CSCI 3753 Operating Systems

Design Issues

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 01/17/13

System Boot

- Operating system manages all programs: where they are stored, when to run them, etc.
- But how does the system know where the operating system is or how to load the kernel?
- *Booting* the system: Procedure od starting a computer by loading the operating system
- Bootstrap program (also called bootstrap loader)
  - Locates the kernel, loads it into main memory, and starts its execution
  - Typically a 2-step process: a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel

System Boot

- When CPU receives a reset event (powered/reboot)
  - IR is loaded with a predefined memory location that contains the initial bootstrap program
  - In ROM: needs no initialization and cannot easily be infected
- Bootstrap program
  - Run diagnostics to determine the state of the machine
  - Initialize registers, main memory, device controllers, etc.
  - Start OS
- Smaller systems: store entire OS in ROM or EPROM (firmware)
- Large systems: bootstrap loader in firmware; OS in disk

Dual Mode Operation

- Processor mode: distinguish between execution on behalf of an OS and execution on behalf of a user.
- Kernel: trusted software module that supports the correct operation of all other software; core part of OS.
- OS interface: how a user interacts with an OS to request OS services.

Protecting the OS via a Mode Bit

- In early CPUs, there was no way to differentiate between the OS and applications:
  - Want to protect OS from being overwritten by app’s
  - Want to prevent applications from executing certain privileged instructions, like resetting the time slice register, resetting the interrupt vector, etc.
- Processors include a hardware *mode* bit that identifies whether the system is in *user* mode or *supervisor/kernel* mode
  - Requires extra support from the CPU hardware for this OS feature

Processor mode

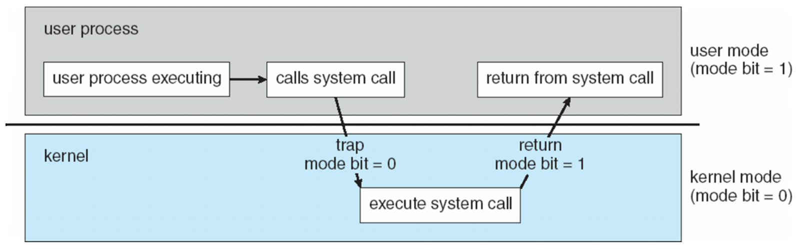
- Supervisor mode or user mode: determined by a mode bit.
  - Supervisor mode (mode bit = 0): processor can execute every instruction available in the instruction set.
  - User mode (mode bit = 1): processor can execute only a subset of instructions available in the instruction set.
- Privileged (protected) instructions:
  - Instructions that can be executed only in supervisor mode.
  - I/O instructions
  - Protection and security: privileged load and store instructions
- Used to define two classes of memory space: user space and system space.

User / Kernel Modes

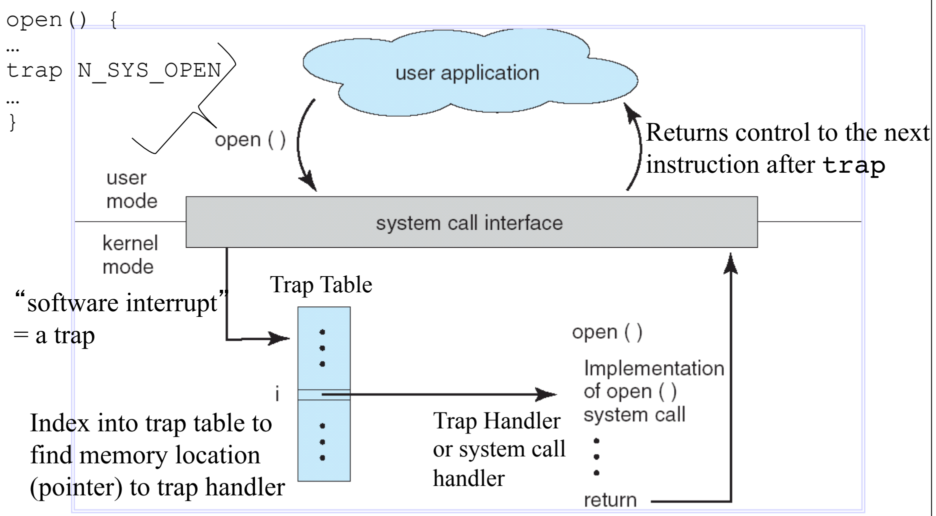
- Most modern CPU's support this mode bit
  - Intel x86 CPUs have four modes or rings, but not all are necessarily used by the OS
    - Example: an OS like Linux or Windows might set itself as ring/mode 0 (highest privilege, can execute any CPU instruction and access any location in memory), while all applications run in ring/mode 3 (lowest privilege, if an app attempts to run a privileged instruction or access restricted memory, it will generate a fault, invoking the higher privileged OS). Rings 1 and 2 unused.
    - Example: a virtual machine monitor (VMM) such as VMWare might set itself as ring 0, while a guest OS VM might run as ring 1 or 2, and user applications would run as ring 3
  - embedded microcontrollers typically don't have a mode bit

