Storage Management

Goals of this Lecture

Help you learn about:

- Locality and caching
- Typical storage hierarchy
- **Virtual memory**
    - How the hardware and OS give application pgms the illusion of a large, contiguous, private address space

Virtual memory is one of the most important concepts in system programming

Agenda

Locality and caching

Typical storage hierarchy

Virtual memory

Improving Storage Device Performance

Facts:

- **CPU** performance is improving **quickly**
- **Storage device** performance is improving **slowly**
- Example:
  - Gap between CPU speed and main memory (RAM) performance is widening
  - Main memory (RAM) is performance bottleneck
    - Many programs stall the CPU waiting for loads and stores

Conclusion:

- To improve **overall** performance, must improve **storage device** performance

Improving Storage Performance

Classes of storage devices:

- **Fast access** & small capacity
- Slow access & **large capacity**

We want:

- **Fast access** & **large capacity**
- But how???

The key: **locality** allows **caching**

- Most programs exhibit good **locality**
- A program that exhibits good **locality** will benefit from proper **caching**

Locality

Two kinds of **locality**

- **Temporal** locality
  - If a pgm references item X now, it probably will reference X again soon
- **Spatial** locality
  - If a pgm references item X now, it probably will reference items in storage nearby X soon

Most programs exhibit good temporal and spatial locality

Locality Example

Locality example

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
```

Typical code
(good locality)

- **Temporal locality**
  - Data: Whenever the CPU accesses **`sum`**, it accesses **`sum`** again shortly thereafter
  - Instructions: Whenever the CPU executes **`sum += a[i]`**, it executes **`sum += a[i]`** again shortly thereafter
- **Spatial locality**
  - Data: Whenever the CPU accesses **`a[i]`**, it accesses **`a[i+1]`** shortly thereafter
  - Instructions: Whenever the CPU executes **`sum += a[i]`**, it executes **`i++`** shortly thereafter

Caching

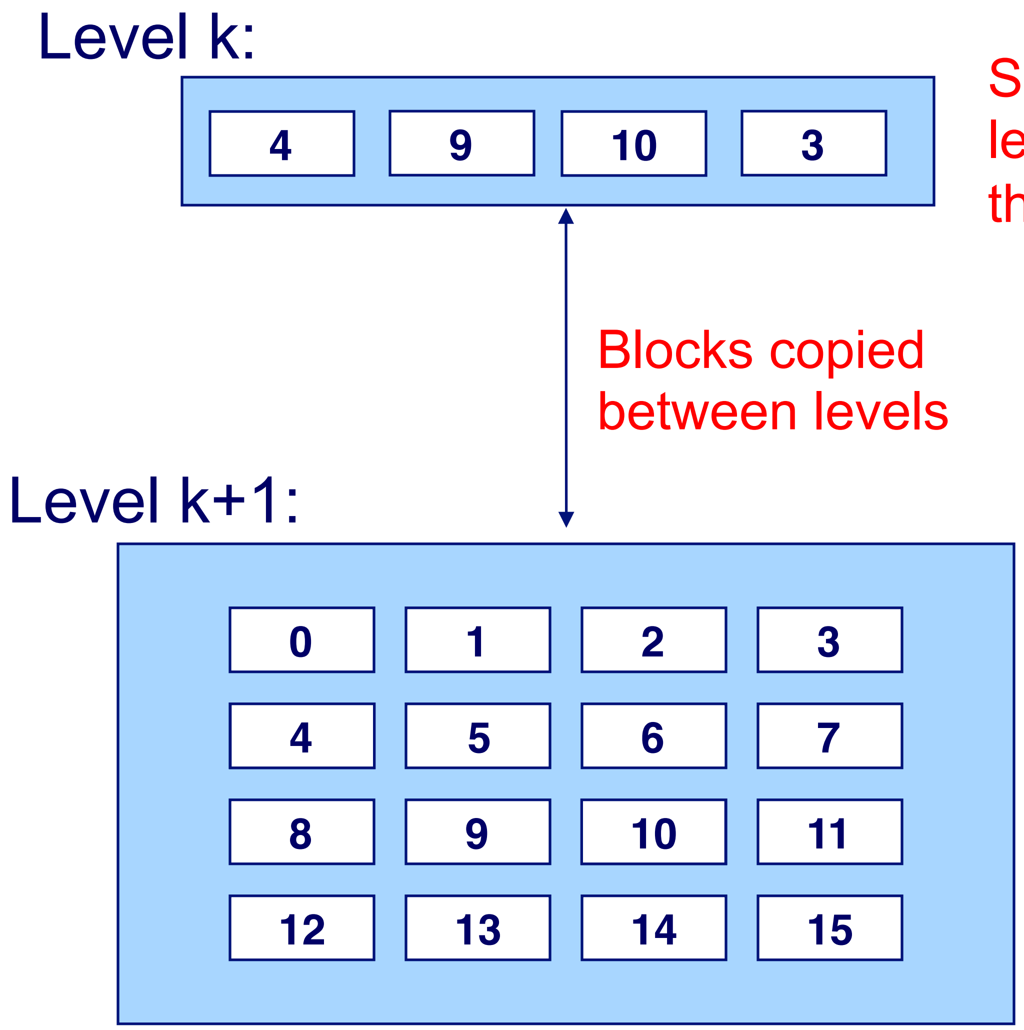
Cache

- Fast access, small capacity storage device
- Acts as a staging area for a subset of the items in a slow access, large capacity storage device

Good locality + proper caching

- => Most storage accesses can be satisfied by cache
