Slide 1

Virtual Memory

Process Abstraction, Part 2: Private Address Space

Motivation: why not direct physical memory access?
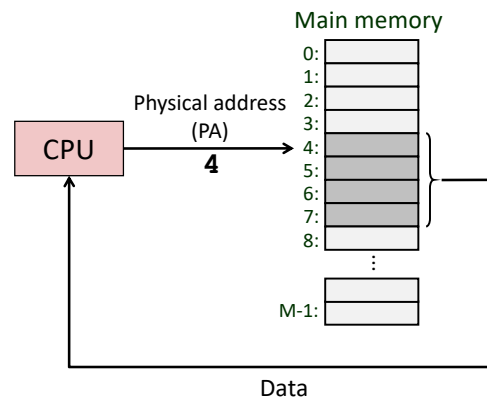**Address translation** with pages
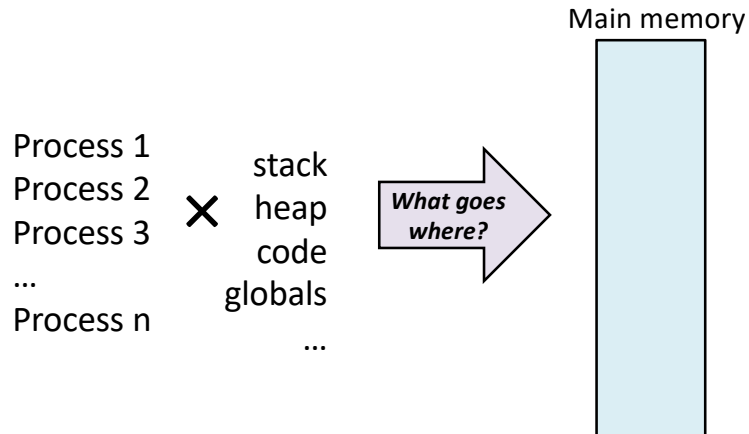**Optimizing translation**: translation lookaside buffer
Extra benefits: sharing and protection

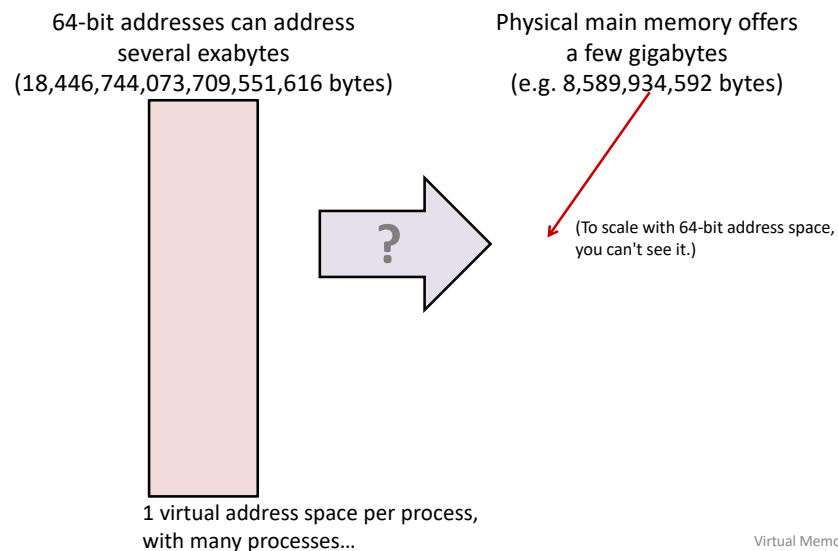Memory as a contiguous array of bytes is a lie! Why?

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

Virtual Memory   1

Slide 2

Problems with physical addressing

Data

Virtual Memory   2

Slide 3

Problem 1: memory management

Main memory

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**?

Virtual Memory   3

Slide 4

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…

Virtual Memory   4

Problem 3: protection

Physical main memory

Process i

Process j

✗

Problem 4: sharing

Physical main memory
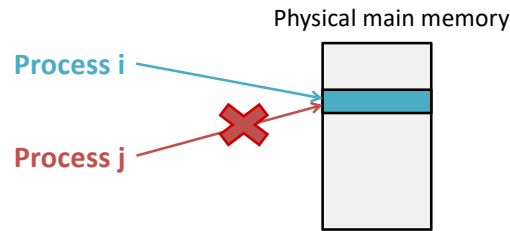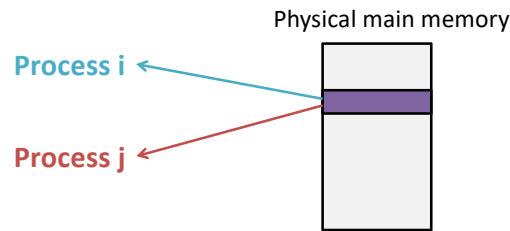
Process i

