6.033 Spring 2015
Lecture #4

- Operating systems
- Virtual memory
- OS abstractions
what if we don’t want our modules to be on entirely separate machines? how can we enforce modularity on a single machine?
Operating systems: enforce modularity on a single machine via virtualization
Enforcing Modularity via Virtualization

in order to enforce modularity + build an effective operating system

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory

2. programs should be able to communicate

3. programs should be able to share a CPU without one program halting the progress of the others

**today’s goal:** virtualize memory so that programs cannot refer to each others’ memory
Single Program

for (; ;) {
    next instruction
}

CPU

interprets instructions

main memory

holds instructions

instructions

data
Single Program

CPU

instruction pointer

EIP

31 0

main memory

instructions

data

2^{32}-1

0
Multiple Programs

CPU₁ (used by program₁)

for (;;) {
    next instruction
}

CPU₂ (used by program₂)

for (;;) {
    next instruction
}

main memory

instructions for program₁

instructions for program₂

data for program₁

data for program₂

2^{32} - 1
Multiple Programs

**CPU₁** (used by program₁)

**CPU₂** (used by program₂)

**main memory**

- Instructions for program₁
- Instructions for program₂
- Data for program₁
- Data for program₂

**problem:** no boundaries
Solution: Virtualize Memory

**CPU**<sub>1</sub> (used by program<sub>1</sub>)

- **EIP**
- virtual address

**MMU**

- virtual address
- physical address

**Main memory**

- physical memory
- instructions for program<sub>1</sub>
- data for program<sub>1</sub>
- instructions for program<sub>2</sub>
- data for program<sub>2</sub>
- table for program<sub>1</sub>
- table for program<sub>2</sub>

**MMU uses program<sub>1</sub>’s table to translate the virtual address to a physical address**
**Storing the Mapping**

**naive method:** store every mapping; virtual address acts as an index into the table

```
0x00000000  →  0xbe26dc9
0x00000001  →  0xc090f81c
0x00000002  →  0xb762a572
0x00000003  →  0x5dcc90ee
...
```

32 bits per entry

= **16GB** to store the table
Storing the Mapping

**space-efficient mapping:** map to **pages** in memory

one page is (typically) \(2^{12}\) bits of memory.

\[
2^{32-12} = 2^{20} \text{ entries}
\]

32 bits\(^*\) per entry

\[
= 4\text{MB} \text{ to store the table}
\]

* you’ll see why it’s not 20 bits in a second
Using Page Tables

CPU$_1$(used by program$_1$)

EIP: 0x00002148

virtual page number: 0x00002
(top 20 bits)

offset: 0x148
(bottom 12 bits)

physical page number: 0x00004

MMU

0x00002148

MMU table for program$_1$

index into page table

0x00002148 → 0x00004148

to main memory

0x00003
0x00000
0x00004
0x00005
...
(exists in main memory)

Katrina LaCurts | lacurts@mit | 6.033 2015
Page Table Entries

Page table entries are 32 bits because they contain a 20-bit physical page number and 12 bits of additional information:

- **present (P) bit**: is the page currently in DRAM?

- **read/write (R/W) bit**: is the program allowed to write to this address?

- **user/supervisor (U/S) bit**: does the program have access to this address?
kernel manages page faults and other interrupts
operating systems: enforce modularity on a single machine via virtualization and abstraction
• Operating systems
  Operating systems enforce modularity on a single machine via **virtualization** and **abstraction**

• Virtual memory
  Virtualizing memory prevents programs from referring to (and corrupting) each other’s memory. The **MMU** translates virtual addresses to physical addresses using **page tables**

• OS abstractions
  The OS presents abstractions for devices via system calls, which are implemented with interrupts. Using interrupts means the **kernel** directly accesses the devices, not the user