IF YOU WANT TO USE A LAPTOP, SIT ON THE LEFT SIDE OF THE CLASSROOM
6.033 Spring 2020
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
operating systems enforce modularity on a single machine

in order to enforce modularity + build an effective operating system
**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**
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
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
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)

CPU₂ (used by program₂)

main memory

instructions for program₁

instructions for program₂

data for program₁

data for program₂

EIP 31 0

EIP 31 0

2^{32} - 1

0
Multiple Programs

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

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

**main memory**

- Instructions for program\textsubscript{1}
- Instructions for program\textsubscript{2}
- Data for program\textsubscript{1}
- Data for program\textsubscript{2}

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

CPU₁ (used by program₁)

EIP

31 0

main memory

2^{32} - 1

0
Solution: Virtualize Memory

CPU_1 (used by program_1)

EIP
31 0

Instructions for program_1
Data for program_1

Instructions for program_2
Data for program_2

Main memory
0

2^{32} - 1
Solution: Virtualize Memory

**CPU** (used by program 1)

| EIP | 31 | 0 |

---

**Main memory**

- **Virtual address**
  - **Instructions for program**
  - **Data for program**

- **Physical memory**
  - **Instructions for program**
  - **Data for program**
Solution: Virtualize Memory

CPU \( \text{CPU}_1 \) (used by program \( \text{program}_1 \))

- EIP
- virtual address

- Instructions for program \( \text{program}_1 \)
- Data for program \( \text{program}_1 \)

- Instructions for program \( \text{program}_2 \)
- Data for program \( \text{program}_2 \)

Main memory

Virtual address

Physical memory

\( 0 \) to \( 2^{32} - 1 \)
Solution: Virtualize Memory

CPU₁ (used by program₁)

MMU

physical memory

main memory

virtual address

0

$2^{32}-1$

instructions for program₁

data for program₁

instructions for program₂

data for program₂

EIP

virtual address

0

$2^{32}-1$
Solution: Virtualize Memory

CPU₁ (used by program₁)

EIP

virtual address

0

31

MMU

physical address

virtual address

physical memory

2^{32}-1

instructions for program₁

data for program₁

instructions for program₂

data for program₂

main memory

0

6.033 | spring 2020 | lacurts@mit.edu
Solution: Virtualize Memory

CPU₁ (used by program₁)

EIP

MMU

physical address

virtual address

2^{32}-1

0

physical memory

main memory

instructions for program₁

data for program₁

table for program₁

instructions for program₂

data for program₂

table for program₂
Solution: Virtualize Memory

CPU₁ (used by program₁)

MMU

virtual address

physical address

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

main memory

physical memory

instructions for program₁

data for program₁

table for program₁

instructions for program₂

data for program₂

table for program₂
**Solution: Virtualize Memory**

- **CPU$_1$** (used by program$_1$)
  - EIP
  - virtual address

- **MMU**
  - virtual address
  - physical address
  - PTR

- **Main Memory**
  - physical memory
  - $2^{32} - 1$
  - 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.

6.033 | spring 2020 | lacurts@mit.edu
**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>Mapping Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0xbe26dc94</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>
</tbody>
</table>

...
Storing the Mapping

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

```
0x00000000 → 0xbe26dc94
0x00000001 → 0xc090f81c
0x00000002 → 0xb762a572
0x00000003 → 0x5dcc90ee
...
```

$2^{32}$ entries

32 bits per entry

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

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

\begin{tabular}{c}
EIP \\
\hline
31 & 0x00002148 \\
\end{tabular}

\begin{tabular}{c}
MMU \\
\hline
\end{tabular}

table for program\textsubscript{1}

\begin{tabular}{c}
0x000003 \\
0x000000 \\
0x000004 \\
0x000005 \\
\ldots \\
\end{tabular}

(exists in main memory)
Using Page Tables

$\text{CPU}_1$ (used by program$_1$)

$\text{EIP} \quad 0\times00002148$

$\quad 31 \quad 0$

$\rightarrow$

$\text{MMU}$

$0\times00002148$

$\text{table for program}_1$

$\begin{array}{l}
0\times00003 \\
0\times00000 \\
0\times00004 \\
0\times00005 \\
\ldots \\
\end{array}$

