Virtual Memory I

CSE 351 Spring 2019

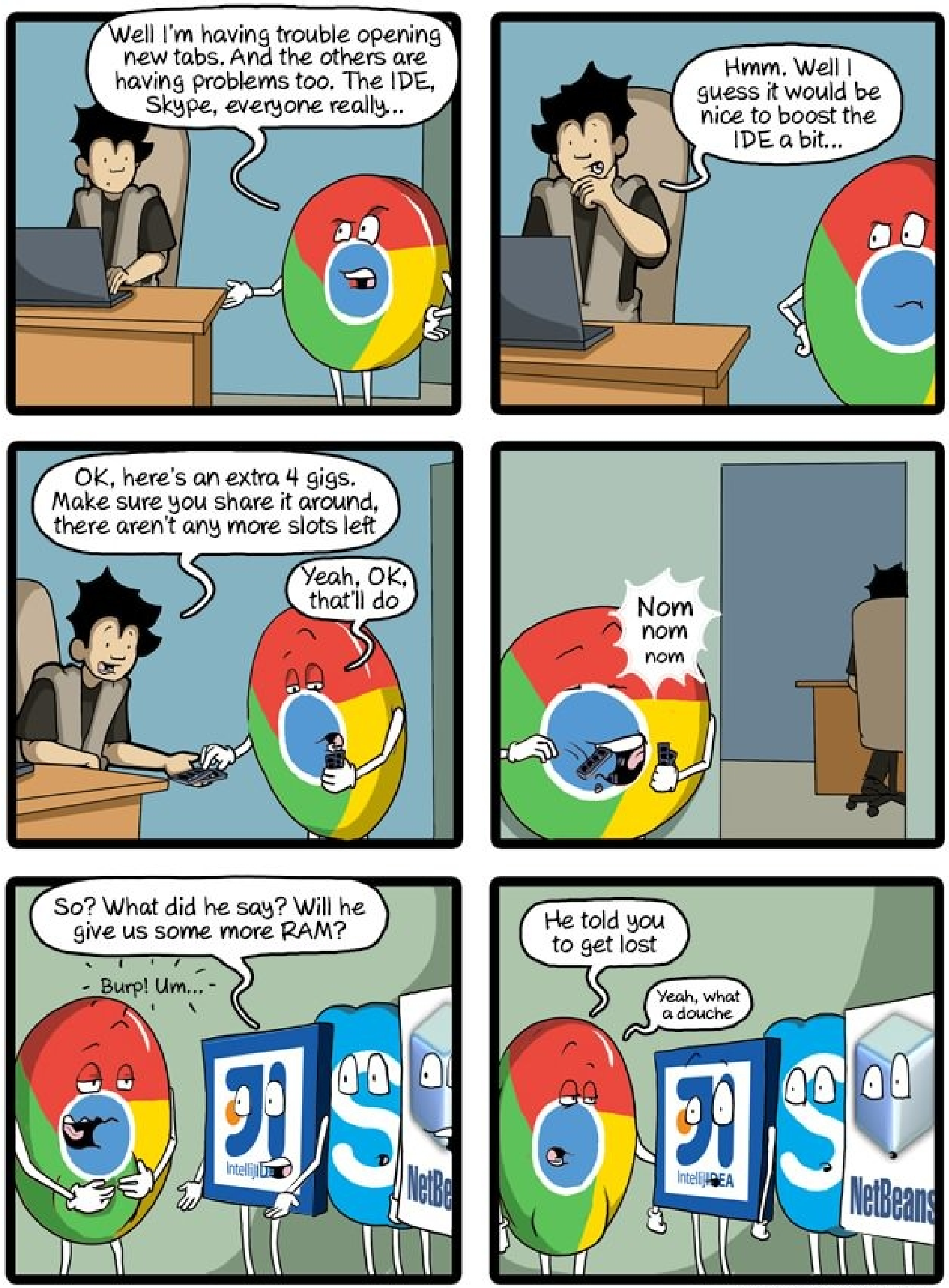
Instructor:

Ruth Anderson

Teaching Assistants:

Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh

http://rebrn.com/re/bad-chrome-1162082/



CommitStrip.com

Administrivia

- Homework 4 , due Wed (5/22) (Structs, Caches)
- Lab 4, due Fri (5/24)

Processes

- Processes and context switching
- Creating new processes
  - `fork()`, `exec*()`, and `wait()`
- **Zombies**

Zombies

- A terminated process still consumes system resources
  - Various tables maintained by OS
  - Called a "zombie" (a living corpse, half alive and half dead)
- *Reaping* is performed by parent on terminated child
  - Parent is given exit status information and kernel then deletes zombie child process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
    - **Note:** on recent Linux systems, `init` has been renamed `systemd`
  - In long-running processes (e.g. shells, servers) we need *explicit* reaping

`wait`: Synchronizing with Children

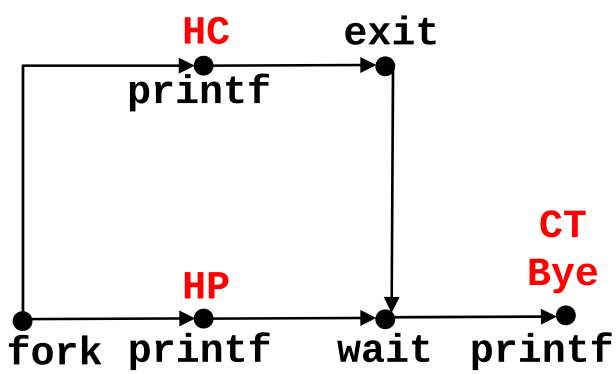
- **`int`** `wait(`**`int`** `*child_status)`
  - Suspends current process (*i.e.* the parent) until one of its children terminates in case we need some of the results from the children to continue
  - Return value is the PID of the child process that terminated
    - *On successful return, the child process is reaped*
  - If `child_status != NULL`, then the `*child_status` value indicates why the child process terminated
    - Special macros for interpreting this status – see **`man wait(2)`**
- **Note:** If parent process has multiple children, `wait` will return when *any* of the children terminates
  - `waitpid` can be used to wait on a specific child process

wait: Synchronizing with Children

```
void fork_wait() {
   int child_status;

   if (fork() == 0) { // child
      printf("HC: hello from child\n");
      exit(0);
   } else { // parent
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
}
```

forks.c



Feasible output:		Infeasible output:
HC	HP	HP
HP	HC	CT
CT	CT	Bye
Bye	Bye	HC

Example: Zombie

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1); /* Infinite loop */
    }
}
```

parent persists

forks.c

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

parent

child

- `ps` shows child process as "defunct"
- Killing parent allows child to be reaped by `init`

Example: Non-terminating Child

```
void fork8() {
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1); /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

child persists

forks.c

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

Process Management Summary

- `fork` makes two copies of the same process (parent & child)
  - Returns different values to the two processes
- `exec*` replaces current process from file (new program)
  - Two-process program:
    - First `fork()`
    - **if** (pid == 0) { */* child code */* } **else** { */* parent code */* }
  - Two different programs:
    - First `fork()`
    - **if** (pid == 0) { execv(…) } **else** { */* parent code */* }
- `wait` or `waitpid` used to synchronize parent/child execution and to reap child process

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 10

OS X Yosemite

Computer system:

Virtual Memory (VM*)

- **Overview and motivation**
- **VM as a tool for caching**
- Address translation
- VM as a tool for memory management
- VM as a tool for memory protection

Warning: Virtual memory is pretty complex, but crucial for understanding how processes work and for debugging performance

