Advanced Operating Systems (CS 202)

Operating System Structure

Jan, 14, 2019
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
• Lets go over conventional OS organizations
  - Largely representative of un*x, windows, OS/X, ...
Some Requirements

- **Need OS to virtualize resources**
  - Each program thinks it has them (via a nice abstraction)
  - The resources are actually shared efficiently and fairly
  - OS provides isolation
  - OS support communication

- **Need OS to mediate access to hardware/OS data structures**
  - Why?
Common model

• Commercial OS's largely use “Controlled Direct Execution”
  - User programs when scheduled run directly on the CPU
  - System calls for services; just another event
  - Directly use memory – its virtualized or partitioned somehow (with hardware help)

• OS is a sleeping beauty
  - Gets woken up by events
  - Enough for time sharing?
Events

• An event is an “unnatural” change in control flow
  – Events immediately stop current execution
  – Changes mode, context (machine state), or both

• The kernel defines a handler for each event type
  – Event handlers typically execute in kernel mode
  – The specific types of events are defined by the machine

• Once the system is booted, all entry to the kernel occurs as the result of an event
handleTimerInterrupt() {
  ...
}
handleDivideByZero() {
  ...
}
handleSystemCall() {
  ...
}
OS structure choices

• OS has to run in privileged mode sometimes
  - But does everything run in privileged mode?

• If the application could benefit from custom operation, can we provide that?
  - Does that come at the cost of performance or safety?

• Can the OS adapt operation to available resources?
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?
OS as library (DOS-like)

Applications

OS Services and Device drivers

Hardware, managed by OS
Monolithic Kernel

Applications

OS Services and Device drivers

Hardware, managed by OS

What is the difference?
Extensibility/customization

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

• Let's 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?
• Spin and exo-kernel papers have similar reasons
  - 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

Applications

File System
Memory manager
CPU scheduler

Micro-kernel

Hardware, managed by OS
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
  – Finally switch back to app
  – Each step is a border crossing
• 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; flush TLB, ...
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

Monolithic

safe

fast

micro kernel

extensible

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 border 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 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 border 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 object file contents and export names

- **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**
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

```plaintext
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?