Process j

Solution: Virtual Memory (address *indirection*)

data

Process 1

Virtual address space

virtual addresses

Physical memory

virtual-to-physical **mapping**

physical addresses

Process n

Virtual address space

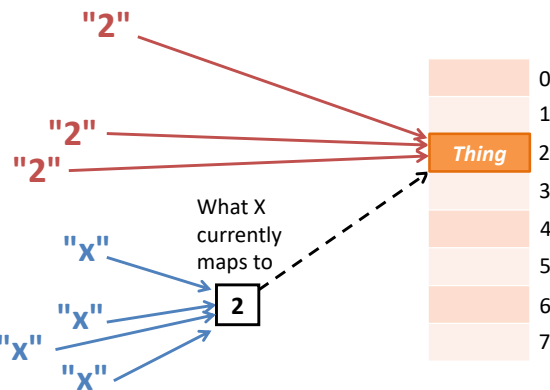virtual addresses

data
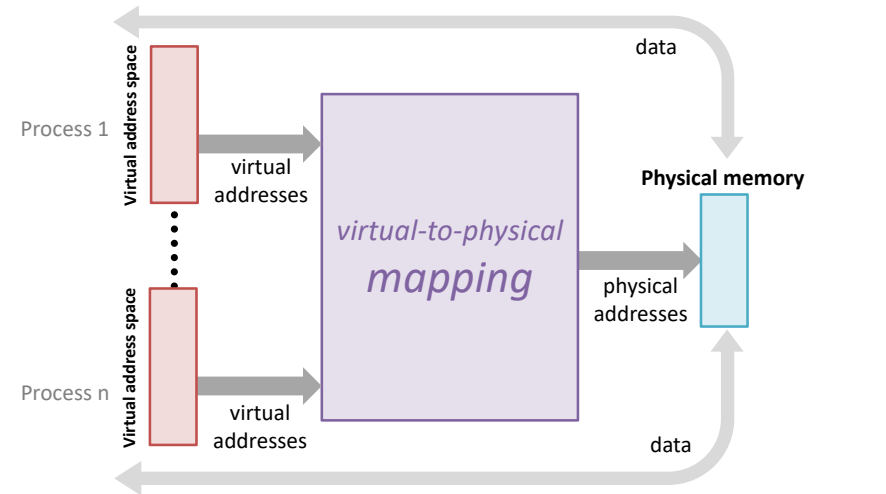
Private virtual address space per process.

Single physical address space managed by OS/hardware.

Indirection
(it's everywhere!)

Direct naming

"2"

"2"

"2"

0
1
2 *Thing*
3
4
5
6
7

What X currently maps to

Indirect naming

"x"

"x"

"x"

"x"

2

What if we move *Thing*?

Tangent: indirection everywhere

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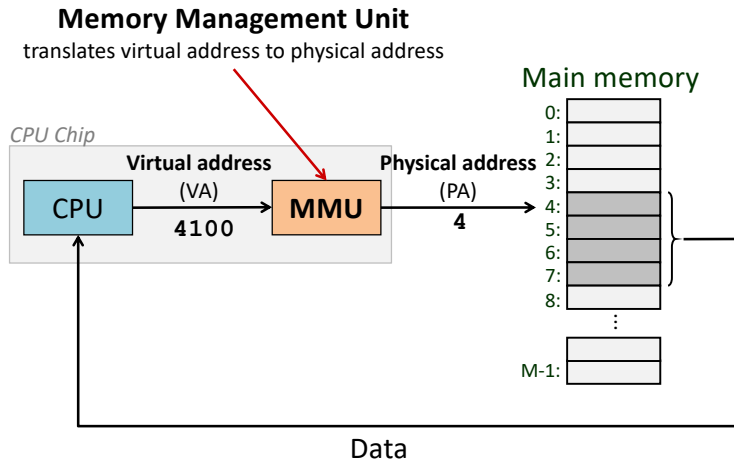- Procedural abstraction
- Domain Name Service (DNS)
- Dynamic Host Configuration Protocol (DHCP)
- Phone numbers
- 911
- Call centers
- Snail mail forwarding

