Systems Design and Implementation

1.3 - Kernel and Operating System Interfaces

System Architecture Group, SS 2009
University of Karlsruhe
May 5, 2009

Jan Stoess
University of Karlsruhe
Overview

- Motivation
- User interfaces
- Kernel interfaces in monolithic OSes
  - Case study 1: Linux kernel modules
  - Case study 2: Windows WDM architecture
- Kernel interfaces in multi-server systems
  - Case study 3: The SawMill Multiserver Architecture
  - Case study 4: Virtualization interfaces
Motivation

- Operating systems run user programs
  - May request service
  - May need event notification
- Operating systems have different subsystems
  - e.g., paging call disk subsystem to swap
  - Need an interface

- Kernel Interfaces
  - Sharing/Transferring Data
  - Sharing/Transferring Code
  - Implications on programming model
    - E.g., C-Routines, RPC, …
User Interfaces

- Required functionality:
  - System Services (system calls)
    - read from disk, send over network, ...
    - Synchronous
    - Enhances privileges
  - Interface data:
    - Kernel service routine identifier
    - Parameters

- Notifications (signals)
  - Division by 0, Protection fault, completion of asynchronous service, ...
  - May be asynchronous
  - Must switch back to user privileges
  - Interface data
    - User-level callback handler identifier
    - Arguments
User Interfaces

- Required functionality:
  - Kernel-accessible user data
    - Statistics, configuration data (/proc), …
    - May be accessed asynchronously

- Constraints:
  - Safety:
    - User may not call arbitrary kernel routines
    - User may not arbitrarily switch to kernel privileges
    - User may not change arbitrary kernel data
User Interfaces

Solution:

- System services:
  - Leverage hardware primitives
    - Safe privilege change
    - Safe system service dispatching
  - Example: IA-32 \texttt{<int n>} instruction
    - Safe call to interrupt procedure
    - Loads kernel stack, changes FLAGS
    - Saves EIP, ESP, FLAGS on kernel stack
    - Transfers control to kernel code
      - Specified by interrupt number
      - Implies hardware privilege change
    - Return to user via \texttt{<iret>}
  - Hardware subject to change
    - Use trampoline page for kernel entry
    - Versatile interface
    - Can execute syscalls at user-level
User Interfaces

Solution:
- Notifications
  - Leverage MMU hardware
  - Kernel shares user address space
    - Can modify user-state
    - Can transfer control
  - But not vice-versa
- User-accessible kernel data
  - Dedicated shared pages
    - E.g. kernel interface page
  - Map to system calls
    - E.g., proc file system
Kernel Interfaces in monolithic OSes

- **Monolithic Kernel design**
  - Design principle: global, shared kernel
  - Programming language defines interface
    - Data interfacing through shared data
    - Control interfacing through direct control transfer
    - Compiler and linker determine and resolve addresses
Kernel Interfaces in monolithic OSes

- **Monolithic Kernel design**
  - Design principle: global, shared kernel
  - Programming language defines interface
  - Logical/Semantical separation of concerns
    - C-structs, extern functions, static functions
    - header files, source files
    - classes, members, namespaces, ...
  - **No** boundary protection
    - Software can easily cross semantic boundaries
    - arbitrary control transfers (e.g., using assembler)
    - arbitrary data access and modifications (e.g., using typecasts and pointers)
Kernel Interfaces in monolithic OSes

- Monolithic Kernel design
  - *No* privilege separation
    - All kernel subsystems can execute all processor instructions
    - All kernel subsystems can access all I/O hardware
  - Motivation: Performance
    - Crucial factor in OS
    - Protection domain switches are costly
      - Full address space switch (Pentium IV):
        - Changes *all* AS translations
        - Implies TLB flush
          (~ 500 cycles)
        - Implies (Virtual) Trace Cache flush
          (up to 4000 cycles)
        - + TLB replacement + Trace cache reloading
          (~ 5000 cycles)

