Virtual Memory

Process Abstraction, Part 2: Private Address Space

**Motivation:** why not direct physical memory access?

**Address translation** with pages

**Optimizing translation:** translation lookaside buffer

**Extra benefits:** sharing and protection

Memory as a contiguous array of bytes is a lie! Why?
Problems with Physical Addressing

Fine for small embedded systems without processes.

Elevators, microwaves, radio-powered devices, ...

What about larger systems?

With many processes?
Problem 1: Memory Management

Main memory

Process 1
Process 2
Process 3
...
Process n

<table>
<thead>
<tr>
<th>stack</th>
<th>heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>globals</td>
</tr>
</tbody>
</table>

Also:

Context switches must swap out entire memory contents. Isn't that expensive?
Problem 2: Capacity

64-bit addresses can address several exabytes
(18,446,744,073,709,551,616 bytes)

Physical main memory offers a few gigabytes
(e.g. 8,589,934,592 bytes)

1 virtual address space per process,
with many processes...

(Actually, it’s smaller than that dot compared to virtual memory.)
Problem 3: Protection

Problem 4: Sharing
Solution: Virtual Memory (address *indirection*)

Private virtual address space per process.

Single physical address space managed by OS/hardware.
Indirection
(it's everywhere!)

Direct naming

Indirect naming

What if we move *Thing*?
Tangent: **Indirection everywhere**

- Pointers
- Constants
- Procedural abstraction
- Domain Name Service (DNS)
- Dynamic Host Configuration Protocol (DHCP)
- Phone numbers
- 911
- Call centers
- Snail mail forwarding
- ...

“Any problem in computer science can be solved by adding another level of indirection.”

—David Wheeler, inventor of the subroutine, or Butler Lampson

Another Wheeler quote? "Compatibility means deliberately repeating other people's mistakes."
Virtual Addressing and Address Translation

Memory Management Unit
translates virtual address to physical address

Physical addresses are invisible to programs.
Page-based Mapping

both address spaces divided into fixed-size, aligned *pages*
page size = power of two

Map virtual pages onto physical pages.

Some virtual pages do not fit! Where are they stored?
Some virtual pages do not fit! Where are they stored?

Virtual Memory Address Space

Virtual Page 0
Virtual Page 1
Virtual Page 2
Virtual Page 3

Virtual address space usually much larger than physical address space

Physical Memory Address Space

Physical Page 0
Physical Page 1

1. On disk (if used)

2. Nowhere! (if not yet used)
Virtual Memory: cache for disk?

Example system

Not drawn to scale
Design for a Slow Disk: Exploit Locality

- Physical Memory Address Space
  - Physical Page 0
  - Physical Page 1
  - Physical Page 2
  - Physical Page 3
  - ... 
  - Physical Page $2^{m} - 1$

- Virtual Memory Address Space
  - Virtual Page 0
  - Virtual Page 1
  - Virtual Page 2
  - Virtual Page 3
  - ... 
  - Virtual Page $2^{n} - 1$

- Page size?
- Associativity?
- Replacement policy?
- Write policy?
Address Translation

What happens in here?

CPU Chip

Virtual address (VA) 4100

MMU

Physical address (PA) 4

Main memory

0:
1:
2:
3:
4:
5:
6:
7:
8:
...
M-1:

Data
Page Table

array of *page table entries* (PTEs)

mapping virtual page to where it is stored

Physical pages (Physical memory)

<table>
<thead>
<tr>
<th>VP 1</th>
<th>VP 2</th>
<th>VP 4</th>
<th>VP 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 0</td>
<td>PP 3</td>
<td>PP 3</td>
<td>PP 3</td>
</tr>
</tbody>
</table>

Swap space (Disk)

| VP 3 | VP 6 |

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Memory resident, managed by HW (MMU), OS

How many page tables are in the system?
Address Translation with a Page Table

**Virtual address** (VA)

- Virtual page number (VPN)
- Virtual page offset (VPO)

**Page table**

- Valid
- Physical page number (PPN)

**Physical address** (PA)

- Physical page number (PPN)
- Physical page offset (PPO)

If so: **Page Hit**

Page table base register (PTBR)

Base address of current process's page table

Virtual page mapped to physical page?
Page **Hit:** virtual page in memory

Virtual Page Number

<table>
<thead>
<tr>
<th>Physical Page Number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
</tr>
<tr>
<td>0    null</td>
</tr>
<tr>
<td>1    PP 0</td>
</tr>
<tr>
<td>1    PP 1</td>
</tr>
<tr>
<td>0    On disk</td>
</tr>
<tr>
<td>1    PP 3</td>
</tr>
<tr>
<td>0    null</td>
</tr>
<tr>
<td>0    On disk</td>
</tr>
<tr>
<td>1    PP 2</td>
</tr>
</tbody>
</table>

Physical pages (Physical memory)

Swap space (Disk)

PTE 0

PTE 7

page table
Page Fault:

Physical Page Number

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Physical Page Number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>PP 0</td>
</tr>
<tr>
<td>1</td>
<td>PP 1</td>
</tr>
<tr>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td>1</td>
<td>PP 3</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td>1</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

Swap space

(Disk)

- VP 3
- VP 6

Physical pages

(Physical memory)

- VP 1
- PP 0
- VP 2
- PP 1
- VP 7
- PP 2
- VP 4
- PP 3
Page **Fault**: exceptional control flow

Process accessed virtual address in a page that is not in physical memory.