- => Overall storage performance improved

Caching in a Storage Hierarchy



Smaller, faster device at level k caches a subset of the blocks from level k+1

Blocks copied between levels

Larger, slower device at level k+1 is partitioned into blocks

Cache Hits and Misses

Cache hit

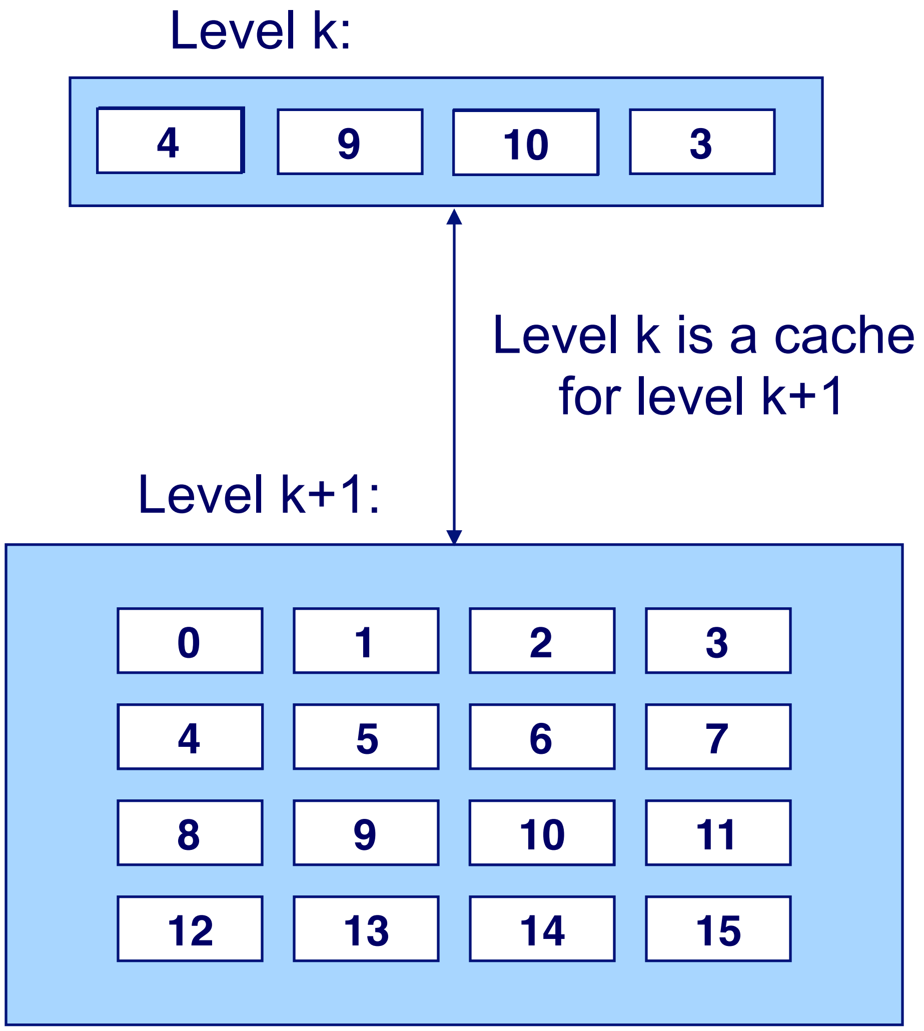
- E.g., request for block 10
- Access block 10 at level k
- Fast!

Cache miss

- E.g., request for block 8
- **Evict** some block from level k to level k+1
- Load block 8 from level k+1 to level k
- Access block 8 at level k
- Slow!

Caching goal:

- Maximize cache hits
- Minimize cache misses



Cache Eviction Policies

Best eviction policy: **“clairvoyant” policy**

- Always evict a block that is *never* accessed again, or…
- Always evict the block accessed the *furthest in the future*
- Impossible in the general case

Worst eviction policy

- Always evict the block that will be accessed next!
- Causes **thrashing**
- Impossible in the general case!

Cache Eviction Policies

Reasonable eviction policy: **LRU policy**

- Evict the “least recently used” (LRU) block
  - With the assumption that it will not be used again (soon)
- Good for straight-line code
- Bad for loops
- Expensive to implement
  - Often simpler approximations are used
  - See Wikipedia “Page replacement algorithm” topic

Locality/Caching Example: Matrix Mult

Matrix multiplication

- Matrix = two-dimensional array
- Multiply n-by-n matrices A and B
- Store product in matrix C

Performance depends upon

- Effective use of caching (as implemented by **system**)
- Good locality (as implemented by **you)**

Locality/Caching Example: Matrix Mult

Two-dimensional arrays are stored in either **row-major** or **column-major** order

a	0	1	2
0	18	19	20
1	21	22	23
2	24	25	26

row-major		col-major	
a[0][0]	18	a[0][0]	18
a[0][1]	19	a[1][0]	21
a[0][2]	20	a[2][0]	24
a[1][0]	21	a[0][1]	19
a[1][1]	22	a[1][1]	22
a[1][2]	23	a[2][1]	25
a[2][0]	24	a[0][2]	20
a[2][1]	25	a[1][2]	23
a[2][2]	26	a[2][2]	26

C uses **row-major** order

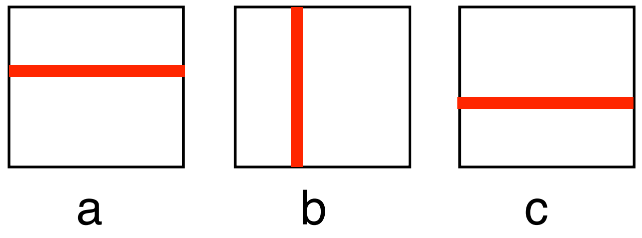
- Access in row-major order => good spatial locality
- Access in column-major order => poor spatial locality

Locality/Caching Example: Matrix Mult

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      for (k=0; k<n; k++)
         c[i][j] += a[i][k] * b[k][j];
```

