm shows times of two processes to ‘ping-pong’ – simulated on Ultrix using signals
- Lrpc is single threaded, does not check permissions and assumes a single function is of interest
## IPC Times Compared to Ultrix

<table>
<thead>
<tr>
<th>Machine</th>
<th>OS</th>
<th>pipe</th>
<th>pipe</th>
<th>shm</th>
<th>lrpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC2100</td>
<td>Ultrix</td>
<td>326.0</td>
<td>n/a</td>
<td>187.0</td>
<td>n/a</td>
</tr>
<tr>
<td>DEC2100</td>
<td>ExOS</td>
<td>30.9</td>
<td>24.8</td>
<td>12.4</td>
<td>13.9</td>
</tr>
<tr>
<td>DEC3100</td>
<td>Ultrix</td>
<td>243.0</td>
<td>n/a</td>
<td>139.0</td>
<td>n/a</td>
</tr>
<tr>
<td>DEC3100</td>
<td>ExOS</td>
<td>22.6</td>
<td>18.6</td>
<td>9.3</td>
<td>10.4</td>
</tr>
<tr>
<td>DEC5000</td>
<td>Ultrix</td>
<td>199.0</td>
<td>n/a</td>
<td>118.0</td>
<td>n/a</td>
</tr>
<tr>
<td>DEC5000</td>
<td>ExOS</td>
<td>14.2</td>
<td>10.7</td>
<td>5.7</td>
<td>6.3</td>
</tr>
</tbody>
</table>

Table 8: Time for IPC using pipes, shared memory, and LRPC on ExOS and Ultrix; times are in microseconds. Pipe and shared memory are unidirectional, while LRPC is bidirectional.
Application-level Virtual Memory

- Does not handle swapping
- Page tables are implemented as a linear vector
- Provides aliasing, sharing, enabling disabling caching on a per page basis, specific page-allocation and DMA
## Virtual Memory Performance

<table>
<thead>
<tr>
<th>Machine</th>
<th>OS</th>
<th>dirty</th>
<th>prot1</th>
<th>prot100</th>
<th>unprotected100</th>
<th>trap</th>
<th>appel1</th>
<th>appel2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC2100</td>
<td>Ultrix</td>
<td>n/a</td>
<td>51.6</td>
<td>175.0</td>
<td>175.0</td>
<td>240.0</td>
<td>383.0</td>
<td>335.0</td>
</tr>
<tr>
<td>DEC2100</td>
<td>ExOS</td>
<td>17.5</td>
<td>32.5</td>
<td>213.0</td>
<td>275.0</td>
<td>13.9</td>
<td>74.4</td>
<td>45.9</td>
</tr>
<tr>
<td>DEC3100</td>
<td>Ultrix</td>
<td>n/a</td>
<td>39.0</td>
<td>133.0</td>
<td>133.0</td>
<td>185.0</td>
<td>302.0</td>
<td>267.0</td>
</tr>
<tr>
<td>DEC3100</td>
<td>ExOS</td>
<td>13.1</td>
<td>24.4</td>
<td>156.0</td>
<td>206.0</td>
<td>10.1</td>
<td>55.0</td>
<td>34.0</td>
</tr>
<tr>
<td>DEC5000</td>
<td>Ultrix</td>
<td>n/a</td>
<td>32.0</td>
<td>102.0</td>
<td>102.0</td>
<td>161.0</td>
<td>262.0</td>
<td>232.0</td>
</tr>
<tr>
<td>DEC5000</td>
<td>ExOS</td>
<td>9.8</td>
<td>16.9</td>
<td>109.0</td>
<td>143.0</td>
<td>4.8</td>
<td>34.0</td>
<td>22.0</td>
</tr>
</tbody>
</table>
Application-Specific Safe Handlers (ASH)

- Downloaded into the kernel
- Made safe by code inspection, sandboxing
- Executes on message arrival
- Decouples latency critical operations such as message reply from scheduling of processes
ASH Continued

- Allows direct message vectoring – eliminating intermediate copies
- Dynamic integrated layer processing – allows messages to be aggregated to a single point in time
- Message initiation – allows for low-latency message replies
- Control initiation – allows general computations such as remote lock acquisition
## Roundtrip Latency of 60-byte packet

<table>
<thead>
<tr>
<th>Machine</th>
<th>OS</th>
<th>Roundtrip latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC5000/125</td>
<td>ExOS/ASH</td>
<td>259</td>
</tr>
<tr>
<td>DEC5000/125</td>
<td>ExOS</td>
<td>320</td>
</tr>
<tr>
<td>DEC5000/125</td>
<td>Ultrix</td>
<td>3400</td>
</tr>
<tr>
<td>DEC5000/200</td>
<td>Ultrix/FRPC</td>
<td>340</td>
</tr>
</tbody>
</table>
Average Roundtrip Latency with Multiple Active Processes on Receiver

![Graph showing roundtrip latency with and without ASH](image-url)
Extensible RPC

- Trusted version of lrpc called tlrpc which saves and restores callee-saved registers

<table>
<thead>
<tr>
<th>Machine</th>
<th>lrpc</th>
<th>tlrpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC2100</td>
<td>13.9</td>
<td>8.6</td>
</tr>
<tr>
<td>DEC3100</td>
<td>10.4</td>
<td>6.4</td>
</tr>
<tr>
<td>DEC50000</td>
<td>6.3</td>
<td>2.9</td>
</tr>
</tbody>
</table>
Extensible Page-table Structures

- Inverted page tables

<table>
<thead>
<tr>
<th>Machine</th>
<th>Method</th>
<th>dirty</th>
<th>prot1</th>
<th>prot100</th>
<th>unprot100</th>
<th>trap</th>
<th>appel1</th>
<th>appel2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC2100</td>
<td>Original page-table</td>
<td>17.5</td>
<td>32.5</td>
<td>213.</td>
<td>275.</td>
<td>13.9</td>
<td>74.4</td>
<td>45.9</td>
</tr>
<tr>
<td>DEC2100</td>
<td>Inverted page-table</td>
<td>8.0</td>
<td>23.1</td>
<td>253.</td>
<td>325.</td>
<td>13.9</td>
<td>54.4</td>
<td>38.8</td>
</tr>
<tr>
<td>DEC3100</td>
<td>Original page-table</td>
<td>13.1</td>
<td>24.4</td>
<td>156.</td>
<td>206.</td>
<td>10.1</td>
<td>55.0</td>
<td>34.0</td>
</tr>
<tr>
<td>DEC3100</td>
<td>Inverted page-table</td>
<td>5.9</td>
<td>17.7</td>
<td>189.</td>
<td>243.</td>
<td>10.1</td>
<td>40.4</td>
<td>28.9</td>
</tr>
</tbody>
</table>
Extensible Schedulers

- Stride scheduling
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