"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."
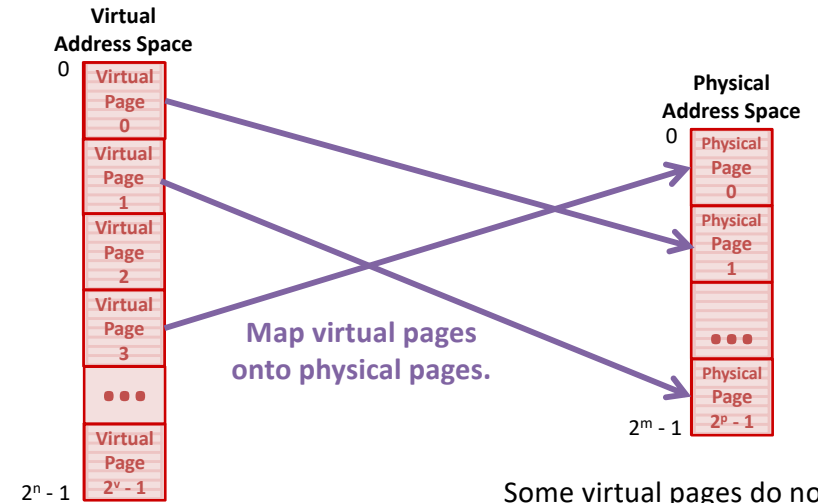
Virtual addressing and address translation

Memory Management Unit
translates virtual address to physical address

CPU Chip

CPU		MMU	

Virtual address (VA) **4100**

Physical address (PA) **4**

Main memory
- 0:
- 1:
- 2:
- 3:
- 4:
- 5:
- 6:
- 7:
- 8:
- ⋮
- M-1:

Data

Physical addresses are invisible to programs.

Page-based mapping

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

Virtual Address Space

0
- Virtual Page 0
- Virtual Page 1
- Virtual Page 2
- Virtual Page 3
- • • •
- Virtual Page $2^v - 1$

$2^n - 1$

Physical Address Space

0
- Physical Page 0
- Physical Page 1
- • • •
- Physical Page $2^p - 1$

$2^m - 1$

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

0
- Virtual Page 0
- Virtual Page 1
- Virtual Page 2
- Virtual Page 3
- • • •
- Virtual Page

$2^n - 1$

virtual address space
usually much larger than
physical address space

Physical Memory Address Space

0
- Physical Page 0
- Physical Page 1
- • • •
- Physical Page $2^p - 1$

$2^m - 1$

1. On disk if used

2. Nowhere if not *(yet?)* used

Not drawn to scale

Virtual memory: cache for disk?

SRAM

DRAM

L1 I-cache

~4 MB
L2 unified cache

~8 GB
Main Memory

~500 GB
Disk

CPU | Reg

32 KB
L1 D-cache

	16 B/cycle	8 B/cycle	2 B/cycle	1 B/30 cycles
Throughput:	3 cycles	14 cycles	100 cycles	millions
Latency:				

Cache miss penalty (latency): 33x

Memory miss penalty (latency): 10,000x

solid-state "flash"
or
spinning
magnetic platter.

Example system

Design for a slow disk: exploit locality

Virtual Memory Address Space

0
- Virtual Page 0
- Virtual Page 1
- Virtual Page 2
- Virtual Page 3
- ● ● ●
- Virtual ✖

$2^n - 1$

on disk

Physical Memory Address Space

0
- Physical Page 0
- Physical Page 1
- ● ● ●
- Physical Page $2^p - 1$

$2^m - 1$

Design for a slow disk: exploit locality

Virtual Memory Address Space

0
- Virtual Page 0
- Virtual Page 1
- Virtual Page 2
- Virtual Page 3
- ● ● ●
- Virtual ✖

$2^n - 1$

on disk

Page size?

Associativity?

Replacement policy?

Write policy?

Physical Memory Address Space

0
- Physical Page 0
- Physical Page 1
- ● ● ●
- Physical Page $2^p - 1$

$2^m - 1$

Address translation

CPU Chip

CPU → 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)