Reasonable cache effects

- Good locality for A
- Bad locality for B
- Good locality for C

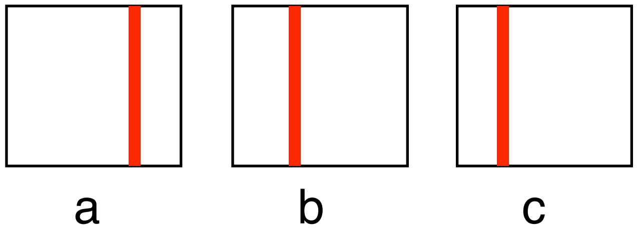


Locality/Caching Example: Matrix Mult

```
for (j=0; j<n; j++)
    for (k=0; k<n; k++)
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * b[k][j];
```

Poor cache effects

- Bad locality for A
- Bad locality for B
- Bad locality for C

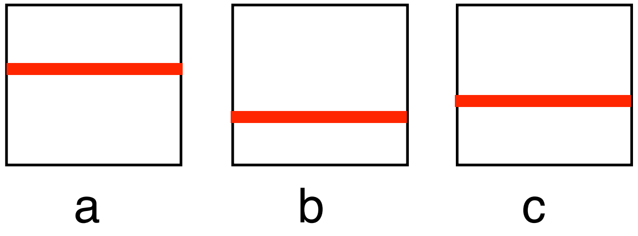


Locality/Caching Example: Matrix Mult

```
for (i=0; i<n; i++)
   for (k=0; k<n; k++)
      for (j=0; j<n; j++)
         c[i][j] += a[i][k] * b[k][j];
```