Kernel Interfaces in monolithic OSes

- Monolithic Kernel design
  - No privilege separation
    - All kernel subsystems can execute all processor instructions
    - All kernel subsystems can access all I/O hardware
  - Motivation: Performance
    - Crucial factor in OS
    - Segmentation (partial AS switch)
      - Changes base offset, accessible limits within AS
      - Changes protection parameters
      - Implies segment register reloading (~300 cycles)
      - No TLB and TC flushing
      - But Restrictions on AS layout and size
  - Monolith lacks protection but retains performance
    - Direct calls, direct data accesses
    - Cross-component accesses and optimizations
    - Ad-hoc extensibility

Kernel Modules: Extensibility in Linux

- Linux is becoming more and more complex
  - vast amount of device drivers, network protocols, file systems
  - Linux should support crufty hardware
  - Support not always needed
- Need dynamic kernel extensibility
  - Loading (and unloading) kernel components on demand
    - E.g., device detection routine loads appropriate drivers
- Two subproblems:
  - Make component functionality available to kernel
  - Make kernel functionality available to component
Kernel Modules: Extensibility in Linux

- Linux kernel interface are defined by programming language ("C")
  - Data layout implicitly defined by compiler
    - structs, enums, arrays, (classes)
  - Global symbol namespace
    - Represents code and data
    - Compiler generates code and local symbols from source file (object files)
      - Relative addresses for internal references, placeholders for external references
      - References are stored within the object file itself (ELF format)
    - Linker resolves local symbols and computes global addresses to combine multiple object files
      - Resolves address collisions
      - Resolves external references
      - Must contain an ELF format parser
Kernel Modules: Extensibility in Linux

- Linux kernel interface are defined by programming language ("C")
  - Idea: Perform run-time linking of additional object files
    - Kernel modules are run-time linked kernel libraries
      - Images are relocatable
    - Store linking information within module
      - Special "__ksymtab" and "modinfo" section in ELF file
      - Contains text names for symbols
    - Store linking information within kernel symbol table
Kernel Modules: Extensibility in Linux

- **Loading kernel modules**
  - Modules are plain object files (.o)
  - User-space helper programs
    - `insmod`, `modprobe` and friends
    - ELF-load and parse modules
    - Pass special structure to kernel
  - Kernel
    - Relocates module image according to its dedicated virtual address space
    - Resolves external references based on kernel symbol table
    - Finds dependencies and loads more modules if required
    - Executes module init routine
      - Can register new driver, or functionality
Kernel Modules: Extensibility in Linux

Analysis:

- Modules serve the need:
  - Provide dynamic extensibility
  - Preserve the normal programming language based kernel interface

- But: Extensibility tied to the source code
  - Floating and volatile interface
  - Loading requires exact module/kernel match
Kernel Modules: Extensibility in Linux

Analysis:
- Kernel modules are not a protection mechanism
  - Modules link into the same address space
  - Can be abused (LKM root kits)
  - Raises dependability and reliability issues

Implementation
- Kernel depends on user-space programs (so what?)
- Module dependencies bear substantial complexity
  - Arbitrary <uses> and <depends> relations
  - circular dependencies
  - Inevitable with modularization?
Windows WDM driver architecture

- Windows is a proprietary, closed-source OS
- Still it…
  - Needs to support various (crufty) hardware devices
  - Needs to enable device manufacturers to develop their own driver software
  - Needs a standardized interface to let drivers interact with
    - I/O hardware
    - Other windows kernel subsystems
    - Applications

Windows WDM driver architecture

Basic Idea:

- Provide a special driver interface
- Use an abstract driver model as foundation
  - I/O request packets
  - WDM driver stack
  - Hardware abstraction layer
- Specify interaction as programming interface
  - WDM API defines standard methods, data structures, ...
  - Windows uses .inf files to install drivers

Source: M. Tsegaye and R. Foss *A comparison of the Linux and Windows device driver architectures*
Operating Systems Review 2004 2:38 p.8-33
Windows WDM driver architecture

Some details

- Driver objects
  - Filter, functional, bus drivers
  - Stackable
  - Specified functionality
    - init, addDevice, dispatch, unload

- Device objects
  - Represent a real HW device
  - Managed by a (set of) drivers
  - Can have a name
    - 128-bit device name space
  - Specify how I/O is transferred from user to kernel
    - Direct, buffered, pinned DMA

Windows WDM driver architecture

- Some details
  - I/O request packet (IRP)
    - represents an abstract I/O process data unit
    - Passed to driver stack by windows kernel subsystem
    - Percolates through the specified dispatch routines
  
- Driver programmer
  - implements driver components
  - links them together to form a stack
  - provides device names

- Application programmers
  - Can perform I/O based on device name.

