# 6. Mechanism: Limited Direct Execution

Operating System: Three Easy Pieces

# How to efficiently virtualize the CPU with control?

- The OS needs to share the physical CPU by time sharing.
- Issue
  - **Performance**: How can we implement virtualization without adding excessive overhead to the system?
  - **Control**: How can we run processes efficiently while retaining control over the CPU?

# Direct Execution (without limits!)

- Just run the program directly on the CPU.

| OS | Program |
|---|---|
| 1. Create entry for process list<br>2. Allocate memory for program<br>3. Load program into memory<br>4. Set up stack with `argc` / `argv`<br>5. Clear registers<br>6. Execute call `main()` | |
| | 7. Run `main()`<br>8. Execute `return` from `main()` |
| 9. Free memory of process<br>10. Remove from process list | |

**Without *limits* on running programs,**
**the OS wouldn't be in control of anything and**
**thus would be "just a library"**

# Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as …
  - Issuing an I/O request to a disk
  - Gaining access to more system resources such as CPU or memory

- **Solution**: Using protected control transfer (processor has to support it)
  - User mode: Applications do not have full access to hardware resources.
  - Kernel mode: The OS has access to the full resources of the machine

# System Call

- Allow the kernel to carefully expose certain key pieces of functionality to user program, such as …
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory

**But, why they look like a regular procedure call "sometimes" (e.g. libc calls)?**

# System Call (Cont.)

- **Trap** instruction
  - Jump into the kernel (how to tell where?)
  - Raise (the processor) privilege level to kernel mode

- **Return-from-trap** instruction
  - Return into the calling user program
  - Reduce (the processor) privilege level back to user mode

# Limited Direction Execution Protocol

| OS @ boot (kernel mode) | Hardware |
|---|---|
| **initialize trap table** | |
| | remember address of … syscall handler |

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| Create entry for process list<br>Allocate memory for program<br>Load program into memory<br>Setup user stack with argv<br>Fill kernel stack with reg/PC<br>**return-from -trap** | | |
| | restore regs from kernel stack<br>move to user mode<br>jump to main | |
| | | Run main()<br>…<br>Call system<br>**trap** into OS |

# Limited Direction Execution Protocol (Cont.)

| **OS @ run (kernel mode)** | **Hardware** | **Program (user mode)** |
|---|---|---|
| | *(Cont.)* | |
| | save regs to kernel stack<br>move to kernel mode<br>jump to trap handler | |
| Handle trap<br>Do work of syscall<br>**return-from-trap** | | |
| | restore regs from kernel stack<br>move to user mode<br>jump to PC after trap | |
| | | …<br>return from main<br>trap (via `exit()`) |
| Free memory of process<br>Remove from process list | | |

# Problem 2: Switching Between Processes

- How can the OS regain control of the CPU so that it can switch between *processes*?
  - A cooperative Approach: **Wait for system calls**
  - A Non-Cooperative Approach: **The OS takes control**

# A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making **system calls** such as `yield`.
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal.
    - Divide by zero
    - Try to access memory that it shouldn't be able to access

  - Ex) Early versions of the Macintosh OS, The old Xerox Alto system

**A process gets stuck in an infinite loop.**
**→ Reboot the machine**

# A Non-Cooperative Approach: OS Takes Control

- **A timer interrupt**
  - During the boot sequence, the OS start the timer (hardware).
  - The timer raise an interrupt every so many milliseconds. (hardware)
  - When the interrupt is raised :
    - The currently running process is halted.
    - Save enough of the state of the program
    - A pre-configured interrupt handler in the OS runs.

**A timer interrupt gives OS the ability to run again on a CPU.**

# Saving and Restoring Context

- Scheduler makes a decision:
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS executes context switch.

# Context Switch

- A low-level piece of assembly code
  - **Save a few register values** for the current process onto its kernel stack
    - General purpose registers
    - PC
    - kernel stack pointer
  - **Restore a few** for the soon-to-be-executing process from its kernel stack
  - **Switch to the kernel stack** for the soon-to-be-executing process

# Limited Direction Execution Protocol (Timer interrupt)

| OS @ boot (kernel mode) | Hardware |
|---|---|
| **initialize trap table** | |
| | remember address of …<br>syscall handler<br>timer handler |
| **start interrupt timer** | |
| | start timer<br>interrupt CPU in X ms |

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | | Process A<br>… |
| | **timer interrupt**<br>save regs(A) to k-stack(A)<br>move to kernel mode<br>jump to trap handler | |

# Limited Direction Execution Protocol (Timer interrupt)

| **OS @ run (kernel mode)** | **Hardware** | **Program (user mode)** |
|---|---|---|
| | *(Cont.)* | |
| Handle the trap<br>Call switch() routine<br>  save regs(A) to proc-struct(A)<br>  restore regs(B) from proc-struct(B)<br>  switch to k-stack(B)<br>**return-from-trap (into B)** | | |
| | restore regs(B) from k-stack(B)<br>move to user mode<br>jump to B's PC | |
| | | Process B<br>… |

# The xv6 Context Switch Code

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
        # Save old registers
        movl 4(%esp), %eax              # put old ptr into eax
        popl 0(%eax)                    # save the old IP (pop from stack to mem)
        movl %esp, 4(%eax)              # and stack
        movl %ebx, 8(%eax)              # and other registers
        movl %ecx, 12(%eax)
        movl %edx, 16(%eax)
        movl %esi, 20(%eax)
        movl %edi, 24(%eax)
        movl %ebp, 28(%eax)
        # Load new registers
        movl 8(%esp), %eax              # put new ptr into eax
        movl 28(%eax), %ebp             # restore other registers
        movl 24(%eax), %edi
        movl 20(%eax), %esi
        movl 16(%eax), %edx
        movl 12(%eax), %ecx
        movl 8(%eax), %ebx
        movl 4(%eax), %esp              # stack is switched here
        pushl 0(%eax)                   # return addr put in place
        ret                             # finally return into new ctxt
```

# Current xv6 Code

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

In `proc.h`
// Don't need to save `%eax, %ecx, %edx,` because the 46
// x86 convention is that the caller has saved them.

No actual change to new `%eip`, because we need to "switch" memory addressing space before (done in the scheduler `switchkvm();`)

# Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
  - Use a number of sophisticated **locking** schemes to protect concurrent access to internal data structures.

- Disclaimer: This lecture slide set is used in AOS course in University of Cantabria. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)