Good cache effects

- Good locality for A
- Good locality for B
- Good locality for C



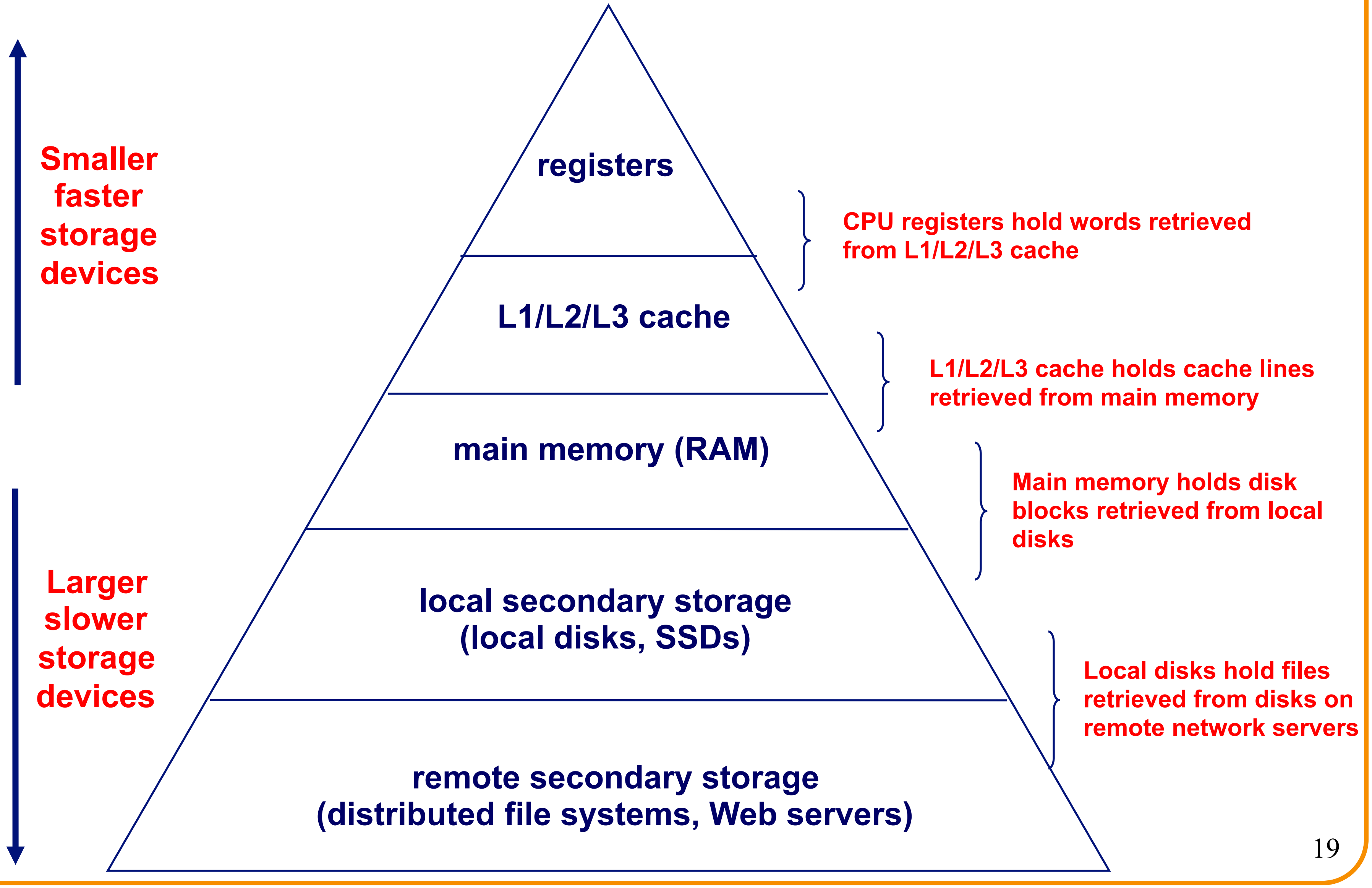
Agenda

Locality and caching

Typical storage hierarchy

Virtual memory

Typical Storage Hierarchy



Smaller faster storage devices
registers
CPU registers hold words retrieved from L1/L2/L3 cache
L1/L2/L3 cache
L1/L2/L3 cache holds cache lines retrieved from main memory
main memory (RAM)
Main memory holds disk blocks retrieved from local disks
Larger slower storage devices
local secondary storage (local disks, SSDs)
Local disks hold files retrieved from disks on remote network servers
remote secondary storage (distributed file systems, Web servers)

Typical Storage Hierarchy

Registers

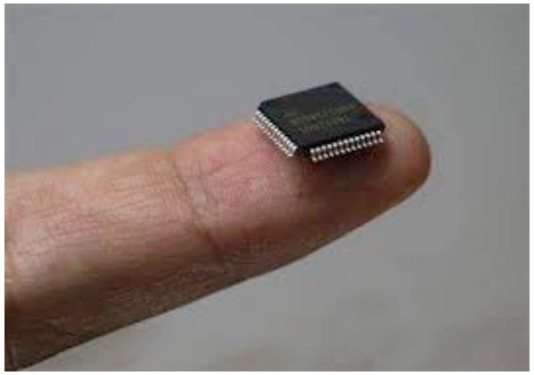
- **Latency**: 0 cycles
- **Capacity**: 8-256 registers
  - 8 general purpose registers in IA-32; 128 in Itanium

L1/L2/L3 Cache

- **Latency**: 1 to 30 cycles
- **Capacity**: 32KB to 32MB

Main memory (RAM)

- **Latency**: ~100 cycles
  - 100 times slower than registers
- **Capacity**: 256MB to 64GB



Typical Storage Hierarchy

Local secondary storage: disk drives

- **Latency**: ~100,000 cycles
  - 1000 times slower than main mem
  - Limited by nature of disk
    - Must move heads and wait for data to rotate under heads
    - Faster when accessing many bytes in a row
- **Capacity**: 1GB to 256TB

Typical Storage Hierarchy

Remote secondary storage

- **Latency**: ~10,000,000 cycles
  - 100 times slower than disk
  - Limited by network bandwidth
- **Capacity**: essentially unlimited

Aside: Persistence

Another dimension: **persistence**

- Do data persist in the absence of power?

Lower levels of storage hierarchy store data persistently

- Remote secondary storage
- Local secondary storage

Higher levels of storage hierarchy **do not** store data persistently

- Main memory (RAM)
- L1/L2/L3 cache
- Registers

Aside: Persistence

Admirable goal: Move persistence upward in hierarchy

Solid state (flash) drives

- Use solid state technology (as does main memory)
- Persistent, as is disk
- Viable replacement for disk as local secondary storage

Storage Hierarchy & Caching Issues

Issue: Block size?

- Slow data transfer between levels k and k+1
    - => use large block sizes at k and k+1 (do data transfer less often)
- Fast data transfer between levels k and k+1
    - => use small block sizes at k and k+1 (reduce risk of cache miss)
- Lower in pyramid => slower data transfer => larger block sizes

Device	Block Size
Register	4 bytes
L1/L2/L3 cache line	32 bytes
Main memory page	4KB (4096 bytes)
Disk block	4KB (4096 bytes)
Disk transfer block	4KB (4096 bytes) to 64MB (67108864 bytes)

Storage Hierarchy & Caching Issues

Issue: Who manages the cache?