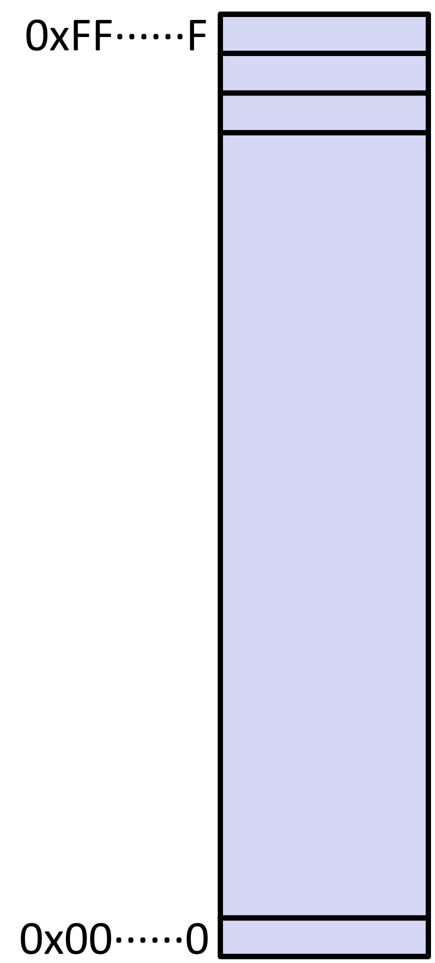
**Not to be confused with "Virtual Machine" which is a whole other thing.*

Memory as we know it so far… is *virtual!*

- Programs refer to virtual memory addresses
  - `movq (%rdi),%rax`
  - Conceptually memory is just a very large array of bytes
  - System provides private address space to each process
- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation within single virtual address space
- But…

  2^{64} bytes

  - We *probably* don't have 2^w bytes of physical memory
  - We *certainly* don't have 2^w bytes of physical memory ***<u>for every process</u>***
  - Processes should not interfere with one another
    - Except in certain cases where they want to share code or data



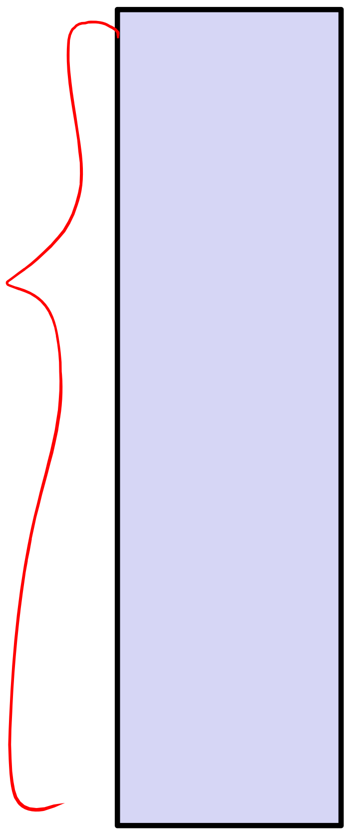
Problem 1: How Does Everything Fit?

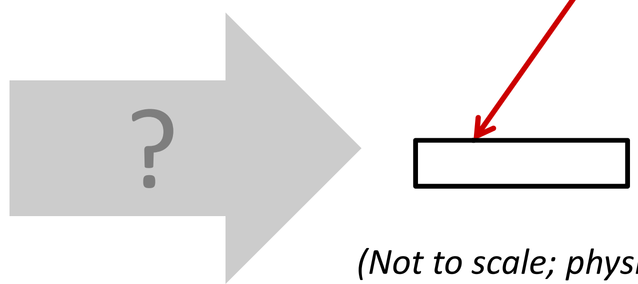
64-bit virtual addresses can address several exabytes (18,446,744,073,709,551,616 bytes)

16 EiB

Physical main memory offers a few gigabytes (*e.g.* 8,589,934,592 bytes)

8 GiB





(Not to scale; physical memory would be smaller than the period at the end of this sentence compared to the virtual address space.)

smaller than this!

1 virtual address space per process, with many processes…

Problem 2: Memory Management

We have multiple processes:

Process 1
Process 2
Process 3
…
Process n

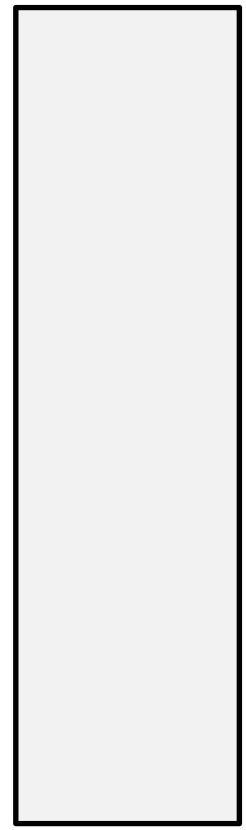
X

Each process has…

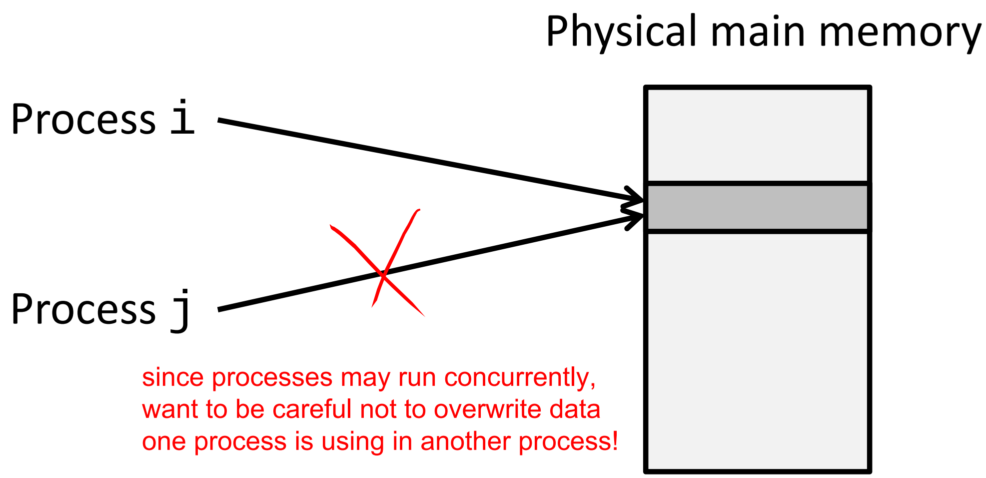
stack
heap
`.text`
`.data`
…

What goes where?

Physical main memory

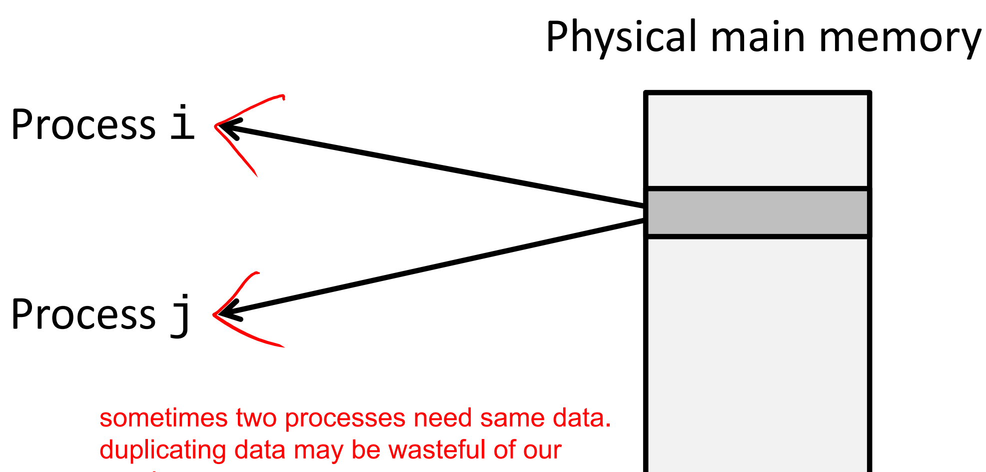


Problem 3: How To Protect



since processes may run concurrently, want to be careful not to overwrite data one process is using in another process!

Problem 4: How To Share?



