6.033 in the news

A Change by Apple Is Tormenting Internet Companies, Especially Meta

Meta’s stock prices plunged after the company reported that Apple’s privacy features would cost it billions this year. It’s not the only tech giant to take a hit.

system design choices impact more than just that system’s users

https://www.nytimes.com/2022/02/03/technology/apple-privacy-changes-meta.html
Lecture #3: Virtual Memory

how does it work, but more importantly, why does an OS use it?
**last time:** enforced modularity via client/server + naming

---

**Class Browser** (on machine 1)

```python
def main():
    html = browser_load_url(URL)
    ...

def browser_load_url(url):
    msg = url  # could reformat
    send request
    wait for reply
    html = reply  # could reformat
    return html
```

**Class Server** (on machine 2)

```python
def server_load_url():
    ...
    return html

def handle_server_load_url(url):
    wait for request
    url = request
    html = server_load_url(url)
    reply = html
    send reply
```

---

Load("kaws.com/buy.html?dogfood")

---

stub

---

stub
**last time:** enforced modularity via client/server + naming

```python
def main():
    html = browser_load_url(URL)
    ...

def browser_load_url(url):
    msg = url  # could reformat
    send request
    wait for reply
    html = reply  # could reformat
    return html
```

---

**today:** what if we don’t want to put each module on a separate machine?
operating systems enforce modularity on a single machine
operating systems enforce modularity on a single machine

in order to enforce modularity + have an effective operating system, a few things need to happen
**operating systems** enforce modularity on a single machine

In order to enforce modularity + have an effective operating system, a few things need to happen

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 + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory

2. programs should be able to communicate with each other
operating systems enforce modularity on a single machine

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory

2. programs should be able to communicate with each other

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

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory

2. programs should be able to communicate with each other

3. programs should be able to share a CPU without one program halting the progress of the others

the primary technique that an operating system uses to enforce modularity is virtualization
operating systems enforce modularity on a single machine 

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory

2. programs should be able to communicate with each other

3. programs should be able to share a CPU without one program halting the progress of the others

the primary technique that an operating system uses to enforce modularity is virtualization

in some sense, we want every program to think that it has access to the full physical hardware, when of course they don’t; the OS virtualizes different components of hardware
operating systems enforce modularity on a single machine

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory
   
   virtualize memory

2. programs should be able to communicate with each other
   
   assume they don’t need to (for today)

3. programs should be able to share a CPU without one program halting the progress of the others
   
   assume one program per CPU (for today)

the primary technique that an operating system uses to enforce modularity is virtualization

in some sense, we want every program to think that it has access to the full physical hardware, when of course they don’t; the OS virtualizes different components of hardware
what we want: virtualization. every program should appear to have access to a full 32-bit address space
what we want: virtualization. every program should appear to have access to a full 32-bit address space

what we have: $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

what we have: $2^{32}$ bytes of memory; every program can’t *actually* have access to the full 32-bit space
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

what we have: $2^{32}$ bytes of memory; every program can’t *actually* have access to the full 32-bit space

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

```
<table>
<thead>
<tr>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

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

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

**CPU<sub>2</sub>** (used by program<sub>2</sub>)

---
**what we want:** virtualization. Every program should appear to have access to a full 32-bit address space.

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space.

---

**CPU**₁ (used by program₁)

