Advanced Operating Systems (CS 202)

Extensible Operating Systems

Jan, 11, 2016
OS Structure

• What does OS structure refer to?
• The way the OS software is organized with respect to the applications it serves and the hardware that it manages so that it can meet its goals
Let's start with the basic idea

• All OS's use “Controlled Direct Execution”
  - User programs when scheduled run directly on the CPU
  - They don’t have to ask permission to use memory – it's virtualized or partitioned somehow (with hardware help)

• OS is a sleeping beauty
  - Gets woken up by events
  - Enough for time sharing?
Why is the structure of an OS important?

• Protection
  - User from user and system from user

• Performance
  - Does the structure facilitate good performance?

• Flexibility/Extensibility
  - Can we adapt the OS to the application

• Scalability
  - Performance goes up with more resources

• Agility
  - Adapt to application needs and resources

• Responsiveness
  - How quickly it reacts to external events

• Can it meet these requirements?
Monolithic Kernel

Hardware, managed by OS

OS Services and Device drivers

Applications
DOS Like Structure

Hardware, managed by OS

OS Services and Device drivers

Applications

What is the difference?
Extensibility/customization

• Applications are very different; consider video game vs. a number crunching application

• Lets consider a concrete application
  – Page fault occurs
  – OS action?
    • Find a free frame (page replacement policy)
    • Update page table
    • Resume process

• Providing extensibility was a big area of research
Motivating Extensibility

• Why not one size fits all?
• Both papers make similar arguments
  – Traditional centralized resource management cannot be specialized, extended or replaced
    • Several papers at the time showed substantial advantages from specializing resource allocation
  – Privileged software must be used by all applications
  – Fixed high level abstractions too costly for good efficiency
    • Also hides information
Micro-kernel

Hardware, managed by OS

File System
Memory manager
CPU scheduler

Applications
IPC, Address Spaces, …
Discussion of microkernels

• Performance loss!
  – A lot of boarder crossings
  – Separate address spaces – expensive
    • Explicit and implicit costs (memory hierarchy)
  – Lets compare a system call for monolithic vs. micro-kernel

• What are the advantages?
Why are microkernels slow?

• Lets consider an example of a file system call
  – Application uses system call to microkernel
  – Microkernel sends message to file server
  – File server does work, then uses ipc to send results back to application
    • This requires a border crossing to micro-kernel
  – Finally switch back to app

• Consider direct cost and loss of locality
• Buffer copies
How expensive are border crossings?

- Procedure call: save some general purpose registers and jump

- Mode switch:
  - Trap or call gate overhead
    - Nowadays syscall/sysreturn
  - Switch to kernel stack
  - Switch some segment registers

- Context switch?
  - Change address space
  - This could be expensive
Summary

• **DOS-like structure:**
  - good performance and extensibility
  - Bad protection

• **Monolithic kernels:**
  - Good performance and protection
  - Bad extensibility

• **Microkernels**
  - Good protection and extensibility
  - Bad performance!
More simply

UNIX

safe

fast

Mach

extensible

Windows/DOS
What should an extensible OS do?

- It should be thin, like a micro-kernel
  - Only mechanisms (or even less?)
  - No policies; they are defined by extensions
- Fast access to resources, like DOS
  - Eliminate boarder crossings
- Flexibility without sacrificing protection or performance
- Basically, fast, protected and flexible
What had been done before?

• **Hydra (Wulf ’81)**
  - Kernel mechanisms for resource allocation
  - Capability based resource access
    - This was expensive as implemented
  - Resource management as coarse grained objects to reduce boarder crossings

• **Microkernel (e.g., Mach in the 90s)**
  - Focus on extensibility and portability
  - Portability hurt performance
  - Gave a bad rep to microkernels
Previous Work (continued)

• Write extensions in “little languages”
  – Allows extensions written in these languages to be added into the kernel
  – Extension code is interpreted by kernel at run time
    • Limited scope of language limits usefulness of approach
Spin Approach to extensibility

- Co-location of kernel and extension
  - Avoid boarder 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
SPIN MECHANISMS/TOOLBOX
Logical protection domains

• Modula-3 safety and encapsulation mechanisms
  – Type safety, automatic storage management
  – Objects, threads, exceptions and generic interfaces

• Fine-grained protection of objects using capabilities. An object can be:
  – Hardware resources (e.g., page frames)
  – Interfaces (e.g., page allocation module)
  – Collection of interfaces (e.g., full VM)

