Chapter 6 Direct Execution

- The OS must share the physical CPU among many jobs at what seems like the same time to virtualize.
- Time Sharing the CPU runs one process for a while then another.
- Challenges for virtualizing in this way:
    - Performance: don't want to add significant overhead.
    - Control: Running the process efficiently while keeping control of the CPU.
        - If control is lost a process could run forever.
        - Access information it should not be able to.

6.1 Limited Direct Execution

- Run a program directly on the CPU.
- When the OS wants to start a process running it does the following:
    - It creates a process entry for it in the process list.
    - Allocates some memory for it
    - Loads the program into memmory
    - Locates the entry point
        - the main() routine or something similar.
    - Jumps to the entry point and starts execution.
- There are problems with this approach:
    - How can the CPU make sure that the program doesn't do anything we don't want it to.
    - How does the operating system stop it from running and switch it to another process.

6.2 Restricted Operations

- Direct execution is fast but what if the process tries running a restricted operation.
  - One answer is to let a process do what it wants.
    - Prevents security checks.
  - Introduce a new processer mode called user mode.
    - A running mode that restricts process execution.
  - Kernel mode is the mode the operating system runs in.
    - no restrictions on execution.
  - System calls allow interaction between user mode and kernel mode.
- System Calls:
  - Most operating systems provide a few hundred system calls.
    - POSIX standard
  - A program must execute a special trap instruction to execute a system call
    - jumps to the kernel and raises the privilege level to kernel mode.
    - The OS can perform whatever operation is needed.
  - When done the sytem performs a return from trap call.
    - Returns to user mode.
- When executing a trap instruction the hardware needs to be careful.
  - Needs to save enough of the caller's registers to return correctly.
  - On x86 the OS pushes the program counter, flags and other registers on the the stack.
  - When returning from trap the values are popped off the stack.
- How the trap knows what code to run in the OS
  - The caller can't specify which address to jump to.
  - The kernel needs to control what is executed in a trap.
  - The Kernel sets up a trap table at boot time.
    - The machine boots in kernel mode.

            - allows the configuring of hardware as needed
        - Tells the hardware what to do for events.
        - The kernel tells the hardware the location of the trap handlers.
            - persists until reboot.
- To specify a system call a system call number is assigned.
    - The user code is responsible for placing the correct syscall number in the proper register.
    - Protects from user programs specifying addresses to jump execution to.
- Two phases of the LDE (limited direct execution) protocol
    - First: Kernel initializes the trap table.
        - CPU remembers its location for later use.
        - Done through a privileged instruction.
    - Second: The Kernel sets up things before using a return from trap instruction
        - Switches the CPU to user mode.
        - Begins running the process
        - When the process issues a syscall it traps back to the OS.

6.3 Switching Between Processes

- When a process is running inside the CPU the OS is not running.
    - When the OS isn't running it can't do anything.
- **Cooperative Approach: Wait for System Calls**
    - The OS trusts the processes of the system to behave reasonably.
    - Long running processes are assumed to periodically give up CPU control.
    - Control is transfered frequently by making system calls.
    - An explicit yield system call transfers control the the OS.

  - Control is transferred when a process does something illegal.
    - Dividing by zero etc.
- **Non-Cooperative Approach: OS Takes Control**
  - The OS needs hardware help when a process refuses to make system calls or mistakes.
    - Rebooting the machine is the only recourse available.
  - The timer interrupt raises an interrupt every few milliseconds.
    - Programmed into the system and handled by the interrupt handler.
    - The OS tells the hardware what to do at boot time.
  - Hardware has some responsibility when an interrupt occurs.
    - Save enough state of the program to restore after the interrupt.
- **Saving and Restoring Context**
  - Once the OS regains control it has to decide to run the current process or switch to a different one.
  - The decision is made by the scheduler.
  - If the process switches the OS will execute a context switch.
    - Save the register values for the current process.
    - Restore registers for the new process.
    - To save the currently running process the OS will execute some low level assembly code.
      - Save the general registers.
      - Save the stack pointer.
      - Restore the registers for the new process.
      - Switch the stack to the new process.

6.4 Concurrency

- An OS may disable interrupts during interupt processing.
  - Keeps multiple interrupts from going to the CPU at the same

time.

- Locking schemes protect concurrent access to internal data structures.
    - Allowing multiple activities in teh kernel at the same time.