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?

https://cs.wellesley.edu/~cs240/s20/

Problem 1: memory management

Process 1
Process 2
Process 3
... Process n

(stack heap code globals ...

What goes where?

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...

(To scale with 64-bit address space, you can’t see it.)
Problem 3: protection

Physical main memory

Process i

Process j

Problem 4: sharing

Physical main memory

Process i

Process j

Solution: Virtual Memory (address indirection)

Virtual-to-physical mapping

Private virtual address space per process.

Single physical address space managed by OS/hardware.

Indirection (it's everywhere!)

Direct naming

"2"

"x"

What X currently maps to

Indirect naming

"2"

"x"

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
Virtual addressing and address translation

Memory Management Unit translates virtual address to physical address

Physical addresses are invisible to programs.

Cannot fit all virtual pages! Where are the rest stored?

virtual address space usually much larger than physical address space

1. On disk if used

2. Nowhere if not (yet?) used

Page-based mapping

fixed-size, aligned pages
page size = power of two

Cannot fit all virtual pages! Where are the rest stored?

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?

Not drawn to scale

Example system
Design for a slow disk: exploit locality

Physical Memory Address Space

Virtual Memory Address Space

Virtual Page 0
Virtual Page 1
Virtual Page 2
Virtual Page 3
...
Virtual Page 2\(^{p-1}\)

Physical Page 0
Physical Page 1
Physical Page 2
...
Physical Page 2\(^{m-1}\)

on disk

2\(^{n-1}\)

2\(^{m-1}\)

Virtual Memory

Address translation

CPU Chip

Virtual address (VA) 4100

Physical address (PA)

Main memory

PTE 0
PTE 7

Valid
0
1
0
1
0
null

Physical Page Number or disk address

PTE 0
PTE 7

Physical pages (Physical memory)

VP 1
VP 2
VP 3
VP 4
VP 7
VP 6

Swap space (Disk)

Memory resident, managed by HW (MMU), OS

Page table

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

 associative

Page size?

Associativity?

Replacement policy?

Write policy?

Design for a slow disk: exploit locality

Fully associative

• Store any virtual page in any physical page

• Large mapping function

Large page size

usually 4KB, up to 2\(^{4}\)MB

Sophisticated replacement policy

• Not just hardware

Write-back

Replacement policy?

Write policy?
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)

Page hit: virtual page is in memory

- Virtual Page Number
- Physical Page Number or disk address

- PTE 0
  - Valid
  - PP 0
  - null

- PTE 7
  - Valid
  - PP 2
  - null

Page fault:

- Virtual Page Number
- Physical Page Number or disk address

- PTE 0
  - Valid
  - PP 0
  - null

- PTE 7
  - Valid
  - PP 2
  - null

Page fault: exceptional control flow

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

User Code

- exception: page fault

OS exception handler

- Load page into memory

movl

- return

Returns to faulting instruction: movl is executed again!

Swap space (Disk)

- VP 3
- VP 6

Physical pages (Physical memory)

- VP 1
- PP 0
- PP 1
- PP 2
- PP 3
- VP 4
- VP 7
- VP 3
- VP 6

Virtual Memory
Page fault: 1. page not in memory

Virtual Page Number

Physical Page Number or disk address

Valid

PTE 0

0 null
1 PP 0
1 PP 1
1 PP 3
0 null
0 On disk
1 PP 2

Physical pages (Physical memory)

VP 0

VP 1

VP 2

VP 7

VP 4

PP 0

PP 3

Swap space (Disk)

VP 3

VP 6

OS handles fault

What now?

Page fault: 2. OS evicts another page.

Virtual Page Number

Physical Page Number or disk address

Valid

PTE 0

0 null
1 PP 0
1 PP 1
1 PP 3
0 null
0 On disk
1 PP 2

Physical pages (Physical memory)

VP 0

VP 1

VP 2

VP 7

VP 4

PP 0

PP 3

Swap space (Disk)

VP 3

VP 6

"Page out"

Page fault: 3. OS loads needed page.

Virtual Page Number

Physical Page Number or disk address

Valid

PTE 0

0 null
1 On disk
1 PP 1
1 PP 0
1 PP 3
0 null
0 On disk
1 PP 2

Physical pages (Physical memory)

VP 0

VP 1

VP 2

VP 7

VP 4

PP 0

PP 3

Swap space (Disk)

VP 3

VP 6

"Page in"

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 computation.
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

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 miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Does a TLB miss require disk access?

Memory system example: page table
Only showing first 16 entries (out of 256 = 2^8)

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>

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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>09</td>
<td>0D</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0A</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>
Memory system example: **cache**

- 16 lines
- 4-byte block size
- Physically addressed
- Direct mapped

### Cache Addressing

<table>
<thead>
<tr>
<th>idx</th>
<th>Tag</th>
<th>Valid</th>
<th>BO</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>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>18</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>72</td>
<td>F0</td>
<td>1D</td>
<td>-</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>

**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 Diagram](image-url)
Virtual memory benefits:

**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**
- 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