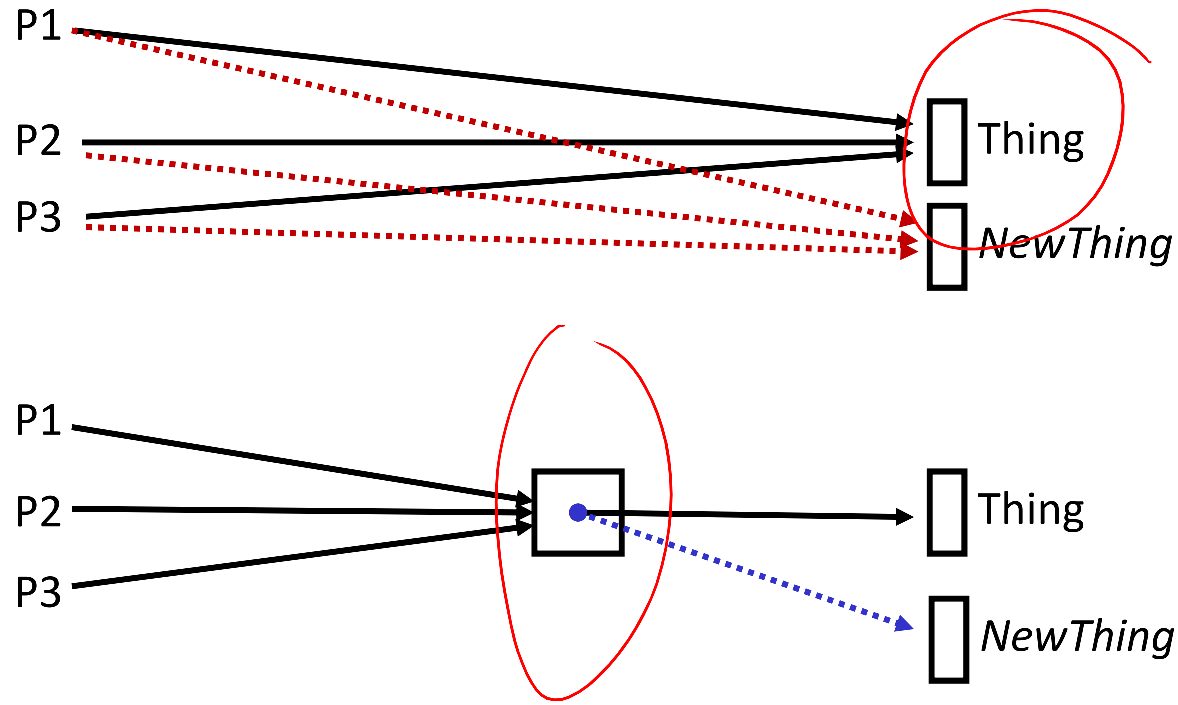
sometimes two processes need same data. duplicating data may be wasteful of our precious memory

How can we solve these problems?

- "Any problem in computer science can be solved by adding another level of **indirection**." *– David Wheeler, inventor of the subroutine*

- Without Indirection

- With Indirection

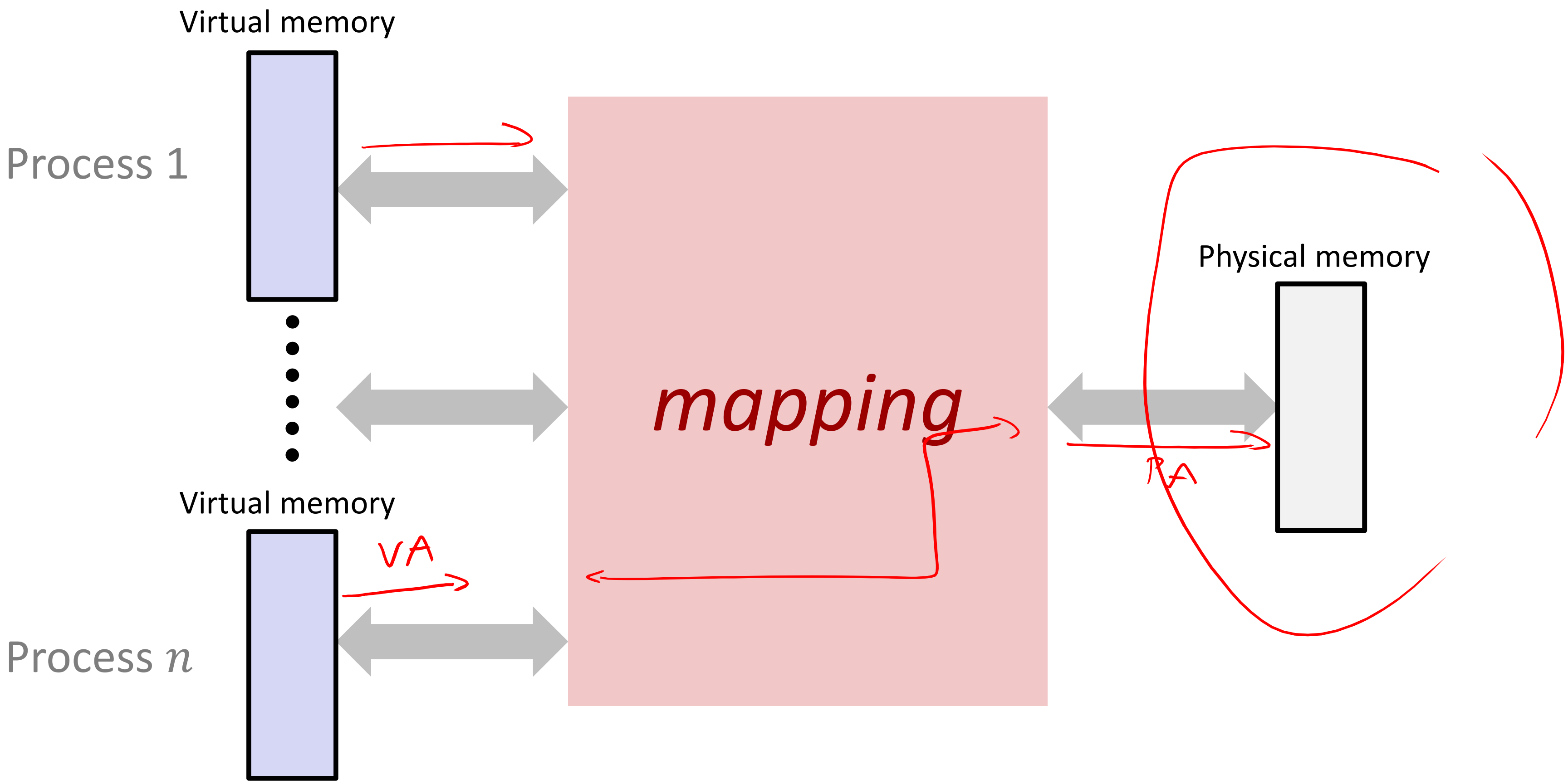


What if I want to move Thing?

Indirection

- *Indirection*: The ability to reference something using a name, reference, or container instead of the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying holders of the name.
  - Adds some work (now have to look up 2 things instead of 1)
  - But don't have to track all uses of name/address (single source!)
- Examples:
  - **Phone system:** cell phone number portability
  - **Domain Name Service (DNS):** translation from name to IP address
  - **Call centers:** route calls to available operators, etc.
  - **Dynamic Host Configuration Protocol (DHCP):** local network address assignment

Indirection in Virtual Memory



- Each process gets its own private virtual address space
- Solves the previous problems!

