Today

- How my device becomes useful for the user?
- HW-OS interface
  - Device controller
  - Device driver
  - Interrupts
- OS-App interface
  - System Call
  - Privilege Levels
  - Exceptions
A modern computer system

- CPU
- Disk controller
- USB controller
- Graphics adapter
- Memory
- Disks
- Keyboard
- Mouse
- Printer
- Monitor
HW-OS interface
HW-OS interface

• Device Controller:
  • A set of chips on a plug-in board.
  • It has local buffer storage and/or a set of special purpose registers.
  • Responsible for moving data between device and registers/buffer.
  • Responsible for making data available to the device driver.
HW-OS interface

• Device Driver:
  • Device-specific software
  • Part of the OS.
  • Communicates with the device controller.
  • Presents a uniform interface to the rest of the OS.
HW-OS interface

- Driver to Controller:
  - Programmed I/O
    - CPU has dedicated, special instructions
    - CPU has additional input/output wires (I/O bus)
    - Instruction specifies device and operation
  - Memory-mapped I/O
    - Device communication goes over the memory bus
    - Reads/Writes to special addresses are converted into I/O operations by dedicated device hardware
    - Each device appears as if it is part of the memory address space
- Memory-mapped I/O is the predominant device interfacing technique in use
HW-OS interface

- controller to driver:
  - Polling
    - CPU constantly checks controller for new data
    - What's wrong with this scheme?
    - How bad is it?
  - Interrupt driven I/O
    - Controller alerts CPU for an event
    - CPU can do other things while it waits
Interrupt Driven I/O

- An interrupt controller mediates between competing devices
- Raises an interrupt flag to get the CPU’s attention
- Identifies the interrupting device
- Can disable (aka mask) interrupts if the CPU so desires
Interrupt Management

- Interrupt controllers manage interrupts
  - Interrupt descriptors describe device
- Maskable/unmaskable interrupts
  - Why?
  - Examples?
- Interrupt priority
  - A priority selector circuit examines all interrupting devices, reports highest level to the CPU
  - Can optionally remap priority levels
- How are masks and priorities set?
Interrupt-driven I/O summary

- Normal interrupt-driven operation with memory-mapped I/O:
  1. Application issues “read()” syscall
  2. OS (driver) writes to device controller register
     - How?
  3. OS finds something useful to do with the CPU
     - For example?
  4. Meanwhile, the device asynchronously performs the operation.
  5. Later, the device controller interrupts the CPU.
  6. The CPU stops the current computation.
  7. The CPU transfers the execution to the service routine (in the device driver).
  8. The interrupt service routine executes.
  9. On completion, the CPU resumes the interrupted computation.
- Can this be better?
Direct Memory Access (DMA)

- Transfer data **directly** between **device** and **memory**
  - No CPU intervention required for moving bits

- Device raises interrupts solely when the block transfer is complete

- Critical for high-performance devices
  - Examples?
OS-App interface

device

device controller

CPU

device driver

OS

Application

memory
OS-App interface

• Application to Driver:
  • System Calls
  • Like calling a routine of the OS.

• Driver to Application:
  • Pass data from OS memory space to application memory space.
System Calls

• Why do we need System Calls?
  • Provide system services to applications
    – Access hardware
    – Communicate
    – Request resources
    – Examples?

• Need access control
  – e.g. HALT, change MMU settings, set clock, reset devices, manipulate device settings, …
Privilege Levels

• How the CPU knows if an application has the right to execute a privileged command?
  • Use a “privilege mode” bit in the processor
  • 0 = Untrusted = user, 1 = Trusted = OS
Privilege Mode

• Privilege mode bit indicates if the current program can perform privileged operations
  • On system startup, privilege mode is set to 1, and the processor jumps to a well-known address
  • The operating system (OS) boot code resides at this address
  • The OS sets up the devices, initializes the MMU, loads applications, and resets the privilege bit before invoking the application

• Applications must transfer control back to OS for privileged operations
• Back to System Calls ...
Sample System Calls

- **Print** character to screen
  - Needs to multiplex the shared screen resource between multiple applications

- **Send** a packet on the network
  - Needs to manipulate the internals of a device whose hardware interface is unsafe

- **Allocate** a page
  - Needs to update page tables & MMU
System Calls

• A system call is a **controlled transfer** of execution from **unprivileged** code to the **OS**
  • A potential alternative is to make OS code read-only, and allow applications to just jump to the desired system call routine. Why is this a bad idea?

