Universidade Estadual de Campinas
Instituto de Computação

MC504 Sistemas Operacionais

T03 CPU Virtualization
Limited Direct Execution

Arthur João Catto, PhD

2º semestre de 2018

To virtualize the CPU, the OS needs to share the physical CPU among many jobs running seemingly at the same time.

- Virtualization is achieved by time sharing the CPU:
  - Run one process for a little while;
  - Then run another one, and so forth.

To virtualize the CPU, the OS needs to share the physical CPU among many jobs running seemingly at the same time.

- Virtualization is achieved by time sharing the CPU:
  - Run one process for a little while;
  - Then run another one, and so forth.
- Building such virtualization machinery poses a few challenges:
  - Performance
  - Control

How to virtualize the CPU
in an efficient manner
while retaining control over the system?

First attempt at a solution:
Direct Execution

- Just run the program directly on the CPU.

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

What is a basic flaw in that proposal?

Without limits on running programs,
the OS wouldn’t be in control of anything
and thus would be “just another library”.

Two problems of direct execution

1. How can the OS make sure a program doesn't do anything that it is not supposed to do, while still running it efficiently?

2. When a process is run, how does the operating system stop it from running and switch to another process, thus implementing the time sharing we require to virtualize the CPU?

Problem #1: Restricted Operations

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

A process must be able to perform I/O and some other restricted operations, but without gaining complete control over the system.

How can the OS and hardware work together to do so?

Solution to Problem #1: Restricted Operations

- Implementing dual-mode operation
  - User mode
    - Code running in this mode does not have full access to hardware resources.
  - Kernel mode
    - Code running in this mode has full access to all resources of the machine.
- User applications always run in user mode. The OS always runs in kernel mode.
- To execute a restricted operation, a user application makes a **system call** to an OS function.

System Calls

- System calls allow the kernel to *carefully expose certain key pieces of functionality* to a user program, such as …
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory
- A system call looks like a normal function call, but hidden inside that function call lies a **trap** instruction.

Implementing system calls

- During a system call, a program eventually executes a special **trap** instruction.
  - A **trap** instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode.
- Once in the kernel, the OS can perform any required privileged operation (if allowed), and thus do the required work on behalf of the calling process.
- When finished, the OS calls a special **return-from-trap** instruction, which returns to the calling process while simultaneously reducing the privilege level back to user mode.
- This creates a Limited Direct Execution Protocol, which is discussed next.

Limited Direct Execution Protocol

OS @ boot (kernel mode)	Hardware	
• Initialize trap table		
	• Remember address of syscall handler	

Limited Direct Execution Protocol (ctd.)

OS @ run (kernel mode)	Hardware	Program (user mode)
• Create entry for process list • Allocate memory for program • Load program into memory • Setup user stack with argv • Fill kernel stack with reg/PC **return-from-trap**		
	• Restore regs from kernel stack • Move to user mode • Jump to main	
		• Run main() • … • Make system call • Trap into OS

Limited Direct Execution Protocol (ctd.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	• save regs to kernel stack • move to kernel mode • jump to trap handler	
• Handle trap • Do work of syscall • **return-from-trap**		
	• Restore regs from kernel stack • Move to user mode • Jump to PC after trap	
		• … • Return from main • **trap** (via exit())
• Free memory of process • Remove from process list		

Problem #2: Switching Between Processes

- In order to being able to use the CPU while a process is doing an I/O operation, for instance, the OS must switch to another process.

Since the OS is not executing at the time the I/O operation is initiated, how can it *regain control* of the CPU so that it can switch between processes?

- We will study two possible approaches:
  - A cooperative approach: **Wait for system calls**
  - A non-cooperative approach: **The OS takes control**

First solution to Problem #2

A cooperative Approach: **Wait for system calls**

- This approach was adopted in early versions of the Macintosh OS and in the old Xerox Alto system.
- In this approach, the OS *trusts* processes to behave reasonably.
- Reasonable processes periodically give up the CPU by making **system calls** such as an I/O operation or even **`yield`**.
- Application also transfer control to the OS when they attempt to do something illegal, such as dividing by zero or try to access memory that it shouldn't be able to access.
- Once the OS regains control of the CPU, it may decide to run some other task.

Isn’t this passive approach less than ideal?

What happens, for example, if a process (whether malicious, or just full of bugs) ends up in an infinite loop, and never makes a system call?

What can the OS do then?
Without additional help from the HW, nothing, really. The system must be rebooted.

Second Solution to Problem #2

A non-cooperative approach: **OS Takes Control**

- Providing a timer interrupt
  - During the boot sequence, the OS starts the timer.
  - The timer raises an interrupt every so many milliseconds.
  - When the interrupt is raised a pre-configured **interrupt handler** runs.
  - At this point, the OS has regained control of the CPU, and thus can do what it pleases.
    - E.g. stop the current process and start a different one.

Saving and Restoring Context

- When the OS regains the control of the CPU, whether cooperatively via a system call, or more forcefully via a timer interrupt, a decision has to be made:
  - Whether to continue running the currently-running process, or switch to a different one.
- This decision is made by the **scheduler**, a critical part of the OS.
  - If the decision is made to switch, the OS executes a low-level piece of code which we refer to as a **context switch**.

Context Switch

- Operation done by a low-level (assembly) piece of code
  - **Save a few register values** for the currently-executing-process onto its kernel stack
    - General purpose registers
    - PC
    - Kernel stack pointer
  - **Restore a few register values** for the soon-to-be-executing process from its kernel stack.
  - **Switch to the kernel stack** for the soon-to-be-executing process

Limited Direct Execution Protocol (timer interrupt)

OS @ boot (kernel mode)	**Hardware**	
initialize trap table		
	remember addresses of… syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU in X ms	

Limited Direct Execution Protocol (timer interrupt)

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

Handle the trap
Call switch() routine
    save regs(A) to PCB(A)
    restore regs(B) from PCB(B)
    switch to k-stack(B)
return-from-trap (into B)		

	restore regs(B) from k-stack(B) move to user mode jump to PC (of B)	
		Process B…

The xv6-rev10 Context Switch 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
```

Worried About Concurrency?

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