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

CPU

for (; ;) {
    next instruction
}

interprets instructions

main memory

instructions

holds instructions

data
Single Program

CPU

instruction pointer

EIP

interprets instructions

main memory

instructions

data

holds instructions

2^{32} - 1
Multiple Programs

CPU₁ (used by program₁)

for (;;) {
    next instruction
}

CPU₂ (used by program₂)

for (;;) {
    next instruction
}

main memory

2^{32}-1

instructions for program₁

instructions for program₂

data for program₁

data for program₂
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\textsubscript{1} (used by program\textsubscript{1})

MMU

main memory

CPU\textsubscript{1} takes the virtual address and passes it to the MMU. The MMU uses program\textsubscript{1}'s table to translate the virtual address to a physical address. The physical address is then used to access the main memory, where the instructions and data for both program\textsubscript{1} and program\textsubscript{2} are stored.
naive method: store every mapping; virtual address acts as an index into the table

\[\begin{array}{c}
0x00000000 & \rightarrow & 0xbe26dc9 \\
0x00000001 & \rightarrow & 0xc090f81c \\
0x00000002 & \rightarrow & 0xb762a572 \\
0x00000003 & \rightarrow & 0x5dcc90ee \\
\ldots & \rightarrow & \ldots \\
\end{array}\]

\[2^{32} \text{ entries}\]

32 bits per entry

\[= 16GB \text{ to store the table}\]
space-efficient mapping: map to pages in memory

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

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

32 bits* per entry

= 4MB to store the table

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

**CPU**

- EIP: 0x00002148
  - 31
  - 0

**MMU**

- virtual page number: 0x000002
  - (top 20 bits)
- offset: 0x148
  - (bottom 12 bits)

**physical page number**: 0x000004

**table for program**

- 0x000003
- 0x000000
- 0x000004
- 0x000005
- ...

(exists in main memory)
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