Device	Managed by:
Registers (cache of L1/L2/L3 cache and main memory)	**Compiler**, using complex code-analysis techniques **Assembly lang programmer**
L1/L2/L3 cache (cache of main memory)	**Hardware**, using simple algorithms
Main memory (cache of local sec storage)	**Hardware and OS**, using virtual memory concept with complex algorithms (since accessing disk is expensive)
Local secondary storage (cache of remote sec storage)	**End user**, by deciding which files to download

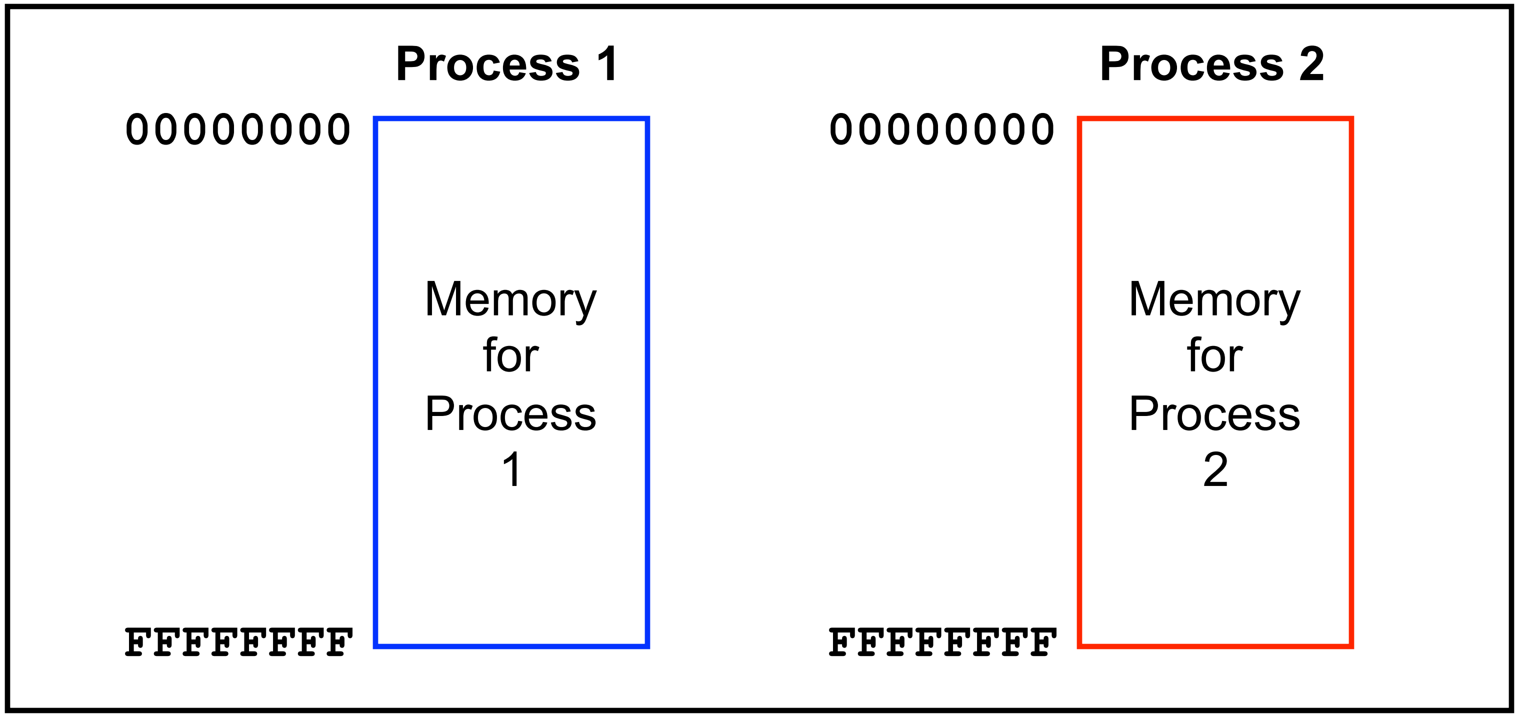
Agenda

Locality and caching

Typical storage hierarchy

Virtual memory

Main Memory: Illusion

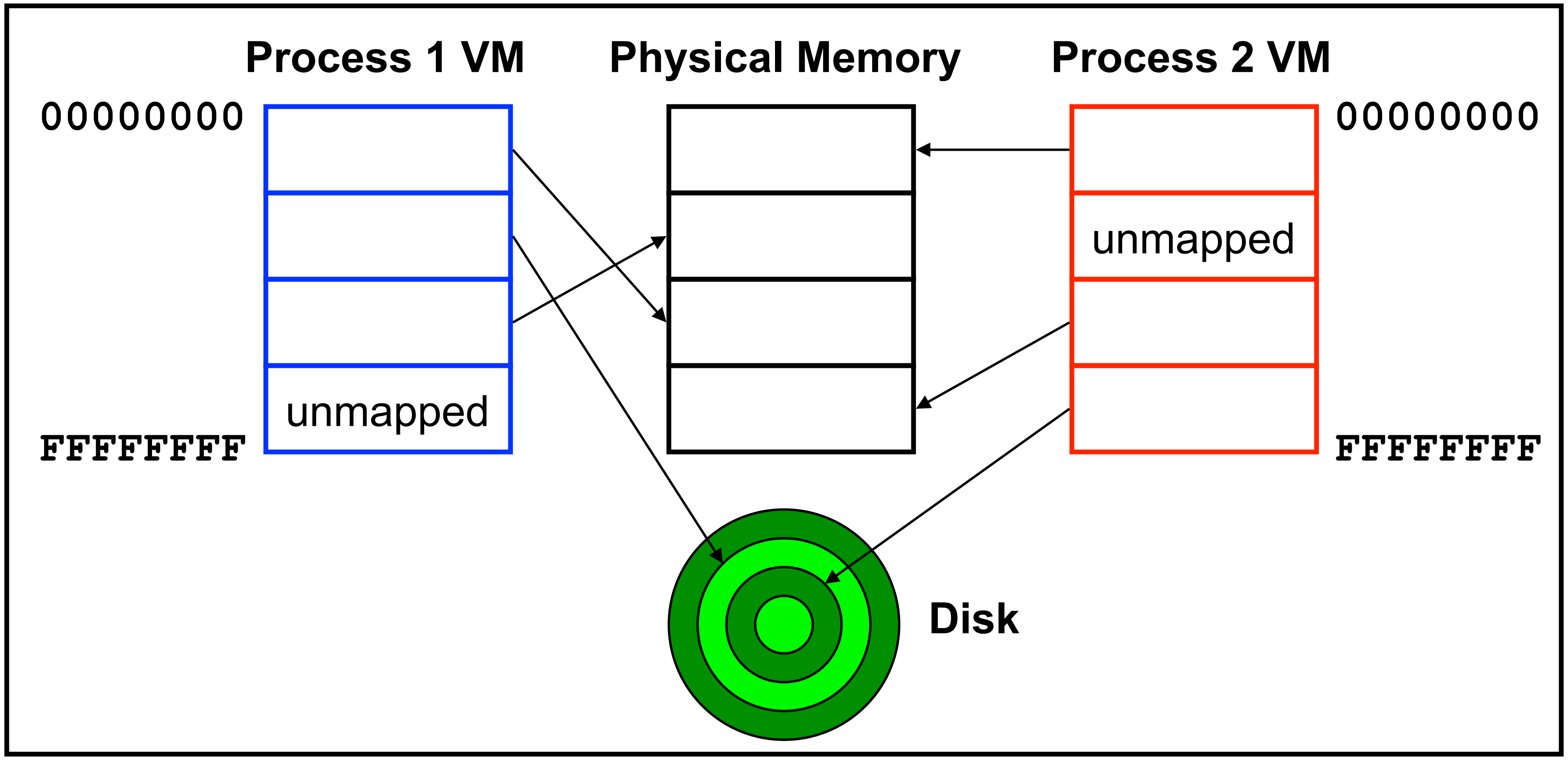


Each process sees main memory as

Large: 2^{32} = 4 GB of memory

Uniform: contiguous memory locations from 0 to 2^{32}-1

Main Memory: Reality



Memory is divided into **pages**

- At any time some pages are in physical memory, some on disk