• A **SYSCALL** instruction transfers control to a system call handler at a fixed address (software interrupt).
SYSCALL instruction

- SYSCALL instruction does an atomic jump to a controlled location
  - Switches the SP to the kernel stack
  - Saves the syscall number
  - Saves arguments
  - Saves the old (user) SP, PC (next command), privilege mode
  - Sets the new privilege mode to 1
  - Sets the new PC to the kernel syscall handler

- Kernel system call handler carries out the desired system call
  - Saves callee-save registers
  - Examines the syscall number
  - Checks arguments for sanity
  - Performs operation
  - Stores result in v0
  - Restores callee-save registers
  - Performs a “return from syscall” instruction, which restores the privilege mode, SP and PC
Libraries and Wrappers

• Compilers do not emit SYSCALL instructions
  • They do not know the interface exposed by the OS

• Instead, applications are compiled with standard libraries, which provide “syscall wrappers”
  • printf() -> write(); malloc() -> sbrk(); recv(); open(); close(); …

• Wrappers are:
  • written in assembler,
  • internally issue a SYSCALL instruction,
  • pass arguments to kernel,
  • pass result back to calling application
Typical Process Layout

- Libraries provide the glue between user processes and the OS
  - libc linked in with all C programs
  - Provides printf, malloc, and a whole slew of other routines necessary for programs
Full System Layout

• The **OS** is omnipresent and steps in where necessary to aid application execution
  • Typically resides in high memory

• When an **application** needs to perform a privileged operation, it needs to **invoke the OS**
Exceptional Situations

- **System calls** are control transfers to the OS, performed under the control of the user application.

- Sometimes, need to transfer control to the OS at a time when the user program least expects it:
  - Division by zero,
  - Alert from the power supply that electricity is about to go out,
  - Alert from the network device that a packet just arrived,
  - Clock notifying the processor that the clock just ticked,

- Some of these causes for interruption of execution have nothing to do with the user application.

- Need a (slightly) different mechanism, that allows resuming the user application.
Interrupts & Exceptions

- On an interrupt or exception
  - Switches the sp to the kernel stack
  - Saves the old (user) SP value
  - Saves the old (user) PC value
  - Saves the old privilege mode
  - **Saves cause of the interrupt/exception**
  - Sets the new privilege mode to 1
  - Sets the new PC to the kernel interrupt/exception handler

- Kernel **interrupt/exception handler** handles the event
  - Saves all registers
  - Examines the **cause**
  - Performs operation required
  - **Restores all registers**
  - Performs a “**return from interrupt**” instruction, which restores the privilege mode, SP and PC
Syscall vs. Interrupt

• The differences lie in how they are initiated, and how much state needs to be saved and restored

• Syscall requires much less state saving
  • Caller-save registers are already saved by the application

• Interrupts typically require saving and restoring the full state of the processor
  • Why?
Terminology

- **Trap**
  - Any kind of a *control transfer* to the OS

- **Syscall**
  - *Synchronous, program-initiated control transfer* from user to the OS to obtain *service* from the OS
  - e.g. SYSCALL

- **Exception**
  - *Asynchronous, program-initiated control transfer* from user to the OS in response to an *exceptional event*
  - e.g. Divide by zero, segmentation fault

- **Interrupt**
  - *Asynchronous, device-initiated control transfer* from device to the OS
  - e.g. Clock tick, network packet
Today

- How my device becomes useful for the user?
- HW-OS interface
  - Device controller
  - Device driver
  - Interrupts
- OS-App interface
  - System Call
  - Privilege Levels
  - Exceptions
Activity

- If we want to add the System Call “machine_active_time”; how many hours the machine is ON, which SW components do we have to add and where?