Address Spaces

bits $n = \lceil \log_2 N \rceil$ ceiling function (round up)

- Virtual address space: Set of $\mathrm{N} = 2^n$ virtual addr
  - {0, 1, 2, 3, …, N-1}

bytes

$m = \lceil \log_2 M \rceil$

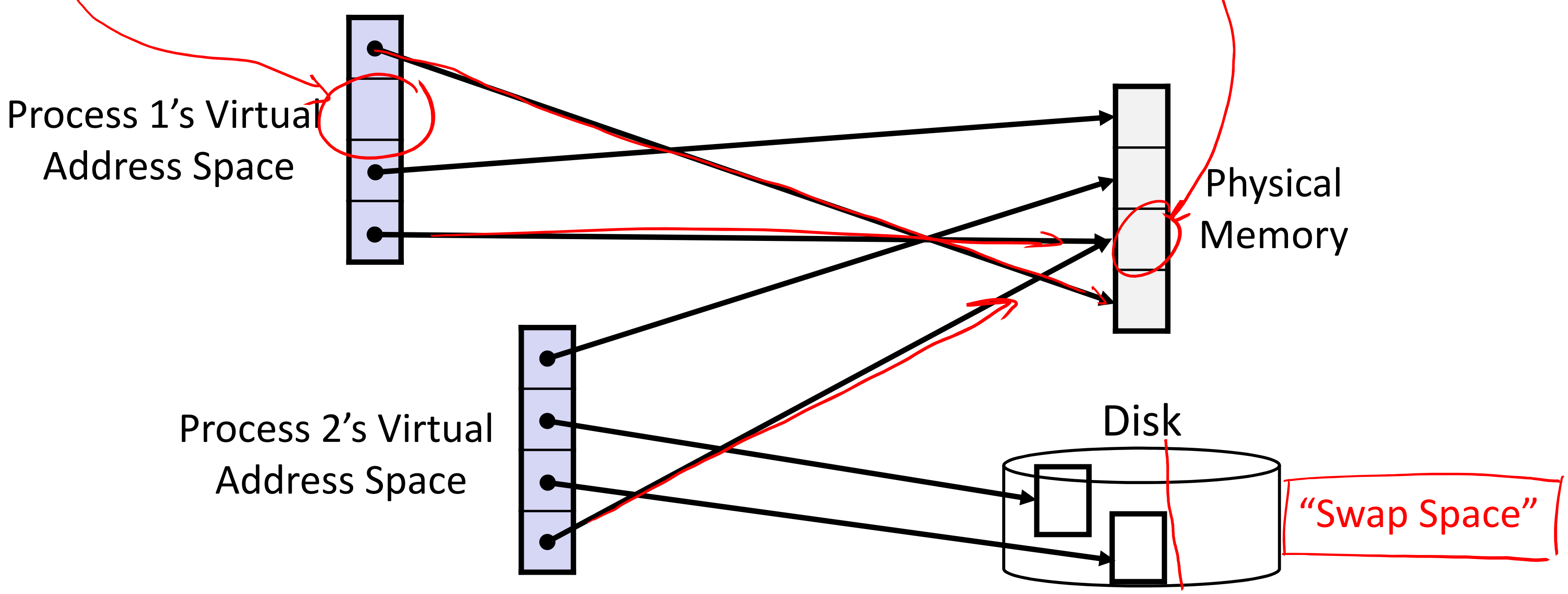
- Physical address space: Set of $\mathrm{M} = 2^m$ physical addr
  - {0, 1, 2, 3, …, M-1}

- Every byte in main memory has:
  - one physical address (PA)
  - zero, one, *or more* virtual addresses (VAs)

unused — used by one process — used by many processes

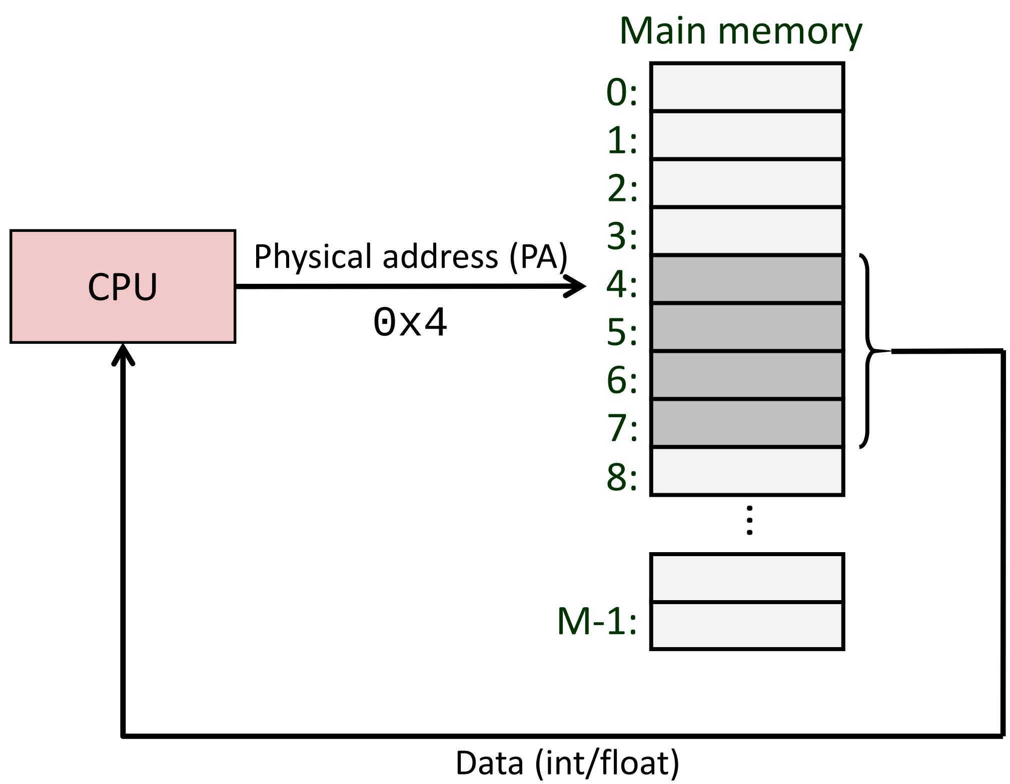
Mapping

- A virtual address (VA) can be mapped to either physical memory or disk
  - Unused VAs may not have a mapping
  - VAs from *different* processes may map to same location in memory/disk



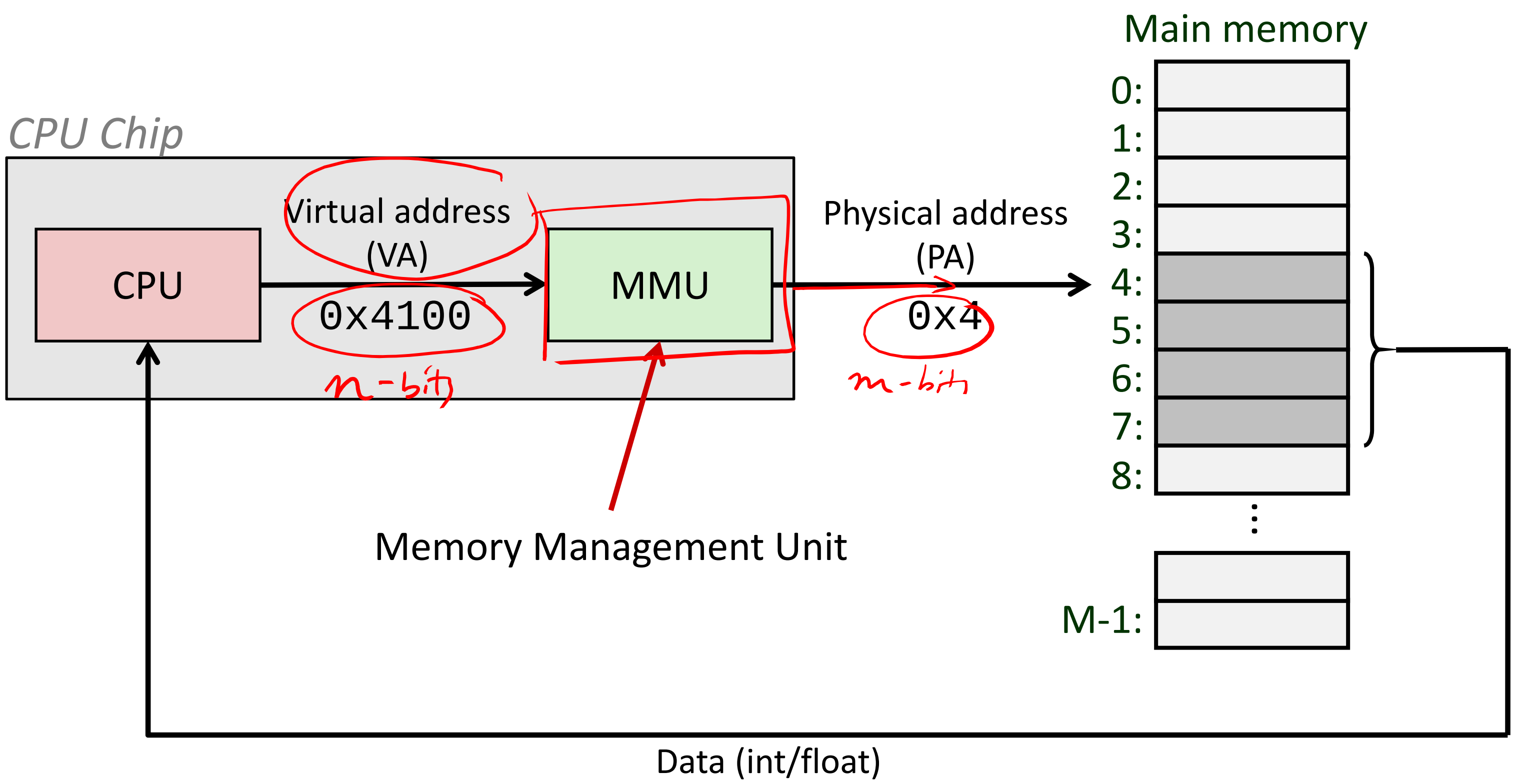
Aside:

