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?

Problem 1: memory management

Process 1
Process 2
Process 3

... Process n

stack
heap
code
globals

What goes where?

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)

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

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

Context switches must swap out entire memory contents.
Isn’t that expensive?

Problem 2: capacity
**Problem 3: protection**

Virtual Memory

**Problem 4: sharing**

Virtual Memory

---

**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

CPU Chip → MMU → Main memory

Virtual address (VA) → Physical address (PA)

Physical addresses are *invisible* to programs.

Virtual Memory

---

**Page-based mapping**

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

Virtual Address Space

Physical Address Space

Map virtual pages onto physical pages.

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

---

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

Virtual Memory Address Space

Physical Memory Address Space

1. On disk if used

2. Nowhere if not (yet?) used

---

**Virtual memory: cache for disk?**

Example system
Design for a slow disk: exploit locality

Virtual Memory Address Space

0  1  2  ...  2^n - 1

Physical Memory Address Space

0  1  2  ...  2^m - 1

on disk

Virtual Memory

Physical Memory

Replacement policy?

Page size?

Associativity?

Write policy?

Address translation

CPU Chip

Virtual address (VA) → Physical address (PA)

Main memory

Valid

Physical Page Number or disk address

Memory resident, managed by HW (MMU), OS

Swap space (Disk)

Page table

array of page table entries (PTEs)

mapping virtual page to where it is stored

Physical pages (Physical memory)

VP 1  VP 2  VP 3  VP 4  VP 5  VP 6  VP 7

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 page offset (PPO)

Physical address (PA)

Virtual Memory

Page hit: virtual page is in memory

Virtual Page Number

Physical Page Number or disk address
- Valid
- PTE 0
- PTE 7

Swap space (Disk)

Physical pages (Physical memory)

Virtual Memory

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

Virtual Memory

Returns to faulting instruction: movl is executed again!
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

0 On disk

1 PP 3

0 null

1 On disk

PTE 7

1 PP 2

Swap space

(Disk)

Virtual Page Number

Physical Page Number or disk address

Valid

PTE 0

0 On disk

1 PP 1

0 On disk

1 PP 3

0 null

1 PP 2

PTE 7

1 PP 0

PP 0

VP 0

PP 1

VP 1

PP 2

VP 2

PP 3

VP 3

PP 4

VP 4

PP 5

VP 5

PP 6

VP 6

PP 7

VP 7

What now?

OS handles fault

Page fault: 2. OS evicts another page.

Virtual Page Number

Physical Page Number or disk address

Valid

PTE 0

0 null

1 PP 1

0 On disk

1 PP 3

0 null

1 On disk

PTE 7

1 PP 0

PP 0

VP 0

PP 1

VP 1

PP 2

VP 2

PP 3

VP 3

PP 4

VP 4

PP 5

VP 5

PP 6

VP 6

PP 7

VP 7

"Page out"

"Page in"

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

0 On disk

1 PP 3

0 null

1 PP 2

PTE 7

1 PP 0

PP 0

VP 0

PP 1

VP 1

PP 2

VP 2

PP 3

VP 3

PP 4

VP 4

PP 5

VP 5

PP 6

VP 6

PP 7

VP 7

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

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 "unclassiest 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 $2^8$)

<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>

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?
Memory system example: cache

16 lines
4-byte block size
Physically addressed
Direct mapped

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

Process 1:
- PP 6: Valid - Yes, WRITE - No, EXEC - Yes
- PP 4: Valid - Yes, WRITE - No, EXEC - Yes
- PP 2: Valid - Yes, WRITE - No, EXEC - Yes

Process 2:
- PP 9: Valid - Yes, WRITE - Yes, EXEC - No
- PP 6: Valid - Yes, WRITE - No, EXEC - Yes
- PP 11: Valid - Yes, WRITE - Yes, EXEC - No

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