6.033 Spring 2019

Lecture #3

- 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

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
operating systems enforce modularity on a single machine using 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
**operating systems** enforce modularity on a single machine using **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

---

6.033 | spring 2019 | lacurts@mit.edu
how does a program use memory?
Multiple Programs

**Problem:** no boundaries
Solution: Virtualize Memory

CPU₁ (used by program₁)

MMU

virtual address

physical address

physical memory

main memory

instructions for program₁

data for program₁

table for program₁

table for program₁

MMU uses program₁’s table to translate the virtual address to a physical address

EIP

31 0

virtual address

31 0

physical address

2^{32}-1

0

2^{32}-1

0

785 | spring 2019 | lacurts@mit.edu
naive method: store every mapping; virtual address acts as an index into the table

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

= 16GB to store the table
Using Page Tables

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

**MMU**

![Diagram showing the process of using page tables](image)

**virtual page number**: 0x00002
(top 20 bits)

**offset**: 0x148
(bottom 12 bits)

**physical page number**: 0x00004

The diagram illustrates the process of converting a virtual address to a physical address using a page table. The EIP register holds the virtual address, which is then used to index into the page table to find the corresponding physical page number. The page table exists in main memory.
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
Page Table Entries

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

31

physical page number

12 11

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

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

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

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

$2^{32-12} = 2^{20}$ entries

32 bits per entry

= **4MB** to store the table

**problem:** 4MB is still a fair amount of space
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

solution: page the page table
did we achieve our goal? is a program’s memory protected from corruption by another program?
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
operating systems: enforce modularity on a single machine via virtualization and abstraction
```c
#include <stdio.h>
#include <unistd.h>

void (*m)();

void f()
{
    printf("child is running m = %p\n", m);
}

int main()
{
    m = f;
    if (fork() == 0)
    {
        printf("child has started\n");
        int i;
        for (i = 0; i < 15; i++)
        {
            sleep(1);
            (*m)();
        }
    }
    else
    {
        printf("parent has started\n");
        sleep (5);
        printf("parent is running; let's write to m = %p\n", m);
        m = 0;
        printf("parent tries to invoke m = %p\n", m);
        (*m)();
        printf("parent is still alive\n");
    }
}
```

- `m` is a pointer to a function that returns `void`.
- Set `m` to point to `f`.
- Child: Every second for 15 seconds, call `m`.
- Parent: Overwrite `m` and then call it.
• **Operating systems** enforce modularity on a single machine via **virtualization** and **abstraction**

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

• 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