A System Using Physical Addressing



- Used in "simple" systems with (usually) just one process:
  - Embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



- Physical addresses are *completely invisible to programs*
  - Used in all modern desktops, laptops, servers, smartphones…
  - One of the great ideas in computer science

Why Virtual Memory (VM)?

- Efficient use of limited main memory (RAM)
  - Use RAM as a cache for the parts of a virtual address space
    - Some non-cached parts stored on disk
    - Some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - Transfer data back and forth as needed

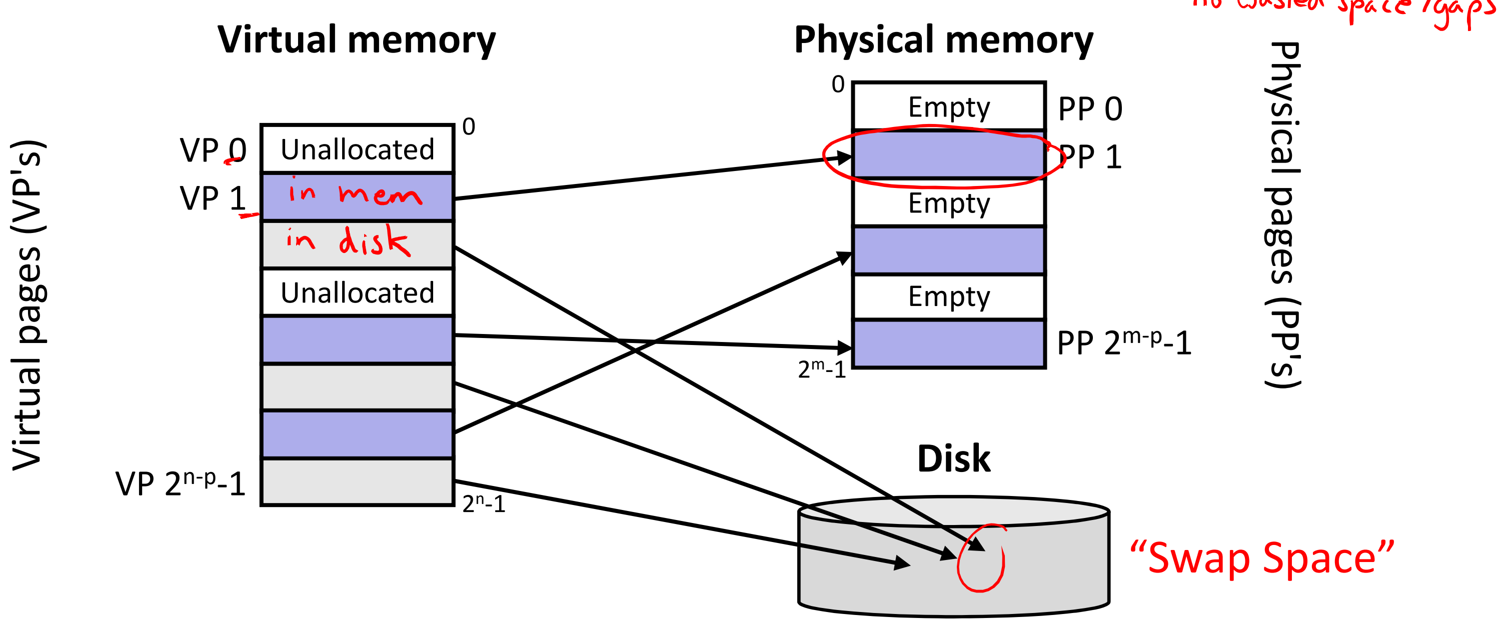
e.g. same ideas as caches; we have limited main memory, so we want to carefully allocate and evict data referred to by these virtual addresses from main memory.

- Simplifies memory management for programmers
  - Each process "gets" the same full, private linear address space
- Isolates address spaces (protection)
  - One process can't interfere with another's memory
    - They operate in *different address spaces*
  - User process cannot access privileged information
    - Different sections of address spaces have different permissions

VM and the Memory Hierarchy

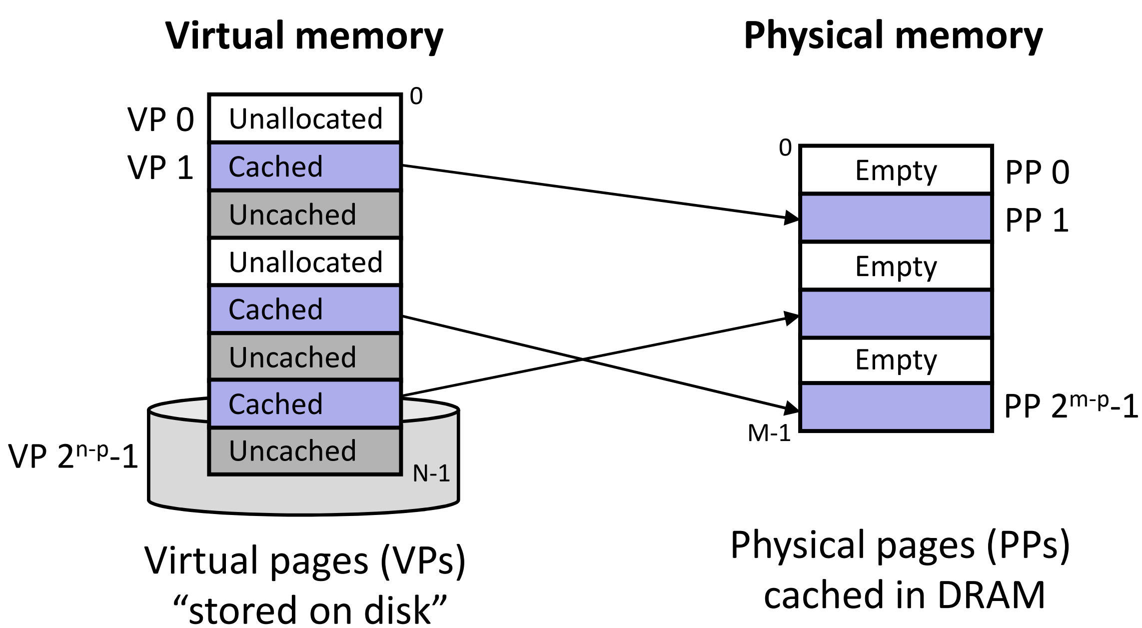
pages are like the blocks of our caches

- Think of virtual memory as array of $N = 2^n$ contiguous bytes
- *Pages* of virtual memory are usually stored in physical memory, but sometimes spill to disk
  - Pages are another unit of aligned memory (size is $P = 2^p$ bytes) $p = \lceil \log_2 P \rceil$
  - Each virtual page can be stored in *any* physical page (no fragmentation!) no wasted space/gaps