Windows WDM driver architecture

- Driver interface details
  - API approach
  - Programmer relies on specified C-functions and data structures

```c
#include <ntddk.h>

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    ...
    return STATUS_SUCCESS;
}
```

- Windows provides a build utility (DDK)

```
TARGETNAME = mydriver
tARGETPATH = obj
TARGETTYPE = DRIVER
INCLUDES = %BUILD%\inc
LIBS = %BUILD%\lib
SOURCES = mydriver.c
```

Source: M. Tsegaye and R. Foss  *A comparison of the Linux and Windows device driver architectures* 
Operating Systems Review 2004 2:38 p.8-33
Windows WDM driver architecture

Analysis:

- WDM
  - provides dynamic extensibility for device drivers
  - API based kernel interface
- Extensibility *not* tied to the source code
  - Fixed interface
  - Build process can produce drivers for different Windows versions
- Interface specialized to device drivers
  - Does not provide generic module/subsystem extensibility
Windows WDM driver architecture

- **Analysis:**
  - WDM does not provide protection
    - Drivers link into the same address space
    - Raises dependability and reliability issues
    - Drivers are known to be highly error-prone*  
  
- **Implementation**
  - Data-centric model
    - I/O Request packets and dispatchers
  - Simple component dependencies
    - Stack of dispatchers

Kernel Interfaces in Multi-server Systems

- Multi-server kernel (system) design
  - Privilege separation through address-space protection
    - $\mu$-Kernel is privileged but limited in functionality
    - Other kernel subsystems are “user programs”
      - Can not execute privileged instructions
      - Can not access arbitrary memory locations
      - Can not access arbitrary I/O hardware

- Motivation: Protection
  - Premise for security, reliability, dependability, …
  - Crucial factor in OS
  - But protection domain switches are costly
  - Multi-server system trades off protection against performance
  - Key problem: Keep good performance
Kernel Interfaces in Multi-server Systems

- Multi-server interfaces
  - Kernel subsystems are “user programs”
    - Normal user interface for μ-Kernel services
    - Direct addressing and data sharing between other subsystems unfeasible
  - μ-kernel must cater for subsystem interaction
  - Should be generic and versatile
    - Support different subsystems
      - resource managers, schedulers, pagers, drivers, UI, ...
    - Support different programming models
      - Different manufacturers, compilers, languages, black-box binaries, ...
    - Support different interaction scenarios
      - Service requests and returns
      - Data sharing
      - Notifications, callbacks, exceptions, ...
Kernel Interfaces in Multi-server Systems

- Multi-server interfaces
  - (L4) Idea: provide simple and generic IPC
    - Used by kernel subsystems
    - Used by user programs
  - Develop specializations on top
    - Subsystem-specific interaction
    - Programming models (APIs, C-like function calls, …)
    - Data sharing (shared memory, request buffers, …)
    - Naming and addressing schemes
Kernel Interfaces in Multi-server Systems

- But how to define interfaces?
  - Subsystem-specific interfaces
  - Programming models (APIs, C-like function calls, …)
  - Data sharing (shared memory, request buffers, …)
  - Naming and addressing schemes

- Idea: Leverage work from distributed systems
  - Same scenario: distributed components + interaction
  - E.g., Remote procedure call model
    - Client/Server model
    - Need transparent, procedure-call like semantics
      - Client calls server for service
      - Server returns after processing
    - Provide remote procedure call (RPC)
      - Synchronous communication
      - Can pass and return arguments
Kernel Interfaces in Multi-server Systems

But how to define interfaces?
- Subsystem-specific interfaces
- Programming models (APIs, C-like function calls, …)
- Data sharing (shared memory, request buffers, …)
- Naming and addressing schemes