```
EIP
31 0
```

**CPU**₂ (used by program₂)

---

**main memory**
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

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

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

**CPU$_2$** (used by program$_2$)

---

**main memory**

- instructions and data for program$_1$
  - $0x00000000$ to $0xFFFFFFFF$ ($2^{32}-1$)
  - $0xF0000000$
  - $0xE000000$
  - $0x0000000$

- instructions and data for program$_2$
  - $0x00000000$
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

what we have: \(2^{32}\) bytes of memory; every program can’t actually have access to the full 32-bit space

---

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

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

**CPU\(_2\)** (used by program\(_2\))

---

**main memory**

- instructions and data for program\(_1\)
  - `0xFFFFFFF (2^{32}-1)`
  - `0xF000000`
  - `0xE000000`

- instructions and data for program\(_2\)
  - `0xE000000`

- `...`

- `0x00000000`
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

**CPU**

- **CPU$_1$** (used by program$_1$)
  - EIP
    - 31 0

- **CPU$_2$** (used by program$_2$)
  - Instructions and data for program$_1$
  - Instructions and data for program$_2$
  - ...
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

**CPU**

1. (used by program 1)

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

2. (used by program 2)

---

**memory management unit (MMU)**

---

**main memory**

- instructions and data for program 1
- instructions and data for program 2
- ...

---

Katrina LaCurts | lacurts@mit.edu | 6.033 2022
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** \(2^{32}\) bytes of memory; every program can’t actually have access to the full 32-bit space

---

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

- EIP
  - \(0x00002148\)

**CPU\(_2\)** (used by program\(_2\))

---

**memory management unit (MMU)**

- \(0x00002148\)

---

**main memory**

- instructions and data for program\(_1\)
  - \(0xFFFFFFFF\)
  - \(0xF0000000\)
- instructions and data for program\(_2\)
  - \(0xE0000000\)
- ...
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

![Diagram](image_url)

- **CPU₁** (used by program₁)
  - EIP: 0x000002148

- **CPU₂** (used by program₂)

- **memory management unit (MMU)**

- **main memory**
  - instructions and data for program₁
  - instructions and data for program₂
  - ... (omitted)
  - table for program₁
  - table for program₂
  - 0xFFFFFFFF (2^{32} - 1)
  - 0xFF000000
  - 0xE000000
  - 0x00000000
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

The MMU is going to use program 1’s table to translate a virtual address from program 1 into a physical address.
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

**attempt 1:** each virtual address acts as an index into this table; there is one entry for every virtual address

---
what we want: virtualization. every program should appear to have access to a full 32-bit address space

what we have: $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

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

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

memory management unit (MMU)

main memory

attempt 1: each virtual address acts as an index into this table; there is one entry for every virtual address
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

**CPU**

1. **CPU**₁ (used by program₁)
   - EIP: 0x00002148

2. **CPU**₂ (used by program₂)

---

**memory management unit (MMU)**

- 0x00002148

---

**main memory**

- instructions and data for program₁
  - 0xFF035113
  - 0xF27A9B77
  - 0xF0110048
  - 0xF8887881
  - ...  
- instructions and data for program₂
- table for program₁
- table for program₂
- 0xFFFF0000
- 0xE000000
- 0xFFFFFFFF
- (2$^{32}$ - 1)

---

**attempt 1:** each virtual address acts as an index into this table; there is one entry for every virtual address

2$^{32}$ virtual addresses each mapping to a 32-bit physical address →
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

**CPU** (used by program 1)

EIP

0x00002148

---

**CPU** (used by program 2)

---

**attempt 1:** each virtual address acts as an index into this table; there is one entry for every virtual address

2$^{32}$ virtual addresses each mapping to a 32-bit physical address → 16GB to store this table
**what we want:** virtualization. Every program should appear to have access to a full 32-bit address space. 

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space.

---

**attempt 1:** Each virtual address acts as an index into this table; there is one entry for every virtual address. 

2$^{32}$ virtual addresses each mapping to a 32-bit physical address → 16GB to store this table. 

