The Operating System

‣ The operating system is
• a C/assembly program
• implements a set of abstractions for applications
• it encapsulates the implementation of these abstractions, including hardware

The Operating System’s Address Space
• a part of every application’s page table is reserved for the OS
• all code and data of OS is part of every page table (exact copies)
• and so the operating system is part of every application’s address space

Dual Protection Domains
• each address space splits into application and system protection domain
• CPU can run in one of two modes: user and kernel
• when in user mode, the OS part of virtual memory is inaccessible
• when in kernel mode, all of virtual memory is accessible

Translation

 implementing the system abstractions

Operating System

OS context switches to server, and unblocks server

Implementing Hardware Encapsulation

Hardware
• mode register (user or kernel)
• certain instructions only legal in kernel mode
• page table entries have protection flag (user or kernel)
• attempting to access a kernel page while in user mode causes fault
• special instructions for switching between user and kernel modes

Translate

Protected Procedure Call

Switching from User Mode to Kernel Mode must be protected
• OS has a fixed set of “entry points”, its public API
• an application can call any of these entry points, but no others
• when in kernel mode the application can access anything
• so, an application can only switch to kernel mode after calling entry point
• but, even entry points are in inaccessible memory

Implementing Protected Calls
• OS boot sets up entry-point jump table in kernel memory
• jump table is indexed by system call number and stores procedure address
• system call instruction change mode and jumps through jump table
• in IA32 this instruction is called “int 80” (i.e., interrupt number 0x80)
• this works like an IO-Controller interrupt, it transfers control to interrupt-handler
• but this also switches the processor into kernel mode (all interrupts do this)

Protected call instruction, assuming syscall number is in r0

Implementing the System Abstractions

‣ We’ve got some cool abstractions
• virtual processors (threads)
• virtual memory
• processes
• authenticated users
• file systems
• inter-process and network communication

What properties do we want from their implementation
• encapsulation of implementation by an interface
• failure and security isolation
• programming-language heterogeneity

We’ve got a problem ...

Readings for Next Two Lectures

‣ Text
• Exceptional Control Flow: Processes, System Call Error Handling
• VM as a Tool for Memory Protection
• 1st edition: 8.2, 8.3, 10.5

Implementing Hardware Encapsulation

Goal
• define a set of interfaces (APIs) whose implementations are protected
• implementation code and data can only be accessed through interface

Obstacle
• can not use language protection without excluding languages like C

Use Hardware for Protection
• virtual memory already provides a way to protect memory
• data in one address space can not even be named by thread in another
• so, we’ve got the protected implementation part
• we’ll need to add the interface part

Encapsulation
• define a set of interfaces (APIs) whose implementations are protected
• translation assumes private memory and processor
• translation assumes private memory and processor

Threads
• an abstraction implemented by software to manage asynchrony and concurrency
• provides the illusion of single processor to applications
• differs from processor in that it can be stopped and restarted

Virtual Memory
• an abstraction implemented by software and hardware
• allows OS to export a public interface and to encapsulate (hide) the implementation