System Calls: How Apps and the OS Communicate

- The `trap` instruction is used to switch from user to supervisor mode, thereby entering the OS
  - `trap` sets the mode bit to 0
  - Also called `syscall` in MIPS
  - mode bit set back to 1 on return
- Any instruction that invokes `trap` is called *a system call*
  - There are many different classes of system calls



API – System Call – OS Relationship


open() {
...
trap N_SYS_OPEN
...
}
user application
open ( )
Returns control to the next instruction after trap
user mode
system call interface
kernel mode
"software interrupt" = a trap
Trap Table
open ( )
Implementation of open ( ) system call
i
Index into trap table to find memory location (pointer) to trap handler
Trap Handler or system call handler
return

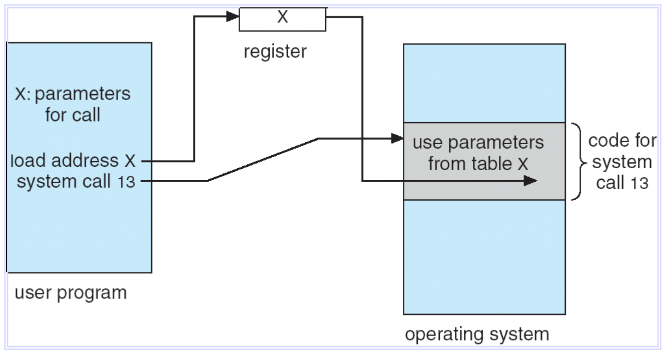
Trap Table

- The process of indexing into the trap table to jump to the trap handler routine is also called dispatching
- The trap table is also called a *jump table* or a *branch table*
- "A trap is a software interrupt"
- Trap handler (or system call handler) performs the specific processing desired by the system call/trap

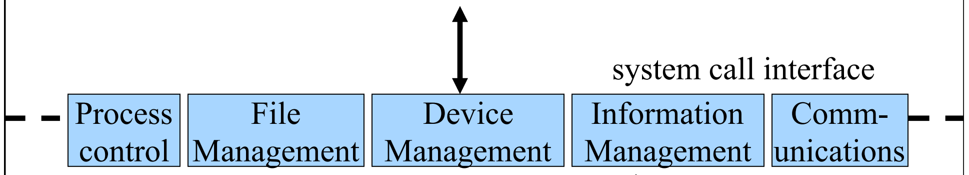
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - Parameters stored in a *block* in memory, and block address passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Classes of System Calls Invoked by `trap`



- end, abort
- load, execute
- fork, create, terminate
- get attributes, set
- wait for time
- wait event, signal event
- allocate memory, free

- create, delete
- open, close
- read, write, reposition
- get attributes, set

- request device, release
- read, write, reposition
- get attributes, set
- logically attach or detach devices

Note Similarity

- get time/date, set
- get system data, set
- get process, file, or device attributes, set

- create connection, delete
- send messages, receive
- transfer status info
- attach remote devices, detach

Examples of Exceptions in x86 Systems

Class	Cause	Examples	Return behavior
Trap	Intentional exception, i.e. "software interrupt"	System calls	always returns to next instruction, synchronous
Fault	Potentially recoverable error	Divide by 0, stack overflow, invalid opcode, page fault	might return to current instruction, sync
Abort	nonrecover-able error	Hardware bus failure	never returns, sync
Interrupt	signal from I/O device	Disk read finished	always returns to next instruction, async

Course Outline

- Device Management (Chapter 13)
  - Managing I/O devices
- Process Management (Chapters 3 – 7)
  - Processes and threads
  - Process synchronization
  - CPU scheduling
  - Deadlocks
- Memory Management (Chapters 8 – 9)
  - Primary memory management
  - Virtual memory

- Storage Management (Chapters 10 – 12)
  - Mass-storage structure
  - File system interface and implementation
- Protection and security(Chapters 14 – 15)
- If time permits …
  - Virtual machines and distributed systems