Physical Page Number or disk address

Valid

	Valid	
PTE 0	0	null
	1	●
	1	●
	0	●
	1	●
	0	null
	0	●
PTE 7	1	●

page table

Memory resident, managed by HW (MMU), OS

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

Swap space (Disk)
- VP 3
- VP 6
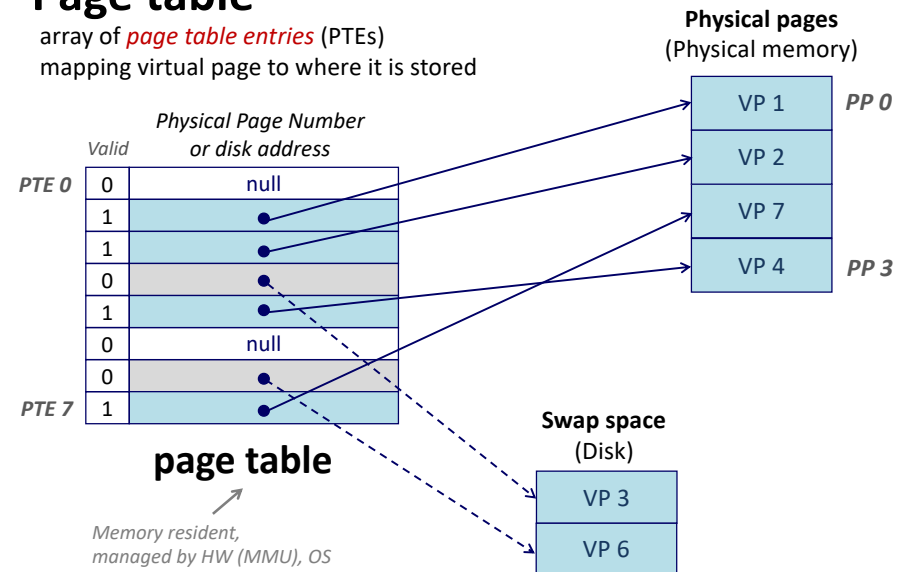
How many page tables are in the system?

Address translation with a page table

Page table base register (PTBR)

Base address of current process's page table

Virtual address (VA)

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

Page table

Valid   Physical page number (PPN)

Virtual page mapped to physical page?

yes = **page hit**

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

Physical address (PA)
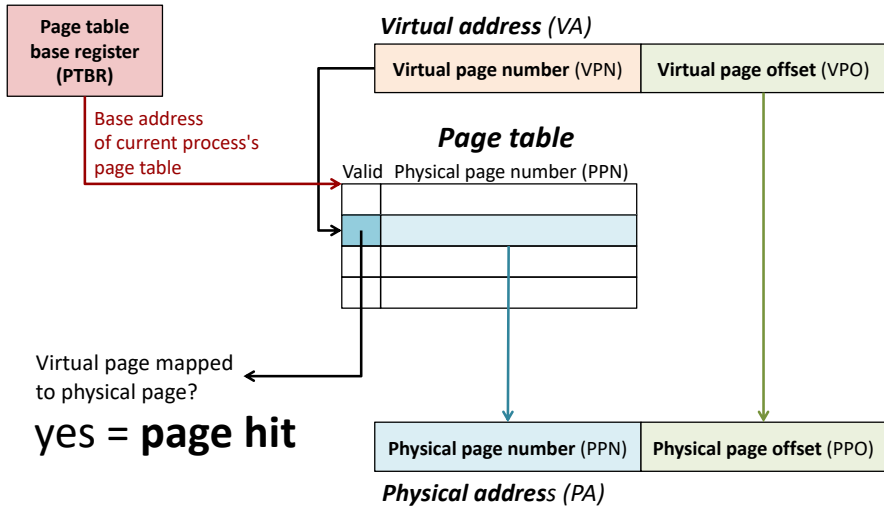
Page *hit:* virtual page is in memory

Virtual Page Number

Physical pages (Physical memory)

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

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
	0	On disk
PTE 7	1	PP 2

page table

Swap space (Disk)

VP 3
VP 6

Page *fault:*

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
	0	On disk
PTE 7	1	PP 2

page table

Physical pages (Physical memory)