- OS and hardware swap pages between physical memory and disk

Multiple processes share physical memory

Virtual & Physical Addresses

Question

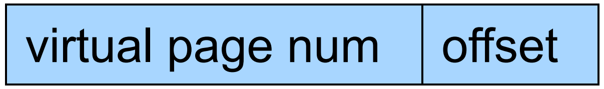
- How do OS and hardware implement virtual memory?

Answer (part 1)

- Distinguish between **virtual addresses** and **physical addresses**

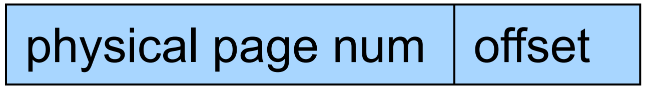
Virtual & Physical Addresses (cont.)

Virtual address



- Identifies a location in a particular process's virtual memory
  - Independent of size of physical memory
  - Independent of other concurrent processes
- Consists of virtual page number & offset
- Used by **application programs**

Physical address

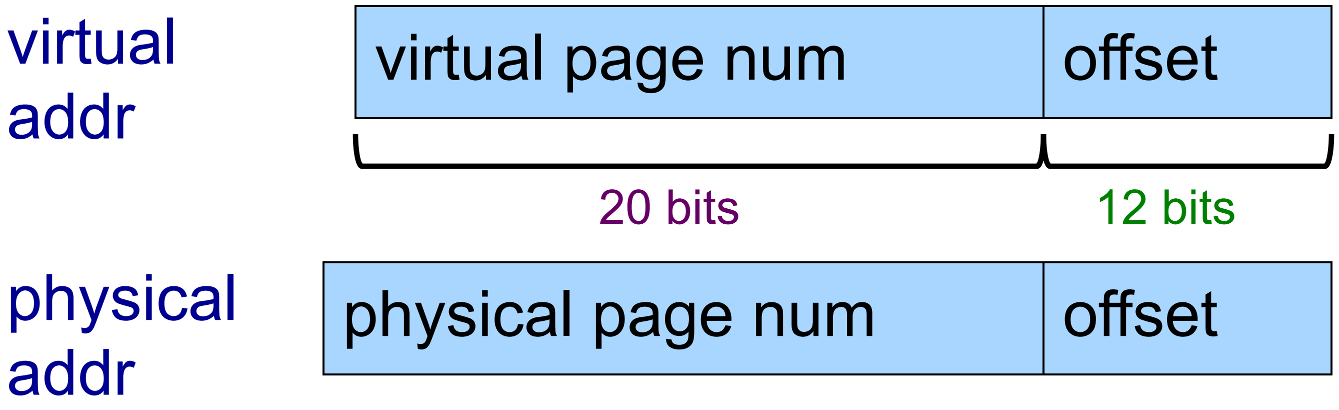


- Identifies a location in physical memory
- Consists of physical page number & offset
- Known only to **OS** and **hardware**

