OS Structure, Processes & Process Management

Recap
- OS as a complex system
- OS functions
  - Manager
  - Poet
  - Coordinator
- History: change driven by technology
  - HW expensive, humans cheap
  - HW cheap, humans expensive
  - HW "free", networks everywhere

In this Unit
- How the OS runs business
  - supervisor mode vs user mode
- How the OS is invoked
  - interrupts, traps, and exceptions
  - the boot process
- OS Structure
  - Monolithic, microkernel, virtual machine
- The Anatomy of a Process

Dual-mode operation
- reading and writing memory, managing resources, accessing I/O... would you trust this to HIM?
- Solution: dual mode operation
  - "User Mode": can only perform a restricted set of operation (applications)
  - "Supervisor Mode": can do anything! (Kernel)
- How would you implement dual mode operation?
Crossing the line

- Three ways to go from user to supervisor mode:
  - **Interrupts**: HW device requests OS service—asynchronous
  - **Traps**: user program requests OS service (system call)—synchronous
  - **Exceptions**: user program acts silly (e.g. divide by zero)—synchronous

On Interrupts

- Hardware calls OS at a pre-specified location
- OS saves state of the user program
- OS identifies the device and cause of interrupt (interrupt-controller)
- Responds to the interrupt
- Operating system restores state of the user program (if applicable) or some other user program
- Execute an RTI instruction to return to the user program
- User program continues where it was interrupted.
  
  **Key Fact**: None of this is visible to the user program

On Exceptions

- Hardware calls OS at a pre-specified location
- OS identifies cause of exception (e.g. divide by 0)
- If user program has exception handling specified, then OS adjusts user program state so that it calls its handler
- Execute RTI to return to the user program
- If user program did not have a specified handler, then OS kills it and runs some other user program, as available

  **Key Fact**: Effects of exceptions are visible to user programs and cause abnormal execution flow

On System Calls

- User program executes a trap instruction (system call)
- Hardware calls OS at a pre-specified location
- OS identifies the required service and parameters (e.g. open(filename, O_RDONLY))
- OS executes the required service
- OS sets a register to contain the result of call
- OS executes RTI to return to the user program
- User program receives the result and continues

  **Key Fact**: To the user program, it appears as a function call executed under program control
Summary

- An OS is just a program:
  - It has a main() function, called only once (during boot)
  - It consumes resources (such as memory), can do silly things (like generating an exception), etc.
- But it is a very strange program:
  - It is "entered" from different locations in response to external events
  - It does not have a single thread of control and can be invoked simultaneously by two different events (e.g. system call & an interrupt)
  - It is not supposed to terminate
  - It can execute any instruction in the machine

Internal OS Structure

- OS provides a more convenient interface than the raw hardware interface
- The structure of the OS affects both the abstractions provided by the OS and their implementation

Monolithic structure (e.g. Unix)

Everything

- Hardware

Monolithic structure (e.g. Unix)

Advantages?

Disadvantages?
Accessing an OS Service: System call

- **Abstraction:** Procedure call
- **Implementation:**
  - Application makes library procedure call `read(fd, buffer, length)`
  - Library converts to system call
  - Writes call number in well-known register
  - Writes arguments in other register, or stack
  - Trap (!)
- Questions:
  - Trap jumps to OS handler which reads arguments, services call, and returns
  - If OS is written in C/C++, handler needs a stack

**Questions:**
- Is it OK to use the application stack?
- What happens on a trap?
- What should the handler stack look like on call?
- What if an interrupt arrives?

Layered OS

- Each system service is a layer
- Each layer defines and implements an abstraction for the layer above it
- Layers in effect "virtualize" the layer below
- Advantages? Disadvantages?

Microkernels
(e.g. Mach, MacOSX)

- Minimize kernel services, implement remaining OS modules as (protected) user level processes
- Client/Server interaction, mediated by uniform message passing interface
- Examples: Hydra (1970), MACH, Chorus, NT
- Advantages? Disadvantages?
Awakening the Giant: System boot

What happens when you turn the power on?

1. CPU loads boot program from ROM
2. Boot program (BIOS in PCs):
   - Examines/checks machine configuration (number of CPU's, how much memory, number & type of HW devices, etc.)
   - Builds a configuration structure describing the hardware
   - Loads the bootloader
     - from "well-known" memory location (e.g. 1st sector of hard disk)
     - to "well known" memory location
   - Jumps to bootloader code at "well-known" entry point

Awakening the Giant: The bootloader

3. Bootloader:
   - Initializes stuff
     - SP = initial stack at well-known location
     - Read OS from disk, jump to well-known entry point

4. OS initialization
   - initialize kernel data structures
   - initialize state of HW devices
   - initialize VM system
   - creates a number of processes to start operations (e.g. daemons tty in Unix, windowing system in NT); soon we'll see how...

A sleepy Giant...

5. Once the OS is initialized
   - Run user programs if available; else run low-priority user-level idle-process
   - In the idle process
     - Infinite loop (Unix)
     - system management and profiling
     - halt processor and enter low-power mode (notebooks)
     - compute some function (DEC's VAX VMS computed π
   - OS wakes up on
     - Interrupts from HW devices
     - traps from user programs
     - exceptions from user programs