(exists in main memory)
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>31</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**MMU**

| 0x00002148 |

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

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

**table for program\textsubscript{1}**

| 0x00003 |
| 0x00000 |
| 0x00004 |
| 0x00005 |

(exists in main memory)
Using Page Tables

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

```
EIP: 0x00002148
31 0
```

**MMU**

```
0x00002148
```

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

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

**table for program**<sub>1</sub>

```
0x000003
0x000000
0x000004
0x000005
...
```

(exists in main memory)
Using Page Tables

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

EIP: 0x00002148
31 0

MMU

0x00002148

virtual page number: 0x00002
(top 20 bits)

offset: 0x148
(bottom 12 bits)

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

Table for program<sub>1</sub>

<table>
<thead>
<tr>
<th>physical page number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00003</td>
</tr>
<tr>
<td>0x00000</td>
</tr>
<tr>
<td>0x00004</td>
</tr>
<tr>
<td>0x00005</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

6.033 | spring 2020 | lacurts@mit.edu
Using Page Tables

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

EIP: 0x00002148
    31 0

virtual page number: 0x00002
(top 20 bits)

offset: 0x148
(bottom 12 bits)

physical page number: 0x00004

MMU

0x00002148 → 0x00004148

table for program<sub>1</sub>

index into page table

<table>
<thead>
<tr>
<th>Physical Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00003</td>
</tr>
<tr>
<td>0x00000</td>
</tr>
<tr>
<td>0x00004</td>
</tr>
<tr>
<td>0x00005</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

(exists in main memory)
Using Page Tables

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

EIP: \texttt{0x00002148}

virtual page number: \texttt{0x00002}
(top 20 bits)

offset: \texttt{0x148}
(bottom 12 bits)

physical page number: \texttt{0x00004}

MMU

0x00002148 \rightarrow 0x00004148

to main memory

table for program\textsubscript{1}

\begin{tabular}{|c|}
\hline
0x00003 \\
0x00000 \\
0x00004 \\
0x00005 \\
\ldots \\
\hline
\end{tabular}

(exists in main memory)
Storing the Mapping

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

One page is (typically) $2^{12}$ bytes of memory.

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

32 bits* per entry

= **4MB** to store the table

* 20 bits to store the physical page address, plus 12 additional bits that we’ll come to shortly
what if we run out of 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

31 12 11 0

physical page number

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

  If it's not, the access triggers a page fault. Some of the other bits in the page entry provide information to help the OS bring the page into memory.
**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

= **4MB** 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

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?
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 12 11 0

- **physical page number**

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

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

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

- **Physical page number**: 20 bits
- **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
#include <stdio.h>
#include <unistd.h>

int main() {
    fork()
}

int main() {

    if (fork() == 0) {
        printf("child has started\n");

    }

else {
    printf("parent has started\n");

}}
```c
#include <stdio.h>
#include <unistd.h>

int main() {

    if (fork() == 0) {
        printf("child has started\n");
        int i;
        for (i = 0; i < 15; i++) {
            sleep(1);
            printf("child is running\n");
        }
    }

    else {
        printf("parent has started\n");
    }
}
```
#include <stdio.h>
#include <unistd.h>

void f() {
  printf("child is running\n");
}

int main() {

  if (fork() == 0) {
    printf("child has started\n");
    int i;
    for (i = 0; i < 15; i++) {
      sleep(1);
      f();
    }
  }

  else {
    printf("parent has started\n");
  }

}
#include <stdio.h>
#include <unistd.h>

void (*m)();

void f() {
    printf("child is running\n");
}

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");
    }
}
```c
#include <stdio.h>
#include <unistd.h>

void (*m)();

void f()
{
    printf("child is running\n");
}

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");
    }
}
```
```c
#include <stdio.h>
#include <unistd.h>

void (*m)();

void f()
{
    printf("child is running\n");
}

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");
    }
}
```

- `m` is a pointer to a function that returns `void`.
- Set `m` to point to `f`. 

6.033 | spring 2020 | lacurts@mit.edu
```c
#include <stdio.h>
#include <unistd.h>

void (*m)();

void f() {
    printf("child is running\n");
}

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");
    }
}
```

- m is a pointer to a function that returns `void`
- set m to point to f
- Child: every second for 15 seconds, call m
```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");
    }
}
```

- `m` is a pointer to a function that returns `void`.
- Set `m` to point to `f`.
- Child: every second for 15 seconds, call `m`. 
```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 $m$
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