# 6.033 in the news on Mars

## Tech Specs

<table>
<thead>
<tr>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Radiation-hardened central processor with PowerPC 750 Architecture: a BAE RAD 750</td>
</tr>
<tr>
<td>- Operates at up to 200 megahertz speed, 10 times the speed in Mars rovers Spirit and Opportunity’s computers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 2 gigabytes of flash memory (~8 times as much as Spirit or Opportunity)</td>
</tr>
<tr>
<td>- 256 megabytes of dynamic random access memory</td>
</tr>
<tr>
<td>- 256 kilobytes of electrically erasable programmable read-only memory</td>
</tr>
</tbody>
</table>

https://mars.nasa.gov/mars2020/spacecraft/rover/brains/

It generally takes about 5 to 20 minutes for a radio signal to travel the distance between Mars and Earth, depending on planet positions. Using orbiters to relay messages is beneficial because they are much closer to Perseverance than the Deep Space Network (DSN) antennas on Earth. The mass- and power-constrained rover can achieve high data rates of up to 2 megabits per second on the relatively short-distance relay link to the orbiters overhead. The orbiters then use their much larger antennas and transmitters to relay that data on the long-distance link back to Earth.

https://mars.nasa.gov/mars2020/spacecraft/rover/communications/

https://mars.nasa.gov/mars2020/
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

```
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

stub

load("kaws.com/buy.html?llama")
```

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

stub

load("kaws.com/buy.html?llama")

```

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

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

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: every program to be able to access 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:** every program to be able to access 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:** every program to be able to access 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$_1$)
**what we want:** every program to be able to access 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
...
```

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

```
**what we want:** every program to be able to access 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

$\text{CPU}_1$ (used by program 1)

- EIP
- \begin{tabular}{l}
  31  \\
  0 
\end{tabular}

$\text{CPU}_2$ (used by program 2)
**what we want:** every program to be able to access 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:** every program to be able to access 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 1 diagram](image)

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

![CPU 2 diagram](image)

---

**main memory**

- $0x00000000$
- $0xF0000000$
- $0xE0000000$
- $0xFFFFFFFF$

- Instructions and data for program$_1$
- Instructions and data for program$_2$
**what we want:** every program to be able to access 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})

- EIP
- 31 0

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

---

**main memory**

- Instructions and data for program\textsubscript{1}
  - 0xFFFFFFFF (2\textsuperscript{32} - 1)
  - 0xF0000000
  - 0xE000000
  - ...

- Instructions and data for program\textsubscript{2}

- 0x00000000
**what we want:** every program to be able to access 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**: every program to be able to access 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)**

**main memory**

```
0xffffffff
0xf0000000
0xe0000000
0x00000000
```
**what we want:** every program to be able to access 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:** every program to be able to access 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)**

- 0x00002148

**main memory**

- Instructions and data for program₁
- Instructions and data for program₂
- ... 
- Table for program₁
- Table for program₂

---

Katrina LaCurts | lacurts@mit.edu | 6.033 2021
**what we want:** every program to be able to access 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:** every program to be able to access 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
```

**main memory**

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

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

- 0xFF035113
- 0xF27A9B77
- 0xF0110048
- 0xF8887881
- …
**what we want:** every program to be able to access 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)

- EIP: 0x00002148

**CPU₂** (used by program₂)

**memory management unit (MMU)**

- 0x00002148

**main memory**

- Instructions and data for program₁
- Instructions and data for program₂
- Table for program₁
- Table for program₂

---

**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:** every program to be able to access 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 $\rightarrow$ 16GB to store this table
**what we want:** every program to be able to access 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:** every program to be able to access 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>
<th>0x00002148</th>
</tr>
</thead>
</table>

31 0

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

---

memory management unit (MMU)

- EIP: 0x00002148

---

main memory

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

---

<table>
<thead>
<tr>
<th>0xFFFF0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF0000000</td>
</tr>
<tr>
<td>0xE0000000</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
</tbody>
</table>
**what we want:** every program to be able to access 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

**memory management unit (MMU)**

**main memory**

- Instructions and data for program$_1$
- Instructions and data for program$_2$
- Page table for program$_1$
- Page table for program$_2$
**what we want:** every program to be able to access 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 program₁)