We don’t even have 16GB of memory.
what we want: *virtualization*. every program should appear to have access to a full 32-bit address space

what we have: $2^{32}$ bytes of memory; every program can’t *actually* have access to the full 32-bit space

---

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

- EIP: 0x00002148

**CPU$_2$** (used by program$_2$)

---

**memory management unit (MMU)**

- Instruction指针 (EIP): 0x00002148

---

**main memory**

- Instructions and data for program$_1$ at 0x00000000
- Instructions and data for program$_2$ at 0xE0000000
- Page table for program$_1$ at 0xF0000000
- Page table for program$_2$ at 0x00000000

---

Katrina LaCurts | lacurts@mit.edu | 6.033 2022
what we want: virtualization. every program should appear to have access to a full 32-bit address space

what we have: $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

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

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

**CPU**<sub>2</sub> (used by program<sub>2</sub>)

**memory management unit (MMU)**

- 0x00002148

**main memory**

- instructions and data for program<sub>1</sub>
- instructions and data for program<sub>2</sub>
- page table for program<sub>1</sub>
- page table for program<sub>2</sub>

---

0xFFFFFFFF

(2<sup>32</sup>-1)

0xF0000000

0x007A1200

0x003D0900

0x00000000

---

Katrina LaCurts | lacurts@mit.edu | 6.033 2022
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can't actually have access to the full 32-bit space

---

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

**CPU$_2$** (used by program$_2$)

**memory management unit (MMU)**

**main memory**

- instructions and data for program$_1$
  - PTR$_1$ 0x007A1200
  - 0xFFFFF

- instructions and data for program$_2$
  - PTR$_2$ 0x003D0900

- page table for program$_1$
  - 0xE000000

- page table for program$_2$
  - 0x003D0900
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

CPU₁ (used by program₁)

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

CPU₂ (used by program₂)

---

memory management unit (MMU)

- 0x00002148
- PTR₁: 0x007A1200
- PTR₂: 0x003D0900

---

main memory

- instructions and data for program₁
- instructions and data for program₂
- page table for program₁
- page table for program₂

---

what we have: 2³² bytes of memory; every program can’t actually have access to the full 32-bit space
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

**page tables:** top 20 bits of the virtual address act as an index into this table

(a page of memory is $2^{32-20}=2^{12}$ bytes)
**what we want:** virtualization. Every program should appear to have access to a full 32-bit address space.

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space.

---

**CPU**

- CPU1 (used by program1)
  - EIP: 0x00002148

- CPU2 (used by program2)

---

**memory management unit (MMU)**

- Virtual page number: 0x00002 (top 20 bits)

---

**main memory**

- Instructions and data for program1
- Instructions and data for program2
- Page table for program1
- Page table for program2

---

**virtual page number:** 0x00002

**page tables:** Top 20 bits of the virtual address act as an index into this table.

(A page of memory is $2^{32-20}=2^{12}$ bytes.)
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

**CPU$^1$** (used by program$^1$)

- EIP: $0x00002148$
  - 31 0

**CPU$^2$** (used by program$^2$)

---

**memory management unit (MMU)**

- Virtual page number: $0x00002$
  - (top 20 bits)
- Physical page number: $0xF0110$

---

**main memory**

- Instructions and data for program$^1$
- Instructions and data for program$^2$
- Page table for program$^1$
- Page table for program$^2$

---

**page tables:** top 20 bits of the virtual address act as an index into this table

(a page of memory is $2^{32-20}=2^{12}$ bytes)
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

**CPU**

1. **EIP**
   - $0x00002148$
   - 31 0

2. **CPU**
   - used by program

**memory management unit (MMU)**

- **EIP**
  - $0x00002148$
  - 31 0

**main memory**

- Instructions and data for program
- Instructions and data for program
- Page table for program
- Page table for program

---

**virtual page number:** $0x00002$

**physical page number:** $0xF0110$

**offset:** $0x148$

**page tables:** top 20 bits of the virtual address act as an index into this table

(a page of memory is $2^{32-20}=2^{12}$ bytes)
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

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

- EIP: 0x00002148
- virtual page number: 0x00002
- physical page number: 0xF0110
- offset: 0x148

CPU<sub>2</sub> (used by program<sub>2</sub>)

---

Memory management unit (MMU)

- instructions and data for program<sub>1</sub>