Idea: Leverage work from distributed systems
- Same scenario: distributed components + interaction
- E.g., Remote procedure call model

Problems:
- Calling convention
  - No shared data
  - Pointers? References?
- Transparency
  - Should “feel” like normal call/ret
  - Latency? IPC Errors?
Kernel Interfaces in Multi-server Systems

- Remote procedure call approaches
  - Client and server stubs
    - Transform call/ret semantics into communication
      - Parameter marshaling/unmarshaling
      - Procedure multiplexing/demultiplexing
      - Message and data layout definition
      - Leverages system communication primitives
  - Steps:
    - Client procedure-calls client stub
    - Client stub
      - marshals parameters
      - builds message
      - calls kernel to send message to server
    - Server stub decodes message
      - dispatches the correct procedure (if needed)
      - unmarshals parameters
      - calls corresponding server-side procedure
    - Server processes the request and returns to the server stub
Kernel Interfaces in Multi-server Systems

- Special considerations for node-local (multi-server) RPC
  - Communication is more efficient, thus stub code efficiency has more impact
  - Same hardware: same endianess, bit width, float precision, …
  - Same μ-kernel, can rely on its data types, interfaces etc.
  - Simplifies/speeds up stub code
Kernel Interfaces in Multi-server Systems

- Remote procedure call
  - Writing stubs is tedious
  - Idea: Automate stub code generation
- Interface definition languages
  - Language that specifies interfaces
    - Remote method definition
    - Special data types for argument passing
  - Compiler generates interface stubs
    - Client stub
    - Server stub
    - Server skeleton (basic dispatcher)
  - Examples: Flick, Corba IDL, DCOM
  - See lab lecture: using IDL^4

```c
void foo_bar(...)
{
  asm volatile (
    "push %%ebp" "push %%ecx" "xor %%eax, %%eax": "d" ((int)a): "cc", "memory"
  );

#include "foo_client.h"

#define IDL4_PUBLISH_FOO_BAR(func) {
  idl4_server_environment _env; 
  func(_par._in._caller, &_env)
  __asm__ __volatile__( 
    "xor %%eax, %%eax" "ret" : "S" (_par._in._caller)
  );

#include "foo_server.h"

server.c
```
The SawMill Multiserver Architecture

- The SawMill Approach
  - Complexity of OS increases
  - Need specialized OS personalities for different scenarios
  - Need a development path to build such specialized operating systems

The SawMill Multiserver Architecture

The SawMill Approach

- **Idea:** *Decompose* existing operating systems for flexibly reusable components
  - Extend existing OS with functionality
  - Customize existing OS: strip them down for application requirements

- **The SawMill approach consists of**
  - An architecture to build systems
  - A set of protocol design guidelines to solve multi-server problems

The SawMill Multiserver Architecture

**Example "SawMill" Multi-Server Linux:**

- (1) isolate Linux services from each other;
- (2) improve them one by one:
  - VM, scheduling, security (denial of service), reliability, SMP, large memory, mmap, async io, select, large files
- Extend Linux, add value:
  - New security policies, …
- Customize Linux for special devices.

![Diagram of SawMill Multiserver Architecture](image)
The SawMill Multiserver Architecture

- SawMill design considerations
  - The multiserver OS must provide
    - Protection
      - Protect execution integrity of servers
      - Protect data integrity/confidentiality of user data
    - Coherent semantics
      - Obtain and enforce system policies
      - Obey atomicity requirements
    - Performance (efficient services)
      - Protection implies more frequent IPC
        - IPC replaces procedure calls
        - Additional IPCs required for consistency, synchronization, resource management, security policies, …
      - Protection implies more complex IPC
        - Parameter transfer
        - Parameter marshaling
        - See previous slides
The SawMill Multiserver Architecture

- SawMill architecture
  - Three types of components
    - System servers
      - Main OS functionality
      - File server, network server, …
    - Resource servers
      - Manage core resources
      - Distributed among system servers
      - Memory, IRQs, security abstractions, …
    - Ubiquitous services
      - “Libraries” that augment servers
      - Multiserver-aware management
      - Synchronization, ACL, Naming, IPC primitives, …
  - Example: Virtual memory
    - VM system server exports a *dataspace*
    - Memory server provides core memory
    - Ubiquitous VM service handles indirection between dataspace and core memory
The SawMill Multiserver Architecture

