6.033 Spring 2018

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 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**  
   virtualize **memory**

2. programs should be able to **communicate**  
   virtualize **communication links**

3. programs should be able to **share a CPU** without one program halting the progress of the others  
   virtualize **processors**
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
how does a program use memory?
Multiple Programs

**CPU₁** (used by program₁)

- EIP: 31 0

**CPU₂** (used by program₂)

- EIP: 31 0

**main memory**

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

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

CPU₁ (used by program₁)

EIP

MMU

virtual address

physical address

physical memory

2^{32}-1

0

virtual address

2^{32}-1

0

main memory

instructions for program₁

data for program₁

instructions for program₂

data for program₂

physical address

0

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

PTR
Storing the Mapping

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

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0xbe26dc9</td>
</tr>
<tr>
<td>0x00000001</td>
<td>0xc090f81c</td>
</tr>
<tr>
<td>0x00000002</td>
<td>0xb762a572</td>
</tr>
<tr>
<td>0x00000003</td>
<td>0x5dcc90ee</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

2^{32} entries

32 bits per entry

= **16GB** to store the table
Using Page Tables

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

**MMU**

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

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

**physical page number**: 0x00004
(exists in main memory)

(index into page table)

to main memory

6.033 | spring 2018 | lacurts@mit.edu
**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

= **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:

- **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
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

**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 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");
    }
}
```

\textbf{m} is a pointer to a function that returns \textbf{void}

\textbf{set m} to point to \textbf{f}

\textbf{Child:} every second for 15 seconds, call \textbf{m}

\textbf{Parent:} overwrite \textbf{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