- instructions and data for program<sub>2</sub>

Main memory

- page table for program<sub>1</sub>
- page table for program<sub>2</sub>

---

Page tables: top 20 bits of the virtual address act as an index into this table

(a page of memory is $2^{32-20}=2^{12}$ bytes)
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

what we have: $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

**CPU**

<table>
<thead>
<tr>
<th>CPU1 (used by program1)</th>
<th>memory management unit (MMU)</th>
<th>main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIP</td>
<td>0x00002148</td>
<td>instructions and data for program1</td>
</tr>
<tr>
<td>0x00002148</td>
<td>0xF0110148</td>
<td>instructions and data for program2</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>page table for program1</td>
</tr>
</tbody>
</table>

**CPU2**

<table>
<thead>
<tr>
<th>CPU2 (used by program2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

---

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

**physical page number**: 0xF0110

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

---

**page tables**: top 20 bits of the virtual address act as an index into this table

(a page of memory is $2^{32-20}=2^{12}$ bytes)
**what we want:** virtualization. every program should appear to have access to a full 32-bit address space

**what we have:** $2^{32}$ bytes of memory; every program can’t actually have access to the full 32-bit space

---

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

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

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

---

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

**physical page number:** 0xF0110

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

---

**page tables:** top 20 bits of the virtual address act as an index into this table
(a page of memory is $2^{32-20}=2^{12}$ bytes)

---

2\textsuperscript{20} virtual page numbers each mapping to a 32-bit page-table entry (PTE) $\rightarrow$ 4MB to store this table
(why 32-bit PTEs, not 20-bit? hang on)
we have two more broad areas to cover:
we have two more broad areas to cover:

*does virtual memory protect programs from accessing each other’s memory?*
(to answer this, we’ll need to address some other issues first)
we have two more broad areas to cover:

*does virtual memory protect programs from accessing each other’s memory?*
(to answer this, we’ll need to address some other issues first)

what performance issues matter here?
what happens if we don’t have enough memory to store all of our programs’ instructions and data?
what happens if we don’t have enough memory to store all of our programs’ instructions and data?
what happens if we don’t have enough memory to store all of our programs’ instructions and data? page table entries contain additional bits that help us deal with this problem (and others)
what happens if we don’t have enough memory to store all of our programs’ instructions and data?

Page table entries contain additional bits that help us deal with this problem (and others).
what happens if we don’t have enough memory to store all of our programs’ instructions and data?

page table entries contain additional bits that help us deal with this problem (and others)

present (P) bit: is the page currently in memory?
what happens if we don’t have enough memory to store all of our programs’ instructions and data?

Page table entries contain additional bits that help us deal with this problem (and others)

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

if the page is not in memory, the access triggers an exception (known as “page fault” in this case), which the OS handles
what happens if we don’t have enough memory to store all of our programs’ instructions and data?

Page table entries contain additional bits that help us deal with this problem (and others)

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

If the page is not in memory, the access triggers an exception (known a “page fault” in this case), which the OS handles.

This also answers the question of why PTEs are 32 bits, not 20: they store information beyond the page number.
interlude: handling exceptions
(such as page faults)

this idea will remain relevant, as we are going to find that there are quite a few exceptions for the OS to handle
this idea will remain relevant, as we are going to find that there are quite a few exceptions for the OS to handle.

the operating system’s **kernel** manages page faults and other **exceptions**
this idea will remain relevant, as we are going to find that there are quite a few exceptions for the OS to handle

the operating system’s **kernel** manages page faults and other **exceptions**