Process

User Code

OS exception handler

movl

exception: page fault

return

Load page into memory

Returns to faulting instruction: \texttt{movl} is executed \textit{again}!
Page Fault: 1. *page not in memory*

What now? OS handles fault
Page **Fault**: 2. OS evicts another page.

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Physical Page Number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td>1</td>
<td>PP 1</td>
</tr>
<tr>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td>1</td>
<td>PP 3</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td>1</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

**Physical pages**

- VP 1
- VP 2
- VP 7
- VP 4

**Swap space**

- VP 3
- VP 6
- VP 1

**OS evicts another page.**

"Page out"
Page Fault: 3. OS loads needed page.

**Page Table**

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Physical Page Number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>On disk</td>
</tr>
<tr>
<td>1</td>
<td>PP 1</td>
</tr>
<tr>
<td>1</td>
<td>PP 0</td>
</tr>
<tr>
<td>1</td>
<td>PP 3</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td>1</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

Finally:
Re-execute faulting instruction.
Page hit!

"Page in"
Terminology

class switch

page in

page out

thrash

\{ swap \}
Address Translation: Page *Hit*

1) Processor sends virtual address to MMU (memory management unit)
2-3) MMU fetches PTE from page table in cache/memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor
Address Translation: Page Fault

1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in cache/memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler identifies victim (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction
How fast is translation?

How many physical memory accesses are required to complete one virtual memory access?

Translation Lookaside Buffer (TLB)

Small hardware cache in MMU just for page table entries e.g., 128 or 256 entries

Much faster than a page table lookup in memory.

In the running for "un/classiest name of a thing in CS"
A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Does a TLB miss require disk access?
Simple Memory System Example (small)

Addressing

14-bit virtual addresses
12-bit physical address
Page size = 64 bytes

Simulate accessing these virtual addresses on the system: 0x03D4, 0xB8F, 0x0020
## Simple Memory System Page Table

Only showing first 16 entries (out of $256 = 2^8$)

<table>
<thead>
<tr>
<th>virtual page #</th>
<th>TLB index</th>
<th>TLB tag</th>
<th>TLB Hit?</th>
<th>Page Fault?</th>
<th>physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What about a real address space? Read more in the book...
# Simple Memory System TLB

## 16 entries

### 4-way associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>–</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>04</td>
<td>–</td>
<td>0</td>
<td>0A</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>08</td>
<td>–</td>
<td>0</td>
<td>06</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

TLB ignores page offset. Why?

---

- **TLB tag**: 13 12 11 10 9 8 7 6 5 4 3 2 1 0
- **TLB index**: virtual page number
- **TLB tag**: virtual page offset
- **valid**: PPN Tag

virtual page #___  TLB index___  TLB tag ____  TLB Hit? __  Page Fault? __  physical page #: ____
## Simple Memory System Cache

16 lines
4-byte block size
Physically addressed
Direct mapped

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>1B</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Simple address space allocation

Process needs private **contiguous** address space.

Storage of virtual pages in physical pages is **fully associative**.

![Diagram of address space allocation]

**Virtual Address Spaces**

**Process 1:**
- 0: VP 1, VP 2, ...
- N-1:

**Process 2:**
- 0: VP 1, VP 2, ...
- N-1:

**Physical Address Space (DRAM)**

- 0: PP 2, PP 6, PP 8, PP 9, ...
- M-1:
Simple cached access to storage > memory

Good locality, or least "small" working set = mostly page hits

- All necessary page table entries fit in TLB
- Working set pages fit in physical memory

If combined working set > physical memory:

*Thrashing*: Performance meltdown. CPU always waiting or paging.

Full indirection quote:

“Every problem in computer science can be solved by adding another level of indirection, *but that usually will create another problem.*”
Freebies

Protection:

All accesses go through translation.

Impossible to access physical memory not mapped in virtual address space.

Sharing:

Map virtual pages in separate address spaces to same physical page (PP 6).
Memory permissions

MMU checks on every access.
Exception if not allowed.

How would you set permissions for the stack, heap, global variables, literals, code?
Summary: Virtual Memory

Programmer’s view of virtual memory
- Each process has its own private linear address space
- Cannot be corrupted by other processes

System view of virtual memory
- Uses memory efficiently (due to locality) by caching virtual memory pages
- Simplifies memory management and sharing
- Simplifies protection — easy to interpose and check permissions
- More goodies:
  - Memory-mapped files
  - Cheap fork() with copy-on-write pages (COW)
Summary: Memory Hierarchy

L1/L2/L3 Cache: Pure Hardware

Virtual Memory: Software-Hardware Co-design