Operating Systems and Networks

Kunwar Shaanjeet Singh Grover

Contents

Chapter 1

Virtualisation

1.0.1 Introduction

(one) Physical thing → Many Virtual thing

We are going to talk about virtualising:

- CPU
- Memory

Virtualisation is an **Illusion**. It makes a running program think that it has its own CPU and its own private memory.

Key aspects of Virtualisation:

- Efficient: No point in making something if it performs worse than the trivial solution
- Secure: We might want to restrict the program to read only its data, and not some other program's data.

1.1 CPU Virtualization

one (or a few) CPUs → Many virtual CPUs

The way we are going to do is **time sharing**. We give each program a little bit of time to use the cpu. We give A some time, the B some time, then C some time and then back to A. This is also called multi-programming.

Abstraction: Process (running program)

Components of a process:

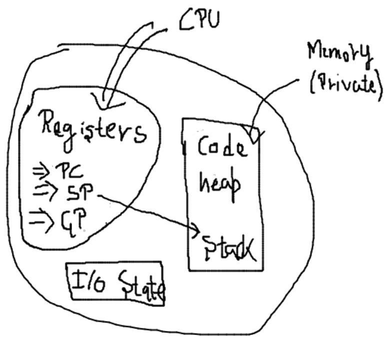

Figure 1.1: A process

At the low-level there are **Mechanisms**: How things work. You are going to need some mechanisms to switch to other programs. You also need some **Policies**: High-level decisions which are built on top of mechanisms.

Core mechanism behind all the cpu and memory virtualisation is **Limited Direct Execution**.

The *Direct Execution* part directly pretains to efficency.

The *Limited* part pretains to Security. The program shouldn't have access to everything on the system.

1.1.1 Direct Execution

We havent defined what an OS is yet. Lets just say for now OS: first program to run. We want to run a program "A".

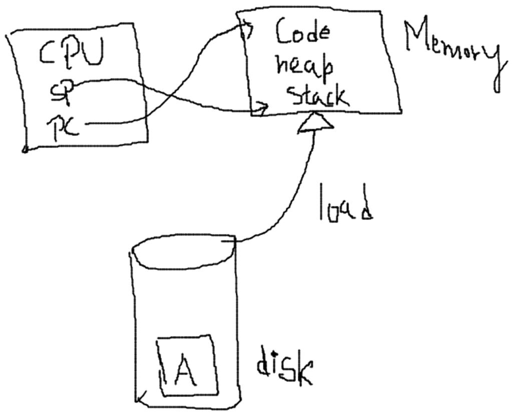

Figure 1.2: Running a program

Timeline: OS → A → . . .

Problems:

- What if "A" (user process) wants to do something *restricted*?
- What if OS wants to stop "A", run "B"? (How does the OS regain control?)
- What if "A" does something that is slow? (Disk I/O)

Problem 1: Restricted operation (in a controlled way)

Mode: per CPU bit.

- OS: "kernal mode"
  This mode can do **anything**
- User program: "user mode"
  Can only do a limited number of things

Its a little bit stored on each (virtual) cpu.

How to get into one of these modes? How to transition?

@boottime: boot in kernal mode. The OS is the first program runs. We are going to have trust on this program.

Want to run a user program: Special instruction that both:

1. Transition into user mode
2. Jumps in user program

User program: Wants to do something restricted (example: disk I/O)

1. Jump into kernal mode
2. Jumps into kernal
   If you could jump into the kernal just like this then you could do anything you want. This can be a security issue. We want to be able to switch to kernal mode from user mode but we dont want to allow to jump to any address in the kernal. We want to make this a restricted jump.

Two instructions:

- Trap
  Jump into the kernal (but @ restricted location) and then elevate "privilege" (use → kernal) also save enough register state so that we can return properly.
- Return from Trap

Timeline:

$$A \xrightarrow{\text{trap}} \text{OS} \rightarrow \text{does some stuff} \xrightarrow{\text{return from trap}} \ldots$$

These traps are called **System Calls**

@boottime: OS starts up in kernal mode. It tells the hardware where to jump to when somebody issues a kind of trap. For example, oh this is trap number 80. When the OS sees this trap number, it has already declared what trap number 80 is and jumps to what piece of code to run.

Setup all these *trap handlers* issuing special instruction to tell hardware where trap handler are in OS memory. Now, if someone issues a trap number, the OS knows where to jump to, to execute the program.

We need to save/restore "state" (register) of process:

For you what it looks like when you call a trapis, your execution was halted in mid and something happened and you go to the instruction after the trap.

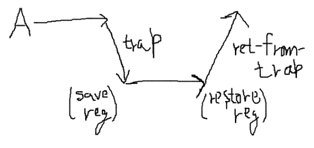

Figure 1.3: A system call

Problem 2: How to stop A,run B?

OS was running → A → while(1); → . . .

cooperative approach: hope that A doesnt do bad stuff

non-cooperative (preemptive) approach: based on h/w support. H/w support in this case is **timer interrupt**.

@boot: OS → kernal mode, installs trap handlers, starts interrupt timer.

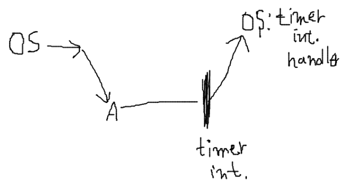

Figure 1.4: Interupt Timer

Background: CPU

Lets quickly refresh our background on how the CPU works.

(fig)

- Fetch (PC)
- Decode: Figure out which instruction it is (inc. PC)
- Execture: (could change PC)

This view is ok for a single program running. But when we have multiple programs running, this isnt enough.

Lets add more stuff to this:

Before execution: Check permission, is this thing OK to execute?

If its not ok: Raise an exception $\rightarrow$ OS gets involved (exception handler).

New instruction after Execution: **Process Interrupts**. Between instructions, there are external events outside of the program which the system needs to know about. A key has been pressed , the mouse has been moved, a timer has gone off, etc. These are things outside the program which need to be handeled.

In this step we do: Handle any pending interrupts $\rightarrow$ OS gets involved.