```c
// special instruction that calls the exception handler for exception x
exception(x):
    // switch from user mode to kernel mode
    // call the handler for this particular exception
    // switch from kernel mode to user mode
```
the operating system’s **kernel** manages page faults and other **exceptions**

```c
// special instruction that calls the exception handler for exception x
exception(x):
  U/K bit = K
  // call the handler for this particular exception
  U/K bit = U
```

the processor stores a **user/kernel (U/K) bit** that indicates whether its operating in user mode or kernel mode. this bit helps the processor control access to certain kernel-specific actions
the operating system’s **kernel** manages page faults and other **exceptions**

```c
// special instruction that calls the exception handler for exception x
exception(x):
    U/K bit = K
    call handlers[x]
    U/K bit = U
```

the processor stores a **user/kernel (U/K) bit** that indicates whether its operating in user mode or kernel mode. this bit helps the processor control access to certain kernel-specific actions

each handler is different. as an example, the page-fault handler would take care of bringing the requested page into memory
what happens if we don’t have enough memory to store all of our programs’ instructions and data? page table entries contain additional bits that help us deal with this problem (and others)

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

if the page is not in memory, the access triggers an exception (known as a “page fault” in this case), which the kernel handles.
what happens if a program tries to write to memory that it doesn’t have write-access to?
what happens if a program tries to write to memory that it doesn't have write-access to?

after all, it's conceivable that we want program\(_1\) to be able to read some data, but not to modify it
what happens if a program tries to write to memory that it doesn't have write-access to?

after all, it's conceivable that we want program_1 to be able to read some data, but not to modify it.

read/write (R/W) bit: is the program allowed to write to this address?
what happens if a program tries to write to memory that it doesn’t have write-access to?

after all, it's conceivable that we want program\textsubscript{1} to be able to read some data, but not to modify it

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

if the program doesn’t have write-access to this page (and is trying to write to it), the access triggers an *exception*, which the kernel handles
what happens if a program tries to access memory that only the kernel should have access to?
what happens if a program tries to access memory that only the kernel should have access to?

we need to enforce modularity between programs and the kernel, not just between programs

physical page number

main memory

instructions and data for program

instructions and data for program

page table for program

page table for program

physical page number

0xFFFFFFFF

(2^32-1)

0xF0000000

0xE000000

0x007A1200

0x00309000

0x00000000
what happens if a program tries to access memory that only the kernel should have access to?

user/supervisor (U/S) bit: is the program allowed to access this address?

we need to enforce modularity between programs and the kernel, not just between programs
what happens if a program tries to access memory that only the kernel should have access to?

we need to enforce modularity between programs and the kernel, not just between programs

user/supervisor (U/S) bit: is the program allowed to access this address?

if not, the access triggers an exception, which the kernel handles
what happens if a program tries to access memory that only the kernel should have access to?

we need to enforce modularity between programs and the kernel, not just between programs

user/supervisor (U/S) bit: is the program allowed to access this address?

if not, the access triggers an exception, which the kernel handles

without this last piece, a determined program could still attempt to circumvent modularity by doing things such as modifying the page-table registers