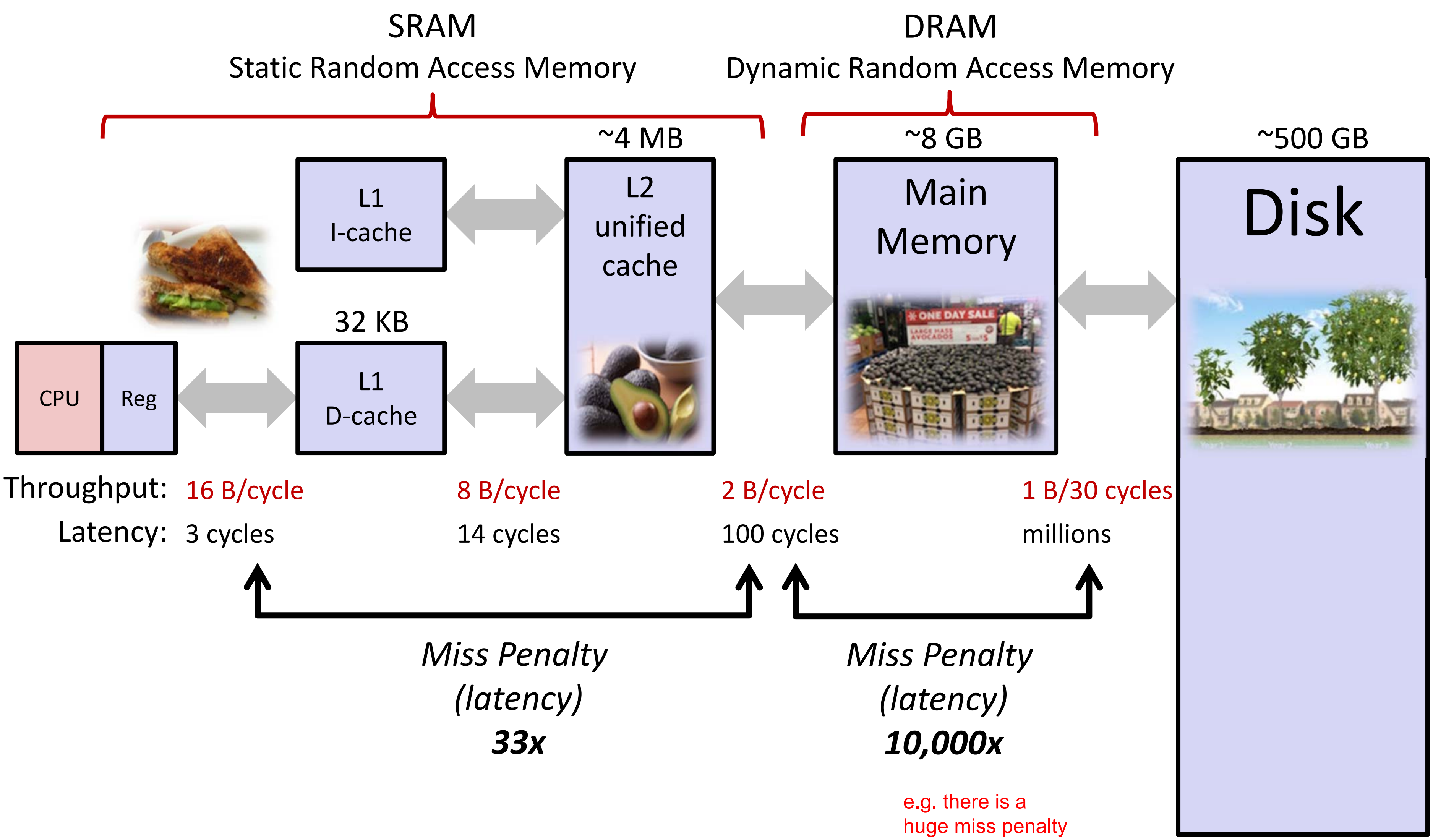
or: Virtual Memory as DRAM Cache for Disk

- Think of virtual memory as an array of $\mathrm{N} = 2^n$ contiguous bytes stored *on a disk*
- Then physical main memory is used as a *cache* for the virtual memory array
  - These "cache blocks" are called *pages* (size is $\mathrm{P} = 2^p$ bytes)



Memory Hierarchy: Core 2 Duo

Not drawn to scale



e.g. there is a
huge miss penalty
when reading from disk, so virtual memory
as a cache wants to lower the miss rate at all costs.

Virtual Memory Design Consequences

- Large page size: typically 4-8 KiB or 2-4 MiB
  - *Can* be up to 1 GiB (for "Big Data" apps on big computers)
  - Compared with 64-byte cache blocks
- Fully associative (physical memory is single set)
  - Any virtual page can be placed in any physical page
  - Requires a "large" mapping function – different from CPU caches
- Highly sophisticated, expensive replacement algorithms in OS
  - Too complicated and open-ended to be implemented in hardware
- *Write-back* rather than *write-through* (track dirty pages)
  - *Really* don't want to write to disk every time we modify something in memory again, our goal is to reduce the number of operations we have to perform on disk as drastically as possible because any such reduction inevitably increases performance no matter the cost as the miss penalty is so high
  - Some things may never end up on disk (*e.g.* stack for short-lived process)

Why does VM work on RAM/disk?

- Avoids disk accesses because of *locality*
  - Same reason that L1 / L2 / L3 caches work

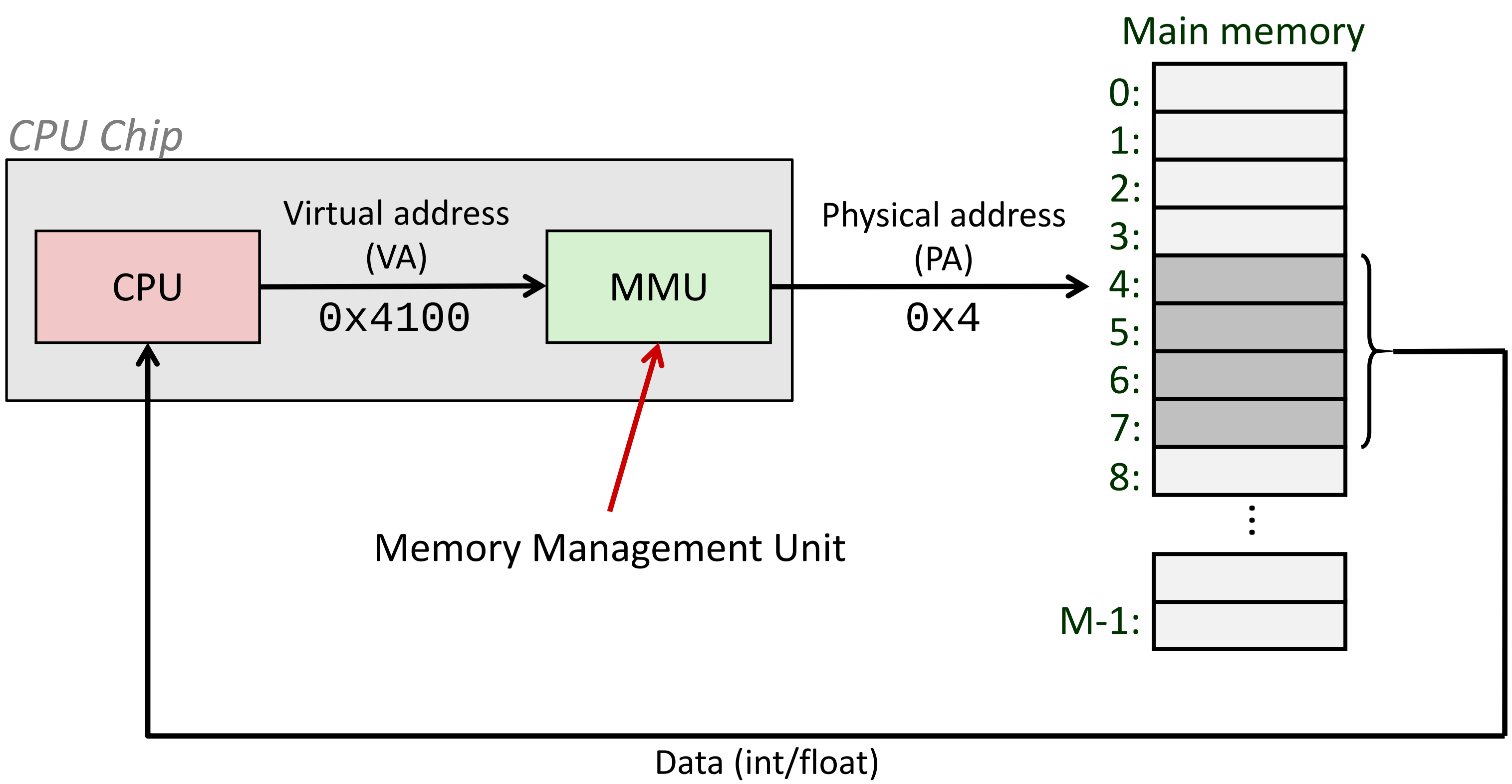
- The set of virtual pages that a program is "actively" accessing at any point in time is called its *working set*
  - If (*working set of one process* ≤ *physical memory*):
    - Good performance for one process (after compulsory misses)
  - If (*working sets of all processes* > *physical memory*):
    - ***Thrashing:*** Performance meltdown where pages are swapped between memory and disk continuously (CPU always waiting or paging)
    - This is why your computer can feel faster when you add RAM