Note:

- Offset is same in virtual addr and corresponding physical addr

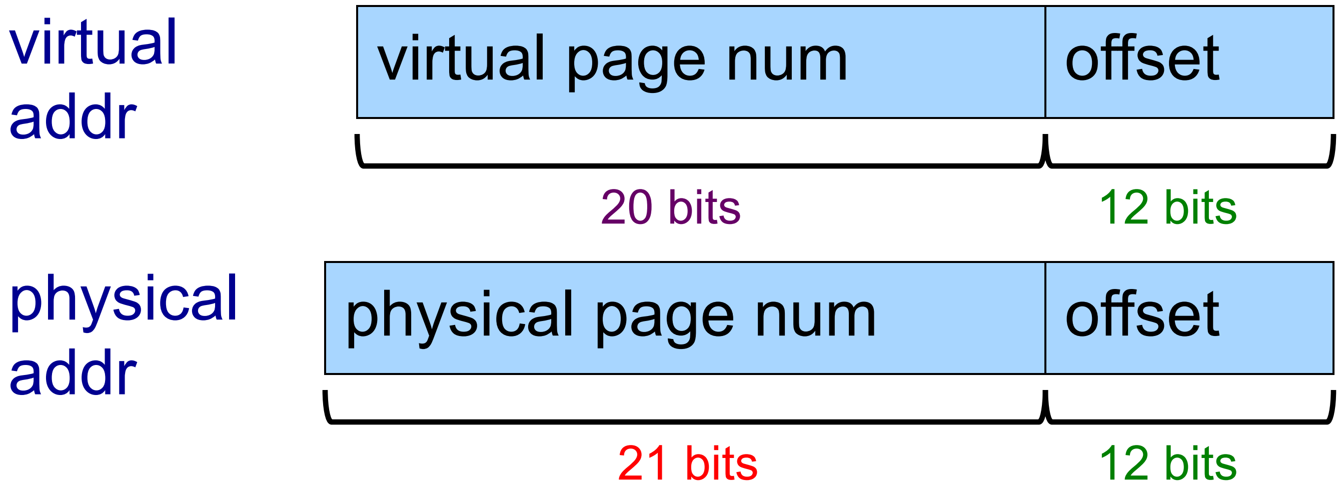
Nobel Virtual & Physical Addresses



On nobel with gcc217:

- Each offset is 12 bits
  - Each page consists of 2^{12} = 4K = 4096 bytes
- Each virtual page number consists of 20 bits
  - There are 2^{20} = 1M = 1,048,576 virtual pages
- Each virtual address consists of 32 bits
  - There are 2^{32} = 4G bytes of virtual memory (per process)

Nobel Virtual & Physical Addresses



On nobel with gcc217:

- Each offset is 12 bits
  - Each page consists of 2^{12} = 4K = 4096 bytes
- Each physical page number consists of 21 bits
  - There are 2^{21} = 2M = 2,097,152 physical pages
- Each physical address consists of 33 bits
  - There are 2^{33} = 8G bytes of physical memory (per CPU)

Page Tables

Question

- How do OS and hardware implement virtual memory?

Answer (part 2)

- Maintain a **page table** for each process

Page Tables (cont.)

Page Table for Process 1234

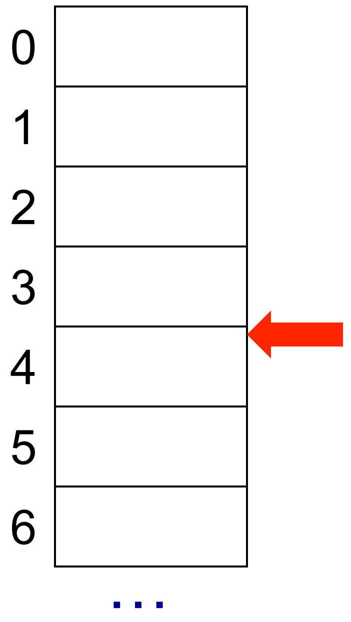
Virtual Page Num	Physical Page Num or Disk Addr
0	Physical page 5
1	(unmapped)
2	Spot X on disk
3	Physical page 8
…	…

Page table maps each in-use virtual page to:

