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

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)

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

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

Virtual address space

Virtual addresses

virtual addresses

virtual addresses

Physical memory

data

data

Private virtual address space per process.

Single physical address space managed by OS/hardware.

**Indirect naming**

"2"

"2"

"2"

"2"

"X"

"X"

"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

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

Virtual address
(VA)

Physical address
(PA)

Main memory

0: 1:

Main Memory

MMU

Data

Virtual address space
usually much larger than
physical address space

2^n - 1

Virtual Memory Address Space
0

Virtual Page 0

Virtual Page 1

Virtual Page 2

Virtual Page 3

Virtual Page 4

Virtual Page 5

Virtual Page 6

Virtual Page 7

Virtual Page 8

Virtual Page 9

Virtual Page 10

Virtual Page 11

Virtual Page 12

Virtual Page 13

Virtual Page 14

Virtual Page 15

Physical Memory Address Space
0

Physical Page 0

Physical Page 1

Physical Page 2

Physical Page 3

Physical Page 4

Physical Page 5

Physical Page 6

Physical Page 7

Physical Page 8

Physical Page 9

Physical Page 10

Physical Page 11

Physical Page 12

Physical Page 13

Physical Page 14

Physical Page 15

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

Virtual memory: cache for disk?

Not drawn to scale

SRAM

~4 MB

32 KB

L1 i-cache

L2 unified cache

CPU

Reg

Main Memory

2 B/cycle
100 cycles

1 B/30 cycles
millions

16 B/cycle
3 cycles

8 B/cycle
14 cycles

Throughput:

Latency:

Cache miss penalty
(latency): 33x

~8 GB

~500 GB

solid-state "flash"
or spinning magnetic platter.

Cache miss penalty
(latency): 10,000x

Example system

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

Virtual addresses are invisible to programs.
Design for a slow disk: exploit locality

Virtual Memory Address Space

Physical Memory Address Space

on disk

$2^n - 1$

$2^m - 1$

Physical Memory

Virtual Memory

Address translation

CPU Chip

Virtual address (VA)

Physical address (PA)

CPU

4100

MMU

Main memory

PTE 0

PTE 7

Valid

null

null

Physical Page Number or disk address

Physical pages (Physical memory)

Swap space (Disk)

Page table

array of page table entries (PTEs)

mapping virtual page to where it is stored

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 base register (PTBR):
- Base address of current process's page table

Virtual page number (VPN) = Physical page number (PPN) + Physical page offset (PPO)

Valid Physical page number (PPN)

Virtual page mapped to physical page?
yes = page hit

Page hit: virtual page is in memory

Physical pages (Physical memory)

Virtual Page Number

Physical Page Number or disk address

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

PTE 7
- 0 null
- 1 PP 2

Swap space (Disk)

VP 0, VP 1, VP 2, VP 3, VP 4, VP 5, VP 6

Swap space

Virtual Memory

Page fault: exceptional control flow

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

Process

User Code

OS exception handler

exception: page fault

movl

Load page into memory

return

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.

**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

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?

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>0</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>0</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>0</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: **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**

<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>

---

**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