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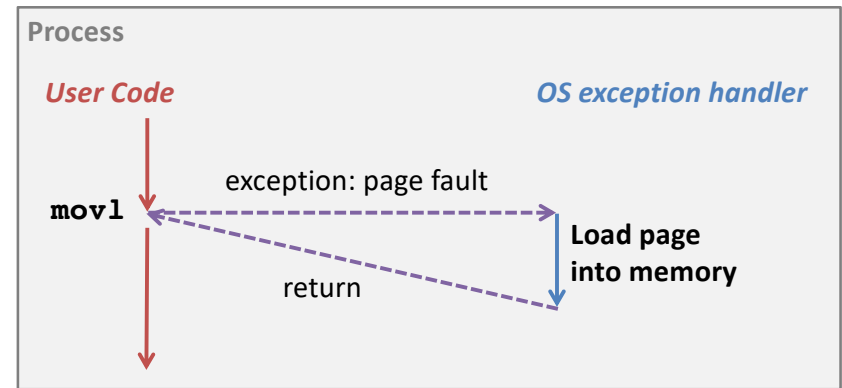VP 2   *PP 1*
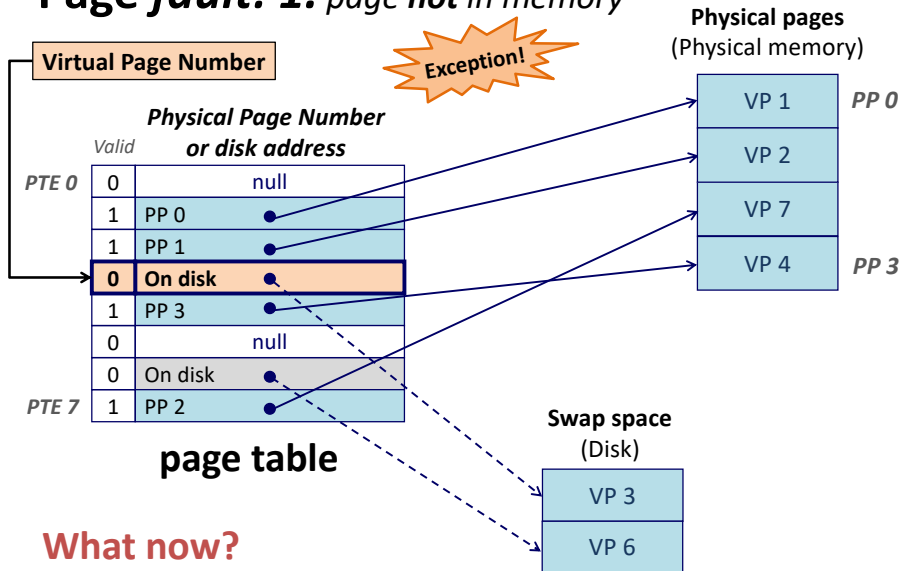VP 7   *PP 2*
VP 4   *PP 3*

Swap space (Disk)

VP 3
VP 6

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

Load page into memory

return

Returns to faulting instruction: `movl` is executed *again*!

Page *fault:* 1. *page **not** in memory*

Exception!

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
	0	On disk
PTE 7	1	PP 2

page table

Physical pages
(Physical memory)

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

Swap space
(Disk)

VP 3
VP 6

What now?
OS handles fault

Page *fault:* 2. *OS evicts another page.*

"Page out"

Virtual Page Number

Physical Page Number or disk address

	Valid	
PTE 0	0	null
	0	On disk
	1	PP 1
	0	On disk
	1	PP 3
	0	null
	0	On disk
PTE 7	1	PP 2

page table

Physical pages
(Physical memory)

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

Swap space
(Disk)

VP 3
VP 6
VP 1
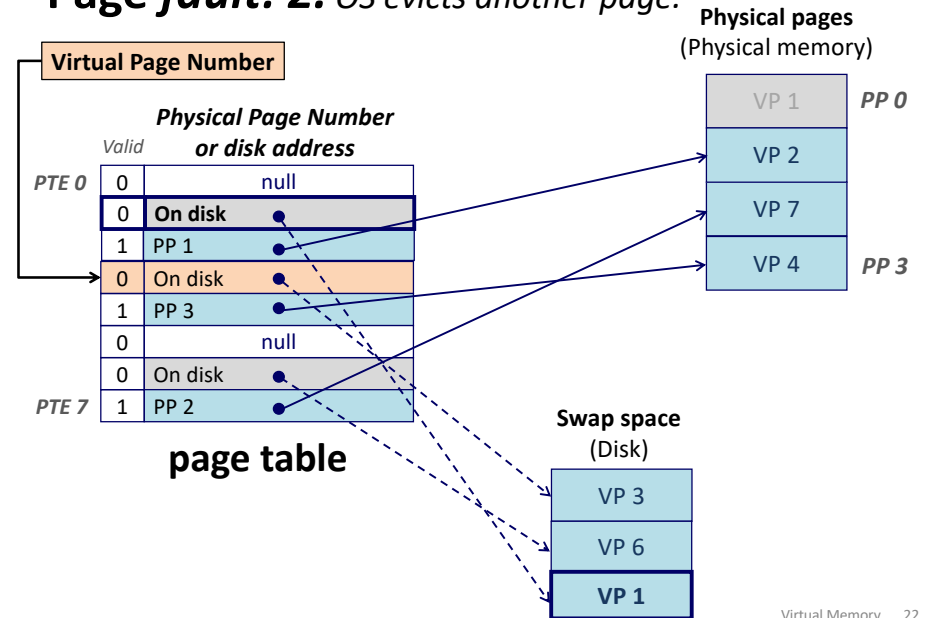
Page *fault:* 3. *OS loads needed page.*