CPU1 (used by program1)

memory management unit (MMU)

0x00002148

PTR1 0x007A1200
PTR2 0x003D0900

main memory

instructions and data for program1

0xFFFFFFF

(2^32-1)

0xF0000000

0xE000000

0x007A1200

0x003D0900

0x00000000

instructions and data for program2

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

memory management unit (MMU)

main memory

instructions and data for program\textsubscript{1}

instructions and data for program\textsubscript{2}

... page table for program\textsubscript{1}

... page table for program\textsubscript{2}

\[\text{2}^{20} \text{ virtual addresses each mapping to a 32-bit page-table entry (PTE)} \rightarrow \text{4MB to store this table}\]
**Performance issue #1:** Page tables are allocated contiguously in memory so that access into them is extremely fast; this means that every page table is 4MB, even if the program only needs to make a few memory accesses.

2^{20} virtual addresses each mapping to a 32-bit page-table entry (PTE) → 4MB to store this table.
multilevel page tables often use less space
**multilevel** page tables often use less space

With multilevel page tables, the MMU interprets this address as referring to a *series* of page tables instead of just a single page table.
multilevel page tables often use less space

CPU₁ (used by program₁)

memory management unit (MMU)

main memory

EIP

0x02013148

0x02013148

PTR₁ 0x007A1200
PTR₂ 0x003D0900

0xFFFFFFFF

0x00000000

0x003D0900

0x007A1200

0xE000000

0xF000000

(2^{32}-1)

0x00000000

0x003D0900

0x007A1200

0xF000000

0xE000000

0x00000000

instructions and data for program₁

instructions and data for program₂

page table for program₁

page table for program₂
**multilevel** page tables often use less space

**CPU** (used by program₁)

<table>
<thead>
<tr>
<th>EIP</th>
<th>0x02013148</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

**memory management unit (MMU)**

0x02013148

PTR₁ 0x007A1200
PTR₂ 0x003D0900

**main memory**

- instructions and data for program₁
- instructions and data for program₂
- page table for program₁
- page table for program₂

0x00000000
0x003D0900
0x007A1200
0xE000000
0xF0000000
0xFFFFFFFF

*(2³²-1)*

**this table** is the only one that will be allocated initially, and the top **eight** bits index into it. so it has **2⁸** entries, not **2²⁰**
**multilevel** page tables often use less space

**CPU** (used by program 1)

- **EIP**
  - \(0x02013148\)
  - \(31\) 0

**Memory Management Unit (MMU)**

- **Base Address**: \(0x02013148\)
- **Page Table Entries**:
  - **PTR1**: \(0x007A1200\)
  - **PTR2**: \(0x003D0900\)

**Main Memory**

- **Instructions and Data for Program 1**
- **Instructions and Data for Program 2**
- **Page Table for Program 1**
- **Page Table for Program 2**

---

- **0x02** indexes into this table

**This table** is the only one that will be allocated initially, and the top eight bits index into it. So it has \(2^8\) entries, not \(2^{20}\)
multilevel page tables often use less space

CPU₁ (used by program₁)

memory management unit (MMU)

main memory

EIP

0x02013148

31 0

0x02013148

PTR₁ 0x007A1200
PTR₂ 0x003D0900

PTR₁ 0x007A1200
PTR₂ 0x003D0900

0xFFFFFFFF

0x00000000

0x003D0900

0x007A1200

0xE000000

0xF000000

0x007A1200

0x003D0900

0xFFFFFFF (2³²-1)

0xF0000000

0xE000000

0x00000000

0x007A1200

0x003D0900

0x00000000

0x02 indexes into this table

row 0x02 points to another table

this table is the only one that will be allocated initially, and the top eight bits index into it. so it has 2⁸ entries, not 2²⁰
**multilevel** page tables often use less space

**CPU** (used by program 1)

- EIP: $0x02013148$
- 31 0

**Memory Management Unit (MMU)**

- Page table for program 1
- Page table for program 2

**Main Memory**

- Instructions and data for program 1
- Instructions and data for program 2

This table is the only one that will be allocated initially, and the top eight bits index into it. So it has $2^8$ entries, not $2^{20}$.

2^8 entries

Row $0x02$ points to another table.

Row $0x02$ indexes into this table.
multilevel page tables often use less space

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