- SawMill protocols
  - Goal: minimize IPC frequency and overhead
  - Design principles:
    - Make direct calls to processing servers
      - Let clients communicate directly with subsequent servers
    - Partition control data
      - Distribute control data among involved servers
      - Use caching in servers if possible
      - Minimize synchronization
        - Minimize writes
        - Weaken consistency models
        - Use “Master copy” schemes
    - Heavily use data sharing
The SawMill Multiserver Architecture

- Envisage direct calls

The SawMill Multiserver Architecture

- Partition control data
The SawMill Multiserver Architecture

- Share user data

Could use paging to share user data but
  - alignment problems
  - decomposition problems

Sources:
The SawMill Multiserver Architecture

- Analysis
  - SawMill
    - Envisages customized, modular OS personalities
    - Uses a *decomposition* approach for reuse
  - Presents a basic architecture
    - μ-kernel based client/server architecture
    - Servers, ubiquitous services, core resource managers
  - And a set of protocol guidelines
    - Make direct calls to processing servers
    - Partition control data
    - Share user data
    - Used to design and implement components and interfaces
The SawMill Multiserver Architecture

- Analysis
  - Problems
    - Decomposition is hard
    - Stripping down is hard
      - SawMill Linux has a huge code base
      - SawMill must maintain/fight against legacy Linux semantics
      - Linux was never designed to deal with multi-server problems
    - Partitioning control data is complicated
    - Sharing user data is complicated
      - Especially together with legacy semantics
      - E.g., how to partition entangled control/user data (skbuffers)?
      - How to share unaligned data?
Virtualization interfaces

- **Background**
  - Complexity of OS increases
  - Want to improve or introduce new OS functionality
    - Effective time sharing (aka server consolidation)
    - Simultaneous support of multiple OS APIs
    - Transparent migration
    - Security services
  - Monolithic OS design has serious limits
    - Complex, entangled, unreliable, insecure, …
    - Hard to customize, hard to extend, hard to decompose
Virtualization interfaces

- Problem: Legacy support
  - New OS must support old programs
  - API support not sufficient
    - Want to support old OS functionality as well
    - Many applications are tailored to specific OS versions
    - Need a development path to incorporate new and keep old functionality at the same time
Virtualization interfaces

- Idea: Virtualization
  - Provide hardware interface
  - But transparently change semantics
- Interface constituted by hardware specification
  - Fixed and well-designed interface
  - Already used by guest OS, no porting effort needed
- Virtualization only changes semantics
  - Restrict side effects to virtual machine and dedicated hardware
  - Keeps illusion of real hardware
Virtualization interfaces

Examples:

- Interrupts
  - Guest executes `<clear IF>`
  - Hypervisor intercepts instruction
  - Monitor/Emulator disables preemption of Guest OS

- Page table modifications
  - Guest inserts page table entry
  - Hypervisor intercepts modification
  - Monitor/emulator modifies physical mapping if necessary
Virtualization interfaces

- Virtualization provides stacked OS model
  - Guest OS contains applications and (unprivileged) OS services
  - Hypervisor/host OS contains privileged OS services and emulation

- Additional OS services can be designed freely
  - No interface requirements
  - Multi-server components
  - Leverage host OS
  - Use specialized virtual machines
Virtualization interfaces

Analysis
- Interface defined by hardware
  - Fixed and well-designed interface
  - Already used, no porting effort needed
- Virtualization changes semantics
  - Semantics are not specified
  - Transparency introduces overhead
- Hardware Interface may be inappropriate
  - Example: Disk I/O
    - Guest performs write to device
    - File access? Swapping?
  - Example: Network I/O
    - Guest calls virtual NIC to send buffer
    - Virtual NIC must decode packets again
- Virtualization only provides legacy
  - It does not address the design of new OS functionality
  - It does not address the design of new or improved interfaces
Thursday

- Maifeiertag