"Page in"

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
PTE 7	1	PP 2

page table

Physical pages
(Physical memory)

VP 3	PP 0
VP 2	
VP 7	
VP 4	PP 3

Swap space
(Disk)

VP 3
VP 6
VP 1
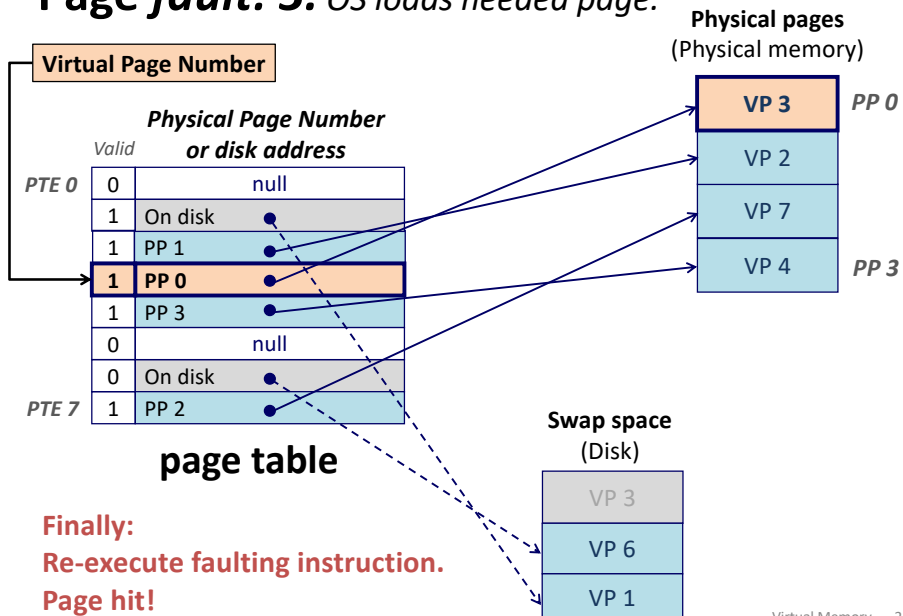
Finally:
Re-execute faulting instruction.
Page hit!

Terminology

context switch
  Switch control between processes on the same CPU.

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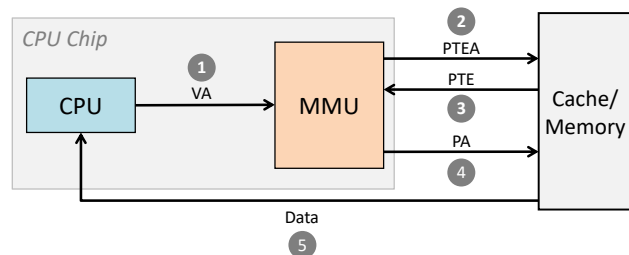 Move page of virtual memory from disk to physical memory.

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*
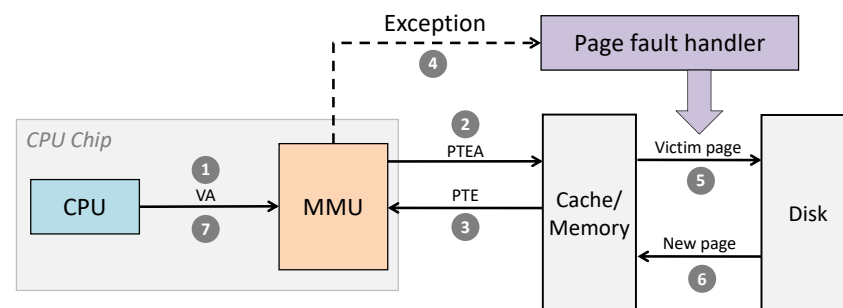
1) Processor sends virtual address to MMU

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?

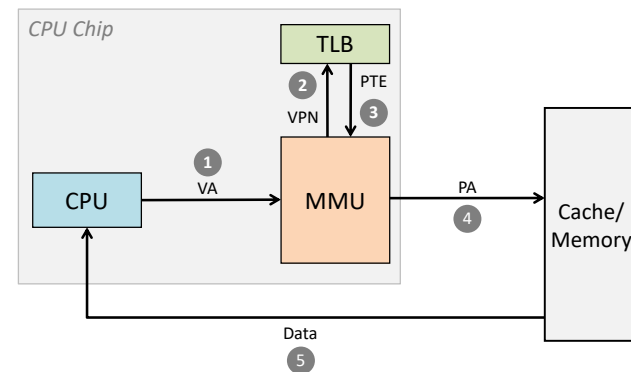How many physical memory accesses are required to complete one virtual memory access?

