6.033 Spring 2016
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
```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 function pointer to a function that returns `void`

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

**Parent**: overwrite \texttt{m} and then call it
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
   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
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

- Data
- Instructions
Single Program

CPU

instruction pointer

EIP

interprets instructions

main memory

instructions

holds instructions

2^{32} - 1

data

0

Katrina LaCurts | lacurts@mit | 6.033 2016
Multiple Programs

**CPU\textsubscript{1}** (used by program\textsubscript{1})

- main memory
  - instructions for program\textsubscript{1}
  - instructions for program\textsubscript{2}
  - data for program\textsubscript{1}
  - data for program\textsubscript{2}

**CPU\textsubscript{2}** (used by program\textsubscript{2})

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

**CPU** \(_1\) (used by program\(_1\))

**MMU**

**main memory**

- **instructions for program\(_1\)**
- **data for program\(_1\)**
- **instructions for program\(_2\)**
- **data for program\(_2\)**
- **table for program\(_1\)**
- **table for program\(_2\)**

MMU uses program\(_1\)’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

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

\(2^{32}\) entries

32 bits per entry

= 16GB 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\textsubscript{1} (used by program\textsubscript{1})

<table>
<thead>
<tr>
<th>EIP</th>
<th>0x00002148</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

MMU

<table>
<thead>
<tr>
<th>virtual page number: 0x00002</th>
</tr>
</thead>
<tbody>
<tr>
<td>(top 20 bits)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>offset: 0x148</th>
</tr>
</thead>
<tbody>
<tr>
<td>(bottom 12 bits)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>physical page number: 0x00004</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>table for program\textsubscript{1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000003</td>
</tr>
<tr>
<td>0x000000</td>
</tr>
<tr>
<td>0x000004</td>
</tr>
<tr>
<td>0x000005</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

(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?

<table>
<thead>
<tr>
<th>Bit</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Physical page number</td>
</tr>
<tr>
<td>12-11</td>
<td>Additional information</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Katrina LaCurts | lacurts@mit | 6.033 2016
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