**CPU**₂ (used by program₂)

---

**EIP**

$0x000002148$

---

**memory management unit (MMU)**

$0x000002148$

---

**main memory**

**instructions and data for program₁**

**instructions and data for program₂**

---

**PTR**

$0x007A1200$

$0x003D0900$

---

**0xFFFFFFFF**

(2$^{32}$ - 1)
**what we want:** every program to be able to access 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:** every program to be able to access 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)

---

**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:** every program to be able to access 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)
- physical page number: 0xF0110

**main memory**

- instructions and data for program₁
  - PTR₁: 0x007A1200
  - PTR₂: 0x003D0900
- page table for program₁
- page table for program₂

---

**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:** every program to be able to access 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>
<th>0x00002148</th>
</tr>
</thead>
</table>

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

---

**EIP:** 0x00002148

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

**physical page number:** 0xF0110

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

---

**memory management unit (MMU)**

** PTR$_1$:** 0x007A1200

** PTR$_2$:** 0x003D0900

---

**main memory**

- instructions and data for program$_1$
  - 0x0FF035
  - 0xF27A9
  - 0xF0110
  - 0xF8887

- page table for program$_1$
  - 0x00000000
  - 0x0E000000
  - 0x007A1200

- instructions and data for program$_2$
  - 0xF000000
  - 0xE000000
  - 0x003D0900

- page table for program$_2$
  - 0x00000000

---

**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)

---

Katrina LaCurts | lacurts@mit.edu | 6.033 2021
**what we want:** every program to be able to access 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$_2$)

**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:** every program to be able to access 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})

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

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

---

**memory management unit (MMU)**

- **virtual page number**: $0x00002$
  - (top 20 bits)
- **physical page number**: $0xF0110$
- **offset**: $0x148$
  - (bottom 12 bits)

**main memory**

- **instructions and data for program\textsubscript{1}**
  - $0x007A1200$
  - $0x000000$
  - $0xE00000$
  - $0x007A1200$
  - $0x000000$
- **instructions and data for program\textsubscript{2}**
  - $0x003D0900$
  - $0x000000$
- **page table for program\textsubscript{1}**
  - $0xFF035$
  - $0xF27A9$
  - $0xF8887$
- **page table for program\textsubscript{2}**
  - $0x000000$
  - $0x0003D0900$

---

**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:** every program to be able to access 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>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</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)

---

2$^{20}$ virtual page numbers each mapping to a 32-bit page-table entry (PTE) → 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?

physical page number

31 12 11 0

main memory

0xFFFFFFFF (2^{32}-1)

0xF0000000

0xE000000

0x007A1200

0x003D0900

0x00000000

instructions and data for program_1

page table for program_1

instructions and data for program_2

page table for program_2

...
what happens if we don’t have enough memory to store all of our programs’ instructions and data?

main memory

<table>
<thead>
<tr>
<th>Instructions and data for program 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions and data for program 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Page table for program 1</td>
</tr>
<tr>
<td>Page table for program 2</td>
</tr>
</tbody>
</table>

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
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

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

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**, which indicates whether it’s operating in user mode or kernel mode. this bit helps the processor control access to certain kernel-specific actions
interlude: handling exceptions
(such as page faults)