Translation Lookaside Buffer (TLB)

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

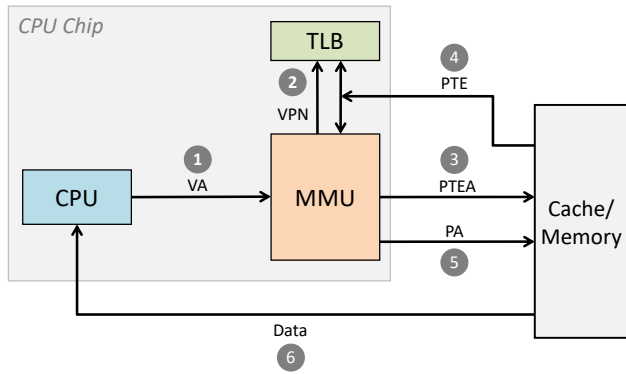Much faster than a page table lookup in memory.

In the running for *"un/classiest name of a thing in CS"*

TLB hit

A TLB hit eliminates a memory access

TLB miss

CPU Chip

TLB
④
② PTE
VPN
③ PTEA
① VA
CPU → MMU
Cache/ Memory
PA
⑤
Data
⑥

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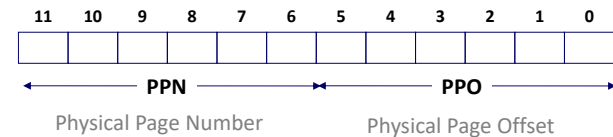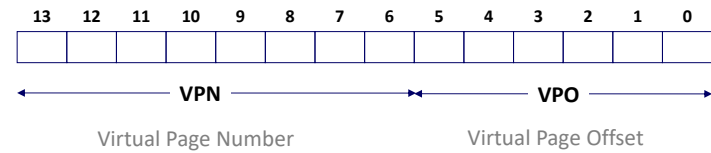

Memory system example (small)

Addressing

Simulate accessing these virtual addresses on the system: `0x03D4, 0x0B8F, 0x0020`

14-bit virtual addresses

12-bit physical address

Page size = 64 bytes

13	12	11	10	9	8	7	6	5	4	3	2	1	0

◄──────────── VPN ────────────►◄──────────── VPO ────────────►

Virtual Page Number   Virtual Page Offset

11	10	9	8	7	6	5	4	3	2	1	0

◄─────────── PPN ───────────►◄─────────── PPO ───────────►

Physical Page Number   Physical Page Offset

Memory system example: **page table**

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

virtual page #____   TLB index____   TLB tag _____   TLB Hit? __ Page Fault? __ physical page #: _____

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

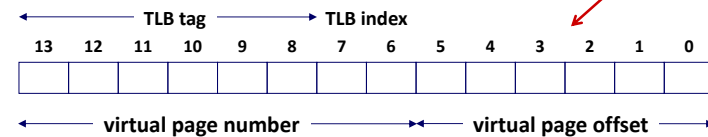| VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

What about a real address space? Read more in the book…

Memory system example: **TLB**

16 entries

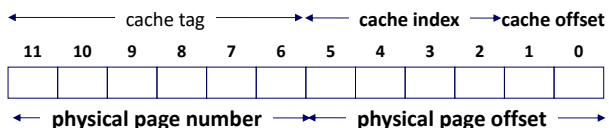4-way associative

TLB ignores page offset. Why?

◄──────── TLB tag ────────► TLB index

13	12	11	10	9	8	7	6	5	4	3	2	1	0

◄──────── virtual page number ────────►◄──────── virtual page offset ────────►

virtual page #____   TLB index____   TLB tag _____   TLB Hit? __ Page Fault? __ physical page #: _____

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Memory system example: **cache**

16 lines
4-byte block size
Physically addressed
Direct mapped

	cache tag				cache index	→	cache offset		
11	10	9	8	7	6	5	4	3	2	1	0

← physical page number → ← physical page offset →

cache offset____   cache index____   cache tag_____   Hit? ___   Byte: ____

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

| Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Virtual memory benefits:
Simple address space allocation

Process needs private *contiguous* address space.