Virtual Memory (VM)

- Overview and motivation
- VM as a tool for caching
- **Address translation**
- VM as a tool for memory management
- VM as a tool for memory protection

Address Translation

How do we perform the virtual → physical address translation?



Address Translation: Page Tables

VPN width $n-p \iff$ we have 2^{n-p} pages in VA space

page size P bytes $\iff p = \lceil \log_2 P \rceil$ bits

- CPU-generated address can be split into:

n-bit address: $\underbrace{\text{Virtual Page Number}}_{n-p \text{ bits}}$ $\underbrace{\text{Page Offset}}_{p \text{ bits}}$

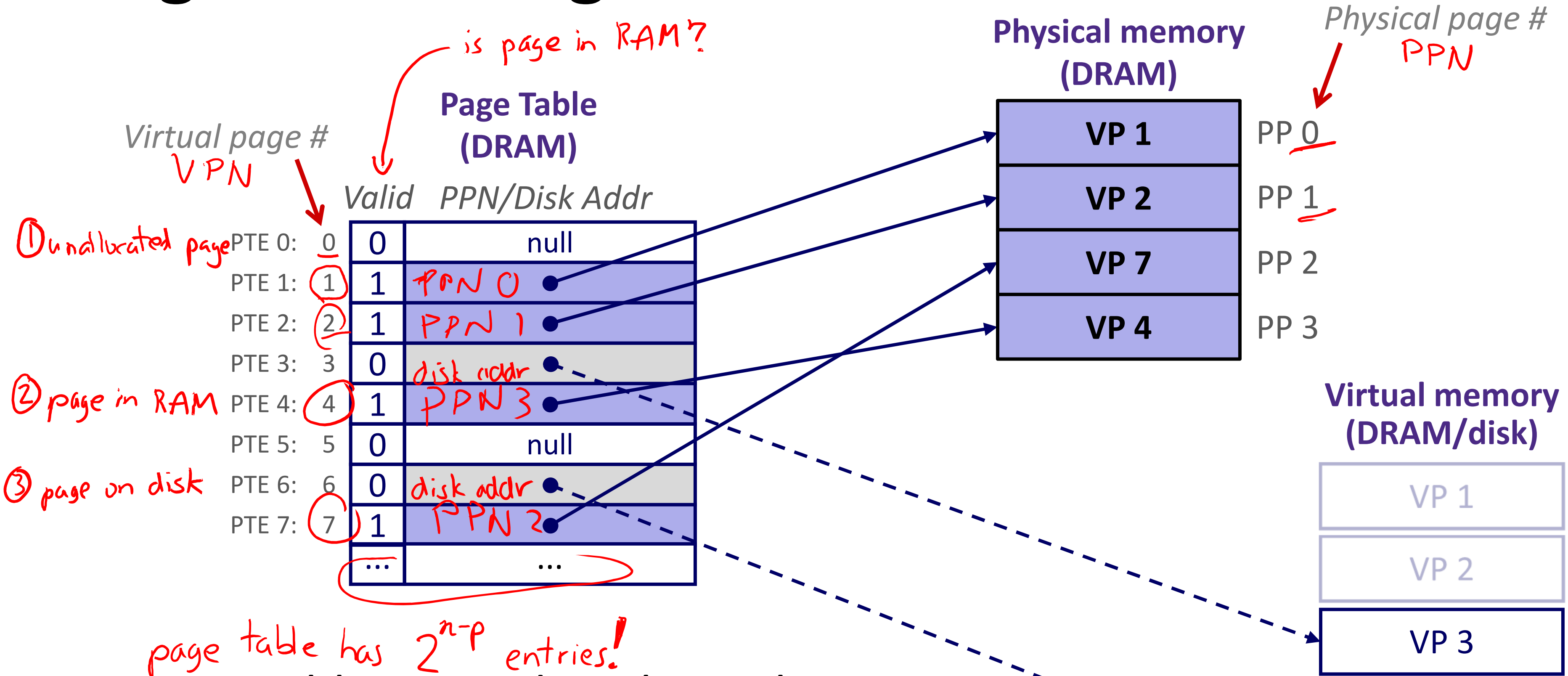
analogous to: | block number | block offset | for caches

  - Request is Virtual Address (VA), want Physical Address (PA)
  - Note that Physical Offset = Virtual Offset (page-aligned)

- Use lookup table that we call the *page table* (PT)
  - Replace Virtual Page Number (VPN) for Physical Page Number (PPN) to generate Physical Address
  - Index PT using VPN: page table entry (PTE) stores the PPN plus management bits (*e.g.* Valid, Dirty, access rights)
  - Has an entry for *every* virtual page – why?