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

```
// 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**, which indicates whether it’s 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 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\textsubscript{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$\text{\textsubscript{1}}$ 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.
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?
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**

---

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

memory management unit (MMU)

main memory

- instructions and data for program\_1
- instructions and data for program\_2
- page table for program\_1
- page table for program\_2

- 0xFFFFF\_FFFF:\(2^{32}-1\)
- 0xF0000000
- 0xE000000
- 0x007A1200
- 0x003D0900
- 0x00000000

- EIP: 0x00002148
- PTR\_1: 0x007A1200
- PTR\_2: 0x003D0900
**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 need to make a few memory accesses.

2¹⁰ virtual addresses each mapping to a 32-bit page-table entry (PTE) → 4MB to store this table
Hierarchical (or “multilevel”) page tables potentially use less space.
hierarchical (or “multilevel”) page tables potentially use less space
“hierarchical” (or “multilevel”) page tables potentially use less space.

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 $2^{20}$. 

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

**memory management unit (MMU)**

**main memory**
**hierarchical** (or “multilevel”) page tables potentially use less space

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

- EIP: 0x02013148

**memory management unit (MMU)**

- Page table for program\(_1\)
  - 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\(_2\))

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\(^{20}\).
Hierarchical (or “multilevel”) page tables potentially use less space.

- **CPU** (used by program₁)
  - EIP: 0x02013148
  - 31 0

- **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

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

  - Row 0x02 indexes into this table
  - Row 0x02 points to another table
hierarchical (or “multilevel”) page tables potentially use less space

CPU₁ (used by program₁)

EIP 0x02013148
31 0

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

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

(2³²-1)

2⁸ entries

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

0x02 indexes into this table

row 0x02 points to another table

Katrina LaCurts | lacurts@mit.edu | 6.033 2021
Hierarchical (or “multilevel”) page tables potentially use less space.

CPU (used by program1)

Memory management unit (MMU)

Main memory

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.
Hierarchical (or “multilevel”) page tables potentially use less space.

CPU₁ (used by program₁)

EIP

0x02013148

31 0

Memory management unit (MMU)

0x02013148

0x007A1200

0x003D0900

Main memory

Instructions and data for program₁

Instructions and data for program₂

Page table for program₁

Page table for program₂

$0xFFFFFFFF$ - $0x00000000$

$0x003D0900$ - $0x007A1200$

$0x00000000$ - $0xE000000$

$0xF000000$ - $0xFFFFFFFF$

$2^{32} - 1$

Hierarchical (or “multilevel”) page tables potentially use less space. 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 $2^{20}$. Each row in this table points to another table.
Hierarchical (or “multilevel”) page tables potentially use less space.
Hierarchical (or “multilevel”) page tables potentially use less space

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

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

Row 0x3 contains the physical page number

0xF0110

2<sup>4</sup> entries

0x007A1200

Row 0x01 points to another table

0x01

2<sup>8</sup> entries

Main memory

<table>
<thead>
<tr>
<th>0xFFFFFFFF (2&lt;sup&gt;32&lt;/sup&gt; - 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF0000000</td>
</tr>
<tr>
<td>0xE000000</td>
</tr>
<tr>
<td>0x003D0900</td>
</tr>
<tr>
<td>0x007A1200</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
</tbody>
</table>

This table is the only one that will be allocated initially, and the top eight bits index into it. So it has 2<sup>8</sup> entries, not 2<sup>20</sup>

memory management unit (MMU)

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

0x007A1200

2<sup>8</sup> entries

0x003D0900

Row 0x02 points to another table

0x01

This table is the only one that will be allocated initially, and the top eight bits index into it. So it has 2<sup>8</sup> entries, not 2<sup>20</sup>
**Hierarchical** (or “multilevel”) page tables potentially use less space.

- **CPU** (used by program 1):
  - EIP: 0x02013148
  - 31
  - 0

- **Memory management unit (MMU)**:
  - 0x02013148
  - 0xF0110148
  - 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:

Row 0x3 contains the physical page number.

2^4 entries

Row 0x01 points to another table.

2^8 entries

Row 0x01 points to another 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.

Katrina LaCurts | lacurts@mit.edu | 6.033 2021
Hierarchical (or “multilevel”) page tables potentially use less space compared to flat page tables. In this diagram, we see a CPU with an EIP (Effective Instruction Pointer) at 0x02013148. This EIP is used by program 1.

The memory management unit (MMU) handles memory requests and maintains page tables for each program. The main memory contains instructions and data for programs 1 and 2. Each program has its own page table.

- The page table for program 1 is at 0xF0110148 and contains 2^8 entries. The top eight bits index into this table, making it the only one that will be allocated initially. It has 2^8 entries, not 2^20.
- The page table for program 2 is at 0x00000000 and contains 2^4 entries.
- Instructions and data for program 1 are located at 0x00000000, and instructions and data for program 2 are located at 0x007A1200.

In this example, I used 8/8/4, but you can generalize to M/N/P.
Hierarchical (or “multilevel”) page tables potentially use less space.

If the program never accesses a virtual memory address starting with 0x03 (say), no first-outer table will be allocated corresponding to row 0x03 in the second-outer table.

(I used 8/8/4 in this example, but you can generalize to M/N/P.)
Hierarchical (or “multilevel”) page tables potentially 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 first-outer table will be allocated corresponding to row 0x03 in the second-outer table.
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?

**CPU** (used by program_1)

- **EIP**: 0x02013148

**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

**Diagrams:**

- **2^4 entries** inner table
- **2^8 entries** first-outer table
- **2^8 entries** second-outer table

---

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.
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.

amount of storage used, speed of access
**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.