• Capabilities are language supported pointers
Logical protection domains -- mechanisms

• Create:
  – Initialize with:

• Resolve:
  – Names are resolved between a source and a target domain
  – Once resolved, access is at memory speeds

• Combine
  – To create an aggregate domain

• This is the key to spin – protection, extensibility and performance

```
INTERFACE Domain;

TYPE T <: REFANY; (* Domain.T is opaque *)

PROCEDURE Create(coff:CoffFile.T):T;
(* Returns a domain created from the specified object file (‘coff’ is a standard object file format). *)

PROCEDURE CreateFromModule():T;
(* Create a domain containing interfaces defined by the calling module. This function allows modules to name and export themselves at runtime. *)

PROCEDURE Resolve(source,target: T);
(* Resolve any undefined symbols in the target domain against any exported symbols from the source.*)

PROCEDURE Combine(d1, d2: T):T;
(* Create a new aggregate domain that exports the interfaces of the given domains. *)

END Domain.
```
Protection Model (I)

- All kernel resources are referenced by *capabilities* [tickets]
- SPIN implements capabilities directly through the use of pointers
- Compiler prevents pointers to be forged or dereferenced in a way inconsistent with its type at *compile time*:
  - No run time overhead for using a pointer
Protection Model (II)

• A pointer can be passed to a user-level application through an *externalized reference*:
  - Index into a per-application table of safe references to kernel data structures

• Protection domains define the set of names accessible to a given execution context
Spin

File System
- CPU scheduler
- Memory manager
- Network

File System
- Memory manager
- CPU scheduler

Hardware, managed by OS

IPC, Address Spaces, …
Spin Mechanisms for Events

• Spin extension model is based on events and handlers
  – Which provide for communication between the base and the extensions

• Events are routed by the Spin Dispatcher to handlers
  – Handlers are typically extension code called as a procedure by the dispatcher
    – One-to-one, one-to-many or many-to-one
      • All handlers registered to an event are invoked
        – Guards may be used to control which handler is used
Event example
PUTTING IT ALL TOGETHER
Default Core services in SPIN

- Memory management (of memory allocated to the extension)
  - Physical address
    - Allocate, deallocate, reclaim
  - Virtual address
    - Allocate, deallocate
  - Translation
    - Create/destroy AS, add/remove mapping

- Event handlers
  - Page fault, access fault, bad address

Figure 3: The interfaces for managing physical addresses, virtual addresses, and translations.
CPU Scheduling

• Spin abstraction: strand
  – Semantics defined by extension

• Event handlers
  – Block, unblock, checkpoint, resume

• Spin global scheduler
  – Interacts with extension threads package

---

INTERFACE Strand;

TYPE T <: REFANY; (* Strand.T is opaque *)

PROCEDURE Block(s:T);
(* Signal to a scheduler that s is not runnable. *)

PROCEDURE Unblock(s: T);
(* Signal to a scheduler that s is runnable. *)

PROCEDURE Checkpoint(s: T);
(* Signal that s is being descheduled and that it
  should save any processor state required for
  subsequent rescheduling. *)

PROCEDURE Resume(s: T);
(* Signal that s is being placed on a processor and
  that it should reestablish any state saved during
  a prior call to Checkpoint. *)

END Strand.

---

Figure 4: The Strand Interface. This interface describes the scheduling events affecting control flow that can be raised within the kernel. Application-specific schedulers and thread packages install handlers on these events, which are raised on behalf of particular strands. A trusted thread package and scheduler provide default implementations of these operations, and ensure that extensions do not install handlers on strands for which they do not possess a capability.
Experiments

• Don’t worry, I won’t go through them

• In the OS community, you have to demonstrate what you are proposing
  – They built SPIN, extensions and applications that use them
  – Focus on performance and size
    • Reasonable size, and substantial performance advantages even relative to a mature monolithic kernel
Conclusions

• Extensibility, protection and performance

• Extensibility and protection provided by language/compiler features and run-time checks
  - Instead of hardware address spaces
  - ...which gives us performance—no border crossing

• Who are we trusting? Consider application and Spin

• How does this compare to Exo-kernel?
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
How have such designs influenced current OS?

• Kernel modules
• Virtualization
• Containers
• Specialized OS
MORE DETAILS -- 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
  - Page 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's</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>unprot100</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
Extensible RPC

- Trusted version of lrpc called tlrpc which saves and restores callee-saved registers
Extensible Page-table Structures

- Inverted page tables
ExtensibleSchedulers

- **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.