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

Physical address (PA) → Main memory

Data
Problem 1: memory management

What goes where?

Process 1
Process 2
Process 3
...
Process n

stack
heap
code
globals
...

Main memory

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...

(To scale with 64-bit address space, you can't see it.)
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.

Virtual-to-physical mapping

- Virtual addresses
- Physical addresses
- Data
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

Virtual Address Space

0

Virtual Page 0

Virtual Page 1

Virtual Page 2

Virtual Page 3

Virtual Page $2^v - 1$

Physical Address Space

0

Physical Page 0

Physical Page 1

Physical Page $2^p - 1$

Physical Page $2^{m-1}$

Map virtual pages onto physical pages.

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

fixed-size, aligned *pages*
page size = power of two
Cannot fit all virtual pages! Where are the rest stored?

Virtual Memory Address Space

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

Physical Memory Address Space

Physical Page 0
Physical Page 1
...
Physical Page 2^n - 1

virtual address space usually much larger than physical address space

1. On disk if used
2. Nowhere if not (yet?) used
Virtual memory: cache for disk?

Example system

<table>
<thead>
<tr>
<th>SRAM</th>
<th>DRAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 I-cache</td>
<td>L2 unified cache</td>
<td>~500 GB</td>
</tr>
<tr>
<td>32 KB</td>
<td>~4 MB</td>
<td>solid-state &quot;flash&quot; or spinning magnetic platter.</td>
</tr>
<tr>
<td>L1 D-cache</td>
<td>Main Memory</td>
<td>~8 GB</td>
</tr>
</tbody>
</table>

Throughput: 16 B/cycle
Latency: 3 cycles

Throughput: 8 B/cycle
Latency: 14 cycles

Throughput: 2 B/cycle
Latency: 100 cycles

Throughput: 1 B/30 cycles
Latency: millions

Cache miss penalty (latency): 33x

Memory miss penalty (latency): 10,000x
Design for a slow disk: exploit locality

Virtual Memory Address Space

Physical Memory Address Space

on disk

Virtual Page 0

Virtual Page 1

Virtual Page 2

Virtual Page 3

\[ \text{Virtual Page } 2^m - 1 \]

\[ \text{Virtual Page } 2^n - 1 \]

\[ \text{Physical Page } 0 \]

\[ \text{Physical Page } 1 \]

\[ \text{Physical Page } 2^p - 1 \]
Design for a slow disk: exploit locality

Virtual Memory Address Space

Physical Memory Address Space

Page size?

Associativity?

on disk

Replacement policy?

Write policy?
Address translation

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
**Virtual Memory**

*Page table*  
array of *page table entries* (PTEs)  
mapping virtual page to where it is stored

**Physical pages**  
(Physical memory)

<table>
<thead>
<tr>
<th>Physical Page Number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

**Swap space**  
(Disk)

| VP 3 |
| VP 6 |

**Physical page**  
Physical pages

| PP 0 |
| PP 3 |

| VP 1 |
| VP 2 |
| VP 7 |
| VP 4 |

*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)

Virtual page mapped to physical page?

**yes = page hit**

Base address of current process's page table

Page table base register (PTBR)
**Page hit:** virtual page is in memory

![Diagram of virtual memory management](image)

- **Virtual Page Number**
- **Physical Page Number or disk address**
- **Valid**

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>0</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>PP 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PP 1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>PP 3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

- **Physical pages** (Physical memory)
  - VP 1
  - VP 2
  - VP 7
  - VP 4

- **Swap space** (Disk)
  - VP 3
  - VP 6
Page fault:

Virtual Page Number

Physical Page Number or disk address

Valid

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>0</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>PP 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>PP 1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>PP 3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

PTE 7

Swap space (Disk)

| VP 3 |
| VP 6 |

Physical pages (Physical memory)

| VP 1 |
| VP 2 |
| VP 4 |
| PP 0 |
| PP 1 |
| PP 2 |
| PP 3 |
Process *fault*: *exceptional control flow*

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

---

**User Code**

```
movl
```

**OS exception handler**

exception: page fault

return

Load page into memory

---

Returns to faulting instruction:

```
movl
```

is executed *again*!
Page fault: 1. page not in memory

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

Virtual Page Number

Physical Page Number or disk address

Valid

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Physical Page Number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 1</td>
<td>VP 2</td>
</tr>
<tr>
<td>VP 2</td>
<td>VP 4</td>
</tr>
<tr>
<td>VP 3</td>
<td>VP 6</td>
</tr>
<tr>
<td>VP 4</td>
<td>VP 1</td>
</tr>
</tbody>
</table>

**Physical pages** (Physical memory)

- PP 0
  - VP 1
  - VP 2
  - VP 7
- PP 3
  - VP 3
  - VP 4

**Swap space** (Disk)

- VP 3
- VP 6
- VP 1

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

Finally:
Re-execute faulting instruction.
Page hit!
Terminology

context switch
Switch control between processes on the same CPU.

page in
Move page of virtual memory from disk to physical memory.

page out
Move page of virtual memory from physical memory to disk.

thrash
Total working set size of processes is larger than physical memory. Most time is spent paging in and out instead of doing useful work.
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?
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: \texttt{0x03D4}, \texttt{0xB8F}, \texttt{0x0020}
Memory system example: page table

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

<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><strong>VPN</strong></td>
<td><strong>PPN</strong></td>
<td><strong>Valid</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
</tbody>
</table>

| **VPN** | **PPN** | **Valid** |          |             |                 |
| 08 | 13 | 1 |          |             |                 |
| 09 | 17 | 1 |          |             |                 |
| 0A | 09 | 1 |          |             |                 |
| 0B | –  | 0 |          |             |                 |
| 0C | –  | 0 |          |             |                 |
| 0D | 2D | 1 |          |             |                 |
| 0E | 11 | 1 |          |             |                 |
| 0F | 0D | 1 |          |             |                 |

What about a real address space? Read more in the book...
Memory system example: TLB

16 entries
4-way associative

TLB ignores page offset. Why?

<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>

virtual page #  TLB index  TLB tag  TLB Hit?  Page Fault?  physical page #:
Memory system example: 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>
Virtual memory benefits: 
Simple address space allocation

Process needs private *contiguous* address space.

Storage of virtual pages in physical pages is *fully associative.*
Virtual memory benefits:
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.”**
Virtual memory benefits:

**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).*
Virtual memory benefits: Memory permissions

MMU checks on every access. Exception if not allowed.

<table>
<thead>
<tr>
<th>Process 1:</th>
<th>Valid</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Physical Page Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process 2:</th>
<th>Valid</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Physical Page Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

Page Table

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

- Purely an optimization
- "Invisible" to program and OS, no direct control
- Programmer cannot control caching, can write code that fits well

Virtual Memory: Software-Hardware Co-design

- Supports processes, memory management
- Operating System (software) manages the mapping
  - Allocates physical memory
  - Maintains page tables, permissions, metadata
  - Handles exceptions
- Memory Management Unit (hardware) does translation and checks
  - Translates virtual addresses via page tables, enforces permissions
  - TLB caches the mapping
- Programmer cannot control mapping, can control sharing/protection via OS