```
<table>
<thead>
<tr>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x02013148</td>
</tr>
</tbody>
</table>
```

memory management unit (MMU)

```
0x02013148
```

```
PTR\textsubscript{1} 0x007A1200
PTR\textsubscript{2} 0x003D0900
```

main memory

```
instructions and data for program\textsubscript{1}

instructions and data for program\textsubscript{2}

... 

page table for program\textsubscript{1}

page table for program\textsubscript{2}
```

```
0xFFFFFFFF
0x00000000
0x003D0900
0x007A1200
0xF0000000
0xE000000
0x007A1200
```

```
(2^{32}-1)
```

0x00000000

```
0x00000000
```

2\textsuperscript{8} entries

```
0x01 indexes into this table

0x02 indexes into this table
```

```
this table is the only one that will be allocated initially, and the top eight bits index into it. so it has 2\textsuperscript{8} entries, not 2^{20}
```

```
row 0x02 points to another table
```

Katrina LaCurts | lacurts@mit.edu | 6.033 2022
multilevel page tables often use less space

- CPU \(_1\) (used by program \(_1\))
- memory management unit (MMU)
- main memory

**CPU**

<table>
<thead>
<tr>
<th>EIP</th>
<th>0x02013148</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

**Memory Management Unit (MMU)**

- **EIP**
  - 0x02013148
- PTR\(_1\) 0x007A1200
- PTR\(_2\) 0x003D0900
- **Instructions and data for program**
  - Program 1
  - Program 2

**Main Memory**

- 0xFFFFFFFF
- 0x00000000
- 0x003D0900
- 0x007A1200
- 0xE000000
- 0xF000000
- 0x007A1200

**Diagram Notes**

- 0x01 indexes into this table
- 2\(^8\) entries
- row 0x01 points to another table
- 0x02 indexes into this table
- row 0x02 points to another table
- This table is the only one that will be allocated initially, and the top eight bits index into it. So it has 2\(^8\) entries, not 2\(^{20}\)
multilevel page tables often use less space

CPU1 (used by program1)

memory management unit (MMU)

main memory

EIP

0x02013148

0x02013148

PTR1 0x007A1200
PTR2 0x0003D0900

0x02013148

instructions and data for program1

instructions and data for program2

0xFFFFFFFF

... 0x00000000 0x003D0900 0xF0000000 0xE000000 0x007A1200 (2^{32}-1)

... 0x007A1200 0x0003D0900 0x00000000

2^4 entries

0xF0110

... 0x01 indexes into this table

0x02 indexes into this table

... 0x02 points to another table

this table is the only one that will be allocated initially, and the top eight bits index into it. so it has 2^8 entries, not 2^{20}

row 0x01 points to another table

row 0x02 points to another table

Katrina LaCurts | lacurts@mit.edu | 6.033 2022
**multilevel** page tables often use less space

---

**CPU₁** (used by program₁)

<table>
<thead>
<tr>
<th>EIP</th>
<th>0x02013148</th>
</tr>
</thead>
</table>

row **0x3** contains the physical page number

**0xF0110**... 2⁴ entries

**0xF0110**

---

**memory management unit (MMU)**

<table>
<thead>
<tr>
<th>EIP</th>
<th>0x02013148</th>
</tr>
</thead>
</table>

---

**main memory**

- instructions and data for program₁
- instructions and data for program₂
- ...  
- page table for program₁  
- page table for program₂

---

**0x007A1200**  
**PTR₁**

---

**0x003D0900**  
**PTR₂**

---

**0xFFFFFFFF**  
**0x00000000**  
**0x003D0900**

---

**this table** is the only one that will be allocated initially, and the top **eight** bits index into it. So it has **2⁸** entries, not **2²⁰**
multilevel page tables often use less space.
multilevel page tables often use less space.

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

- **EIP**
  - `0x02013148`

**Memory management unit (MMU)**

- `0x02013148` points to `0xF0110148`
- `0xF0110148` contains a page table for program<sub>1</sub>
- `0xF0110148` contains a page table for program<sub>2</sub>

**Main memory**

- `0xFFFFFFFF`
- `0x00000000`
- `0x003D0900`
- `0xF0000000`
- `0xE000000`
- `0x007A1200`
- `0x0FF00000`
- `0x0E00000`
- `0x007A1200`
- `0x003D0900`
- `0x00000000`

- Pointer (PTR) to `0x007A1200`
- Pointer (PTR) to `0x003D0900`

**Notes:**

- Row `0x3` contains the physical page number.
- Row `0x1` points to another table.
- Row `0x2` points to another table.
- This table is the only one that will be allocated initially, and the top eight bits index into it. So it has `2^8` entries, not `2^20`.

(I used `8/8/4` in this example, but you can generalize to `M/N/P`.)

Katrina LaCurts | lacurts@mit.edu | 6.033 2022
**multilevel** page tables often use less space

---

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

- **EIP**
  - 0x02013148
  - 31 0

