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

**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
   
   virtualize 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
   
   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 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></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00002148</td>
<td></td>
</tr>
</tbody>
</table>

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

The MMU is going to use program$_1$’s table to *translate* a virtual address from program$_1$ into a physical address.

---

**main memory**

- instructions and data for program$_1$
- instructions and data for program$_2$
- table for program$_1$
- table 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 program₁)

- EIP
- 0x00002148
- 31 0

**CPU**₂ (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 → 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$_1$)

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

---

**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$^{20}$ 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:

*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?

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.

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

```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 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 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? 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?

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.
**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

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

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

\begin{align*}
\text{EIP} & : 0x02013148 \\
31 & : 31 \\
30 & : 0
\end{align*}

row \texttt{0x3} contains the physical page number

\begin{align*}
0xF0110148 & \rightarrow 0x02013148 & 0xF0110148
\end{align*}

\begin{itemize}
\item \texttt{0x01} indexes into this table
\item \texttt{0x02} indexes into this table
\item \texttt{0x03} contains the physical page number
\end{itemize}

\begin{itemize}
\item \texttt{0xF0110} \begin{itemize}
\item \texttt{2^{4}} entries
\item row \texttt{0x01} points to another table
\end{itemize}
\item \texttt{0x0213148} \begin{itemize}
\item \texttt{2^{8}} entries
\item row \texttt{0x02} points to another table
\end{itemize}
\item \texttt{0x018100} \begin{itemize}
\item \texttt{2^{8}} entries
\end{itemize}
\end{itemize}

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

(I used \texttt{8/8/4} in this example, but you can generalize to \texttt{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)

(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?

---

**CPU**

- EIP: 0x02013148

**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

---

**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

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.
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.