6.033 in the news on Mars

Tech Specs

| Processor | Radiation-hardened central processor with PowerPC 750 Architecture: a BAE RAD 750  
| Memory | 2 gigabytes of flash memory (~8 times as much as Spirit or Opportunity)  
| | 256 megabytes of dynamic random access memory  
| | 256 kilobytes of electrically erasable programmable read-only memory |

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/brains/

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

**today:** what if we **don’t** want to put each module on a separate 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

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
**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** (used by program)  
EIP: 0x000002148

**memory management unit (MMU)**

---

**CPU** (used by program)  

---

**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

---

**main memory**

---

**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$_1$)

```
CPU
EIP 0x00002148
31 0
```

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

**physical page number:** 0xF0110

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

---

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

---

2$^{20}$ 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)

---

**main memory**

```
0xFFFFFFFF
0xF0000000
0xE0000000
0x007A1200
0x003D0900
```

---

---

Katrina LaCurts | lacurts@mit.edu | 6.033 2021
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?

main memory

<table>
<thead>
<tr>
<th>Physical page number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0x003D0900</td>
</tr>
<tr>
<td>0xF0000000</td>
</tr>
<tr>
<td>0xE0000000</td>
</tr>
<tr>
<td>0x007A1200</td>
</tr>
<tr>
<td>(2^{32} - 1)</td>
</tr>
</tbody>
</table>

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.

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

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

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

// 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**

```
// 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 a program tries to write to memory that it doesn’t have write-access to?

After all, it’s conceivable that we want program1 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?

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

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

---

2^{20} 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**

**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>0</td>
</tr>
</tbody>
</table>

row 0x3 contains the physical page number

<table>
<thead>
<tr>
<th>0xF0110</th>
</tr>
</thead>
</table>

... 2<sup>4</sup> entries

<table>
<thead>
<tr>
<th>0xF0110</th>
</tr>
</thead>
</table>

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

0x02013148

memory management unit (MMU)

<table>
<thead>
<tr>
<th>0xF0110148</th>
</tr>
</thead>
</table>

0xF0110148

<table>
<thead>
<tr>
<th>PTR&lt;sub&gt;1&lt;/sub&gt;</th>
<th>0x007A1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTR&lt;sub&gt;2&lt;/sub&gt;</td>
<td>0x0003D0900</td>
</tr>
</tbody>
</table>

main memory

<table>
<thead>
<tr>
<th>instructions and data for program&lt;sub&gt;1&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF_FFFF (2&lt;sup&gt;32&lt;/sup&gt;-1)</td>
</tr>
<tr>
<td>0xF000000</td>
</tr>
<tr>
<td>0xE000000</td>
</tr>
<tr>
<td>0x007A1200</td>
</tr>
<tr>
<td>0x0003D0900</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instructions and data for program&lt;sub&gt;2&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>page table for program&lt;sub&gt;1&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x007A1200</td>
</tr>
<tr>
<td>0x0003D0900</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>page table for program&lt;sub&gt;2&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
</tbody>
</table>

**I used 8/8/4 in this example, but you can generalize to M/N/P**

**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, at the expense of more table look-ups and more exceptions (to allocate additional tables).

---

**CPU** (used by program 1)

<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
- **Address**: 0xF0110148
- **Page Table**
  - **1.** First-outer table
  - **2.** Second-outer table

**Main Memory**

- **Instructions and Data for Program 1**: 0x007A1200
- **Instructions and Data for Program 2**: 0x003D0900

---

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

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 other’s 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**.

**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.