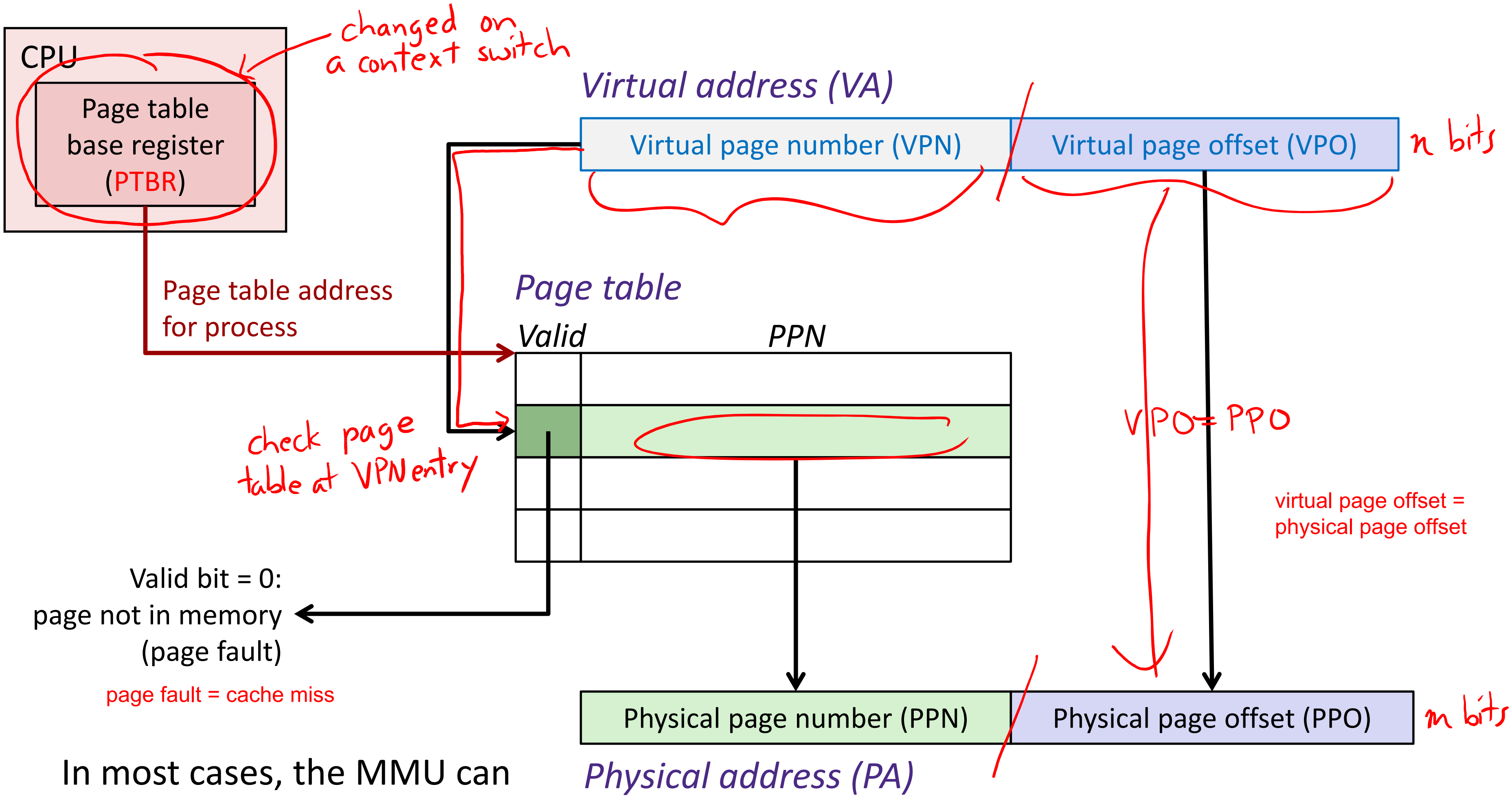
e.g. we just create a giant jump table!

Page Table Diagram



- Page tables stored in physical memory
  - Too big to fit elsewhere – managed by MMU & OS
- How many page tables in the system?
  - *One per process*

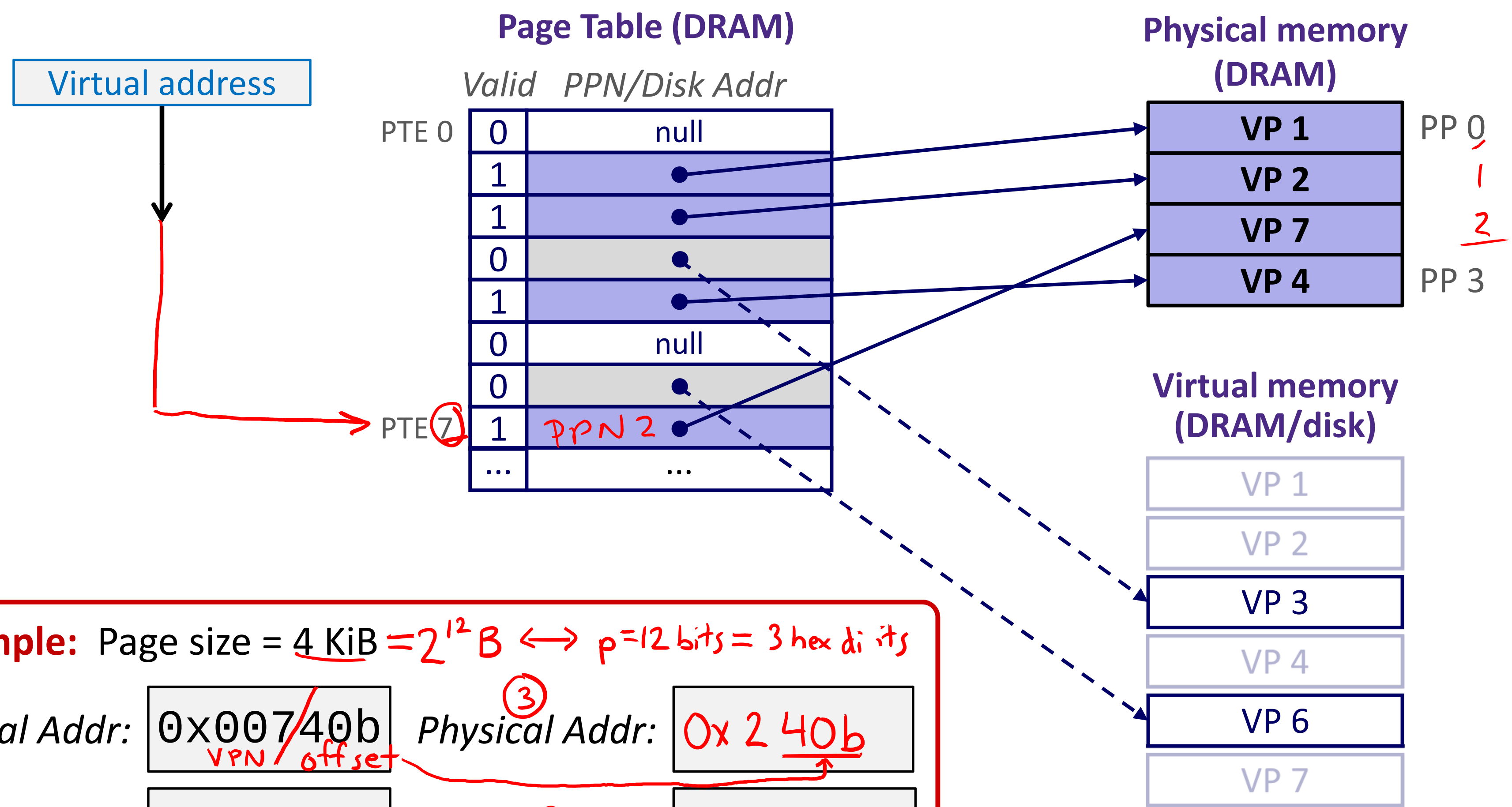
Page Table Address Translation



In most cases, the MMU can perform this translation without software assistance

Page Hit

- ***Page hit:*** VM reference is in physical memory



Example: Page size = 4 KiB $= 2^{12}$ B $\longleftrightarrow$ $p = 12$ bits $=$ 3 hex digits

Virtual Addr:	0x007/40b (VPN / offset)	③ *Physical Addr:*	0x 2 40b
① *VPN:*	7	② *PPN:*	2

Summary

- Virtual memory provides:
  - Ability to use limited memory (RAM) across multiple processes
  - Illusion of contiguous virtual address space for each process
  - Protection and sharing amongst processes
- Indirection via address mapping by page tables
  - Part of memory management unit and stored in memory
  - Use virtual page number as index into lookup table that holds physical page number, disk address, or NULL (unallocated page)
  - On page fault, throw exception and move page from swap space (disk) to main memory

BONUS SLIDES

Detailed examples:

- `wait()` example
- `waitpid()` example

wait() Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10() {
   pid_t pid[N];
   int i;
   int child_status;
   for (i = 0; i < N; i++)
      if ((pid[i] = fork()) == 0)
         exit(100+i); /* Child */
   for (i = 0; i < N; i++) {
      pid_t wpid = wait(&child_status);
      if (WIFEXITED(child_status))
         printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
      else
         printf("Child %d terminated abnormally\n", wpid);
   }
}
```

`waitpid()`: Waiting for a Specific Process

`pid_t waitpid(pid_t pid, int &status, int options)`

- suspends current process until specific process terminates
- various options (that we won't talk about)

```
void fork11() {
   pid_t pid[N];
   int i;
   int child_status;
   for (i = 0; i < N; i++)
      if ((pid[i] = fork()) == 0)
         exit(100+i); /* Child */
   for (i = 0; i < N; i++) {
      pid_t wpid = waitpid(pid[i], &child_status, 0);
      if (WIFEXITED(child_status))
         printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
      else
         printf("Child %d terminated abnormally\n", wpid);
   }
}
```