Storage of virtual pages in physical pages is **fully associative.**
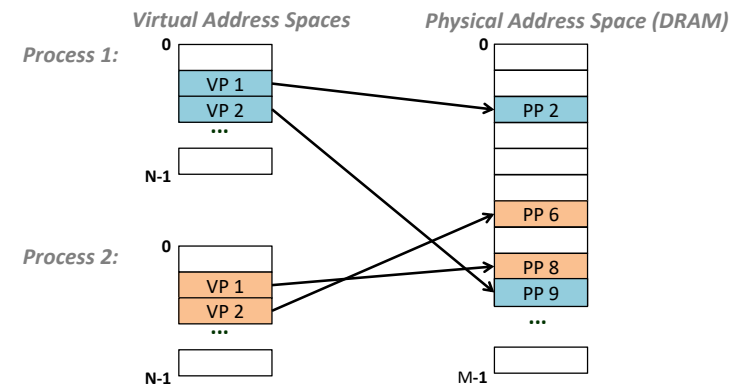
Virtual memory benefits:
Simple cached access to storage > memory

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

All necessary
page table entries
fit in TLB

Working set pages
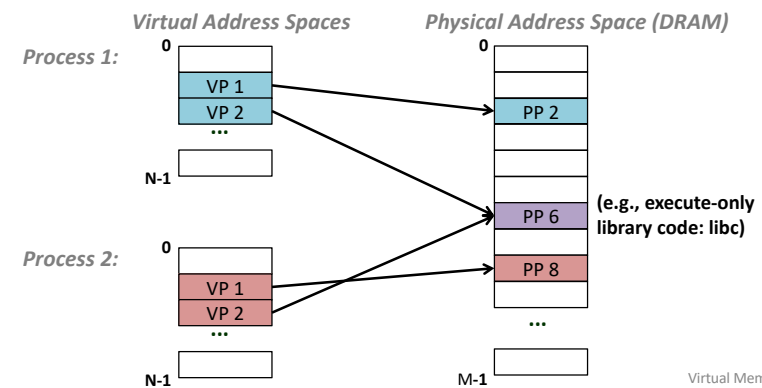fit in physical memory
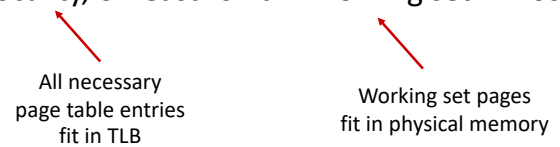
If combined working set > physical memory:

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.*"
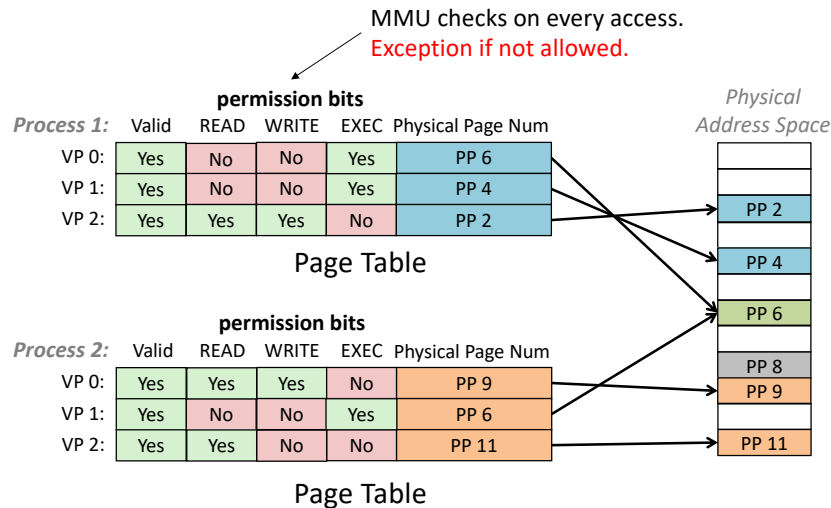

Virtual memory benefits:

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).*

(e.g., execute-only
library code: libc)

Virtual memory benefits: **Memory permissions**

MMU checks on every access.
Exception if not allowed.

Process 1:

permission bits

	Valid	READ	WRITE	EXEC	Physical Page Num
VP 0:	Yes	No	No	Yes	PP 6
VP 1:	Yes	No	No	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

Page Table

Process 2:

permission bits

	Valid	READ	WRITE	EXEC	Physical Page Num
VP 0:	Yes	Yes	Yes	No	PP 9
VP 1:	Yes	No	No	Yes	PP 6
VP 2:	Yes	Yes	No	No	PP 11

Page Table

Physical Address Space

PP 2
PP 4
PP 6
PP 8
PP 9
PP 11

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