**memory management unit (MMU)**

- 0x02013148
- 0xF0110148
- PTR$_1$ 0x007A1200
- PTR$_2$ 0x003D0900

**main memory**

- instructions and data for program$_1$
- instructions and data for program$_2$
- ... page table for program$_1$
- page table for program$_2$

---

- 2$^4$ entries
  - level 3 table
- 2$^8$ entries
  - level 2 table
- 2$^8$ entries
  - level 1 table

---

(if the program never accesses a virtual memory address starting with 0x03 (say), no **level 2 table** will be allocated corresponding to row 0x03 in the **level 1 table**)

(I used 8/8/4 in this example, but you can generalize to M/N/P)
multilevel page tables often use less space, at the expense of more table look-ups and more exceptions (to allocate additional tables)

If the program never accesses a virtual memory address starting with 0x03 (say), no level 2 table will be allocated corresponding to row 0x03 in the level 1 table.
CPU1 (used by program1)

memory management unit (MMU)

EIP
0x02013148
31 0

0x02013148

0xF0110148

PTR1 0x007A1200
PTR2 0x003D0900

main memory

instructions and data for program1

0xFFFFFFFF

0x00000000

0x003D0900

0xF0000000

0xE000000

0x007A1200

0xF0110148

2^4 entries
inner table

2^8 entries
first-outer table

2^8 entries
second-outer table

instructions and data for program2

...
**Performance Issue #2:** Looking up the same piece of data over and over again takes time; can we make it faster?
performance issue #2: looking up the same piece of data over and over again takes time; can we make it faster?

CPU₁ (used by program₁)

memory management unit (MMU)

main memory

CPU

EIP

0x02013148

31 0

memory management unit (MMU)

0x02013148

0xF0110148

PTR₁ 0x007A1200

PTR₂ 0x003D0900

main memory

instructions and data for program₁

instructions and data for program₂

... page table for program₁

page table for program₂

0xFFFFFFFF

0x00000000

0x003D0900

0xF0110148

0x007A1200

2⁴ entries inner table

2⁸ entries first-outer table

2⁸ entries second-outer table

0x02013148

0xF0110148

yes. caches are involved in a variety of places here, to (in theory) make common look-ups faster. you’ve also seen caching in the context of DNS, now.
operating systems enforce modularity on a single machine

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory

   → virtualize memory

2. programs should be able to communicate with each other

   → assume they don’t need to (for today)

3. programs should be able to share a CPU without one program halting the progress of the others

   → assume one program per CPU (for today)
operating systems enforce modularity on a single machine

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory
   → virtualize memory

2. programs should be able to communicate with each other
   → assume they don’t need to (for today)

3. programs should be able to share a CPU without one program halting the progress of the others
   → assume one program per CPU (for today)

the primary technique that an operating system uses to enforce modularity is virtualization. some components are difficult to virtualize (e.g., the disk); for those, the operating system presents abstractions
operating systems enforce modularity on a single machine via virtualization and abstraction
**operating systems** enforce modularity on a single machine via **virtualization** and **abstraction**

you’ll talk much more about abstractions during the recitations on UNIX; designing good abstractions is part of designing a good operating system
**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**, and there are a number of **performance issues** to take into account.

you’ll talk much more about abstractions during the recitations on UNIX; designing good abstractions is part of designing a good operating system
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, and there are a number of performance issues to take into account.

you’ll talk much more about abstractions during the recitations on UNIX; designing good abstractions is part of designing a good operating system.
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, and there are a number of performance issues to take into account.

the kernel handles any exceptions triggered in this process; protecting the kernel from user programs is just as important as protecting user programs from each other.

you’ll talk much more about abstractions during the recitations on UNIX; designing good abstractions is part of designing a good operating system.