- A physical page, or
- A spot (track & sector) on disk

Virtual Memory Example 1

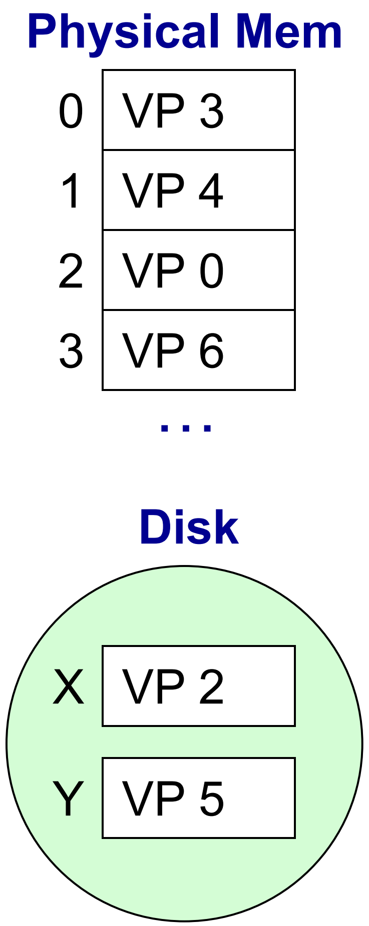
Process 1234 Virtual Mem



Process 1234 Page Table

VP	PP
0	2
1	
2	X
3	0
4	1
5	Y
6	3

. . .



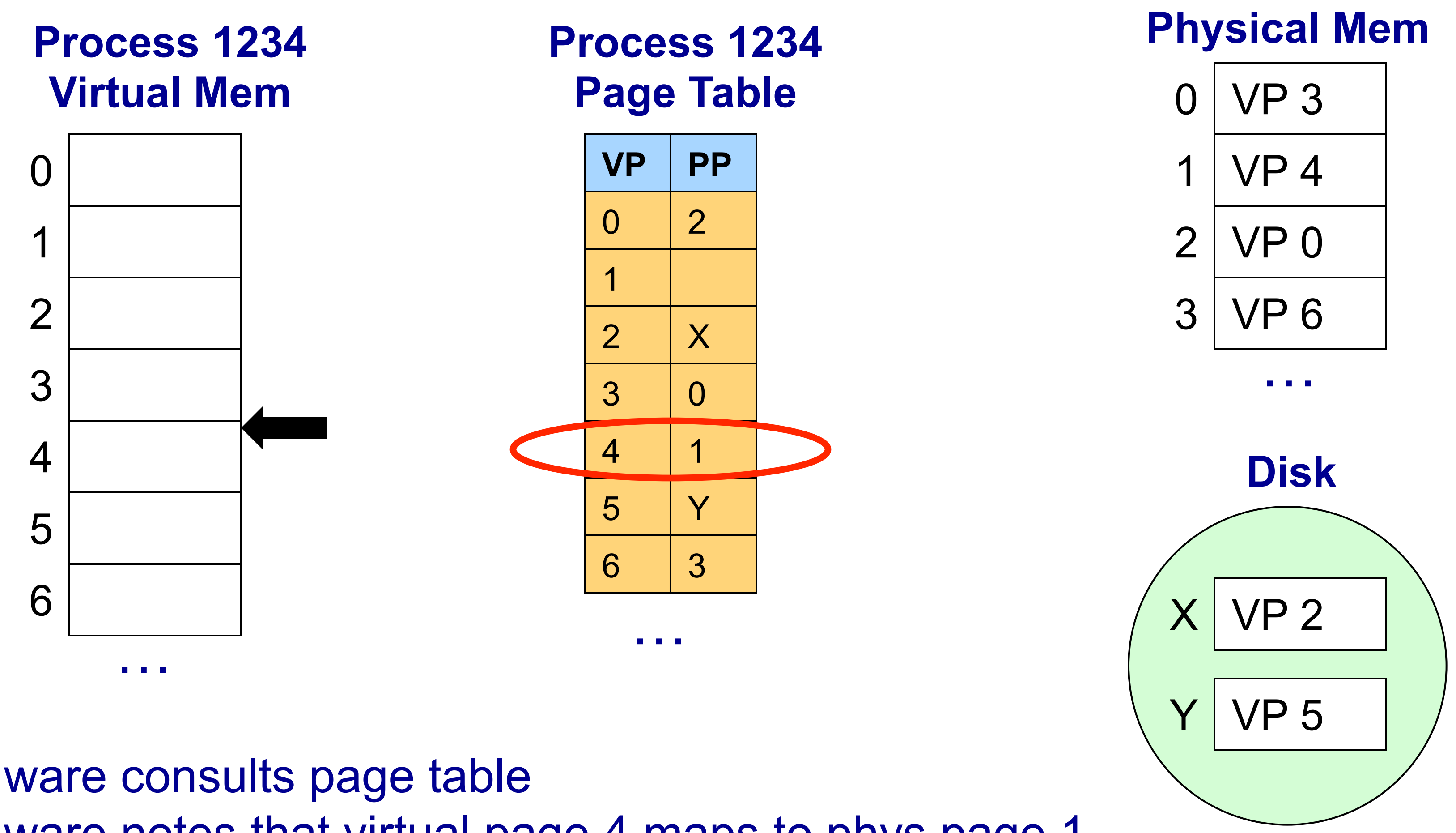
Process 1234 accesses mem at virtual addr 16386

16386 =

$00000000000000000100000000000010_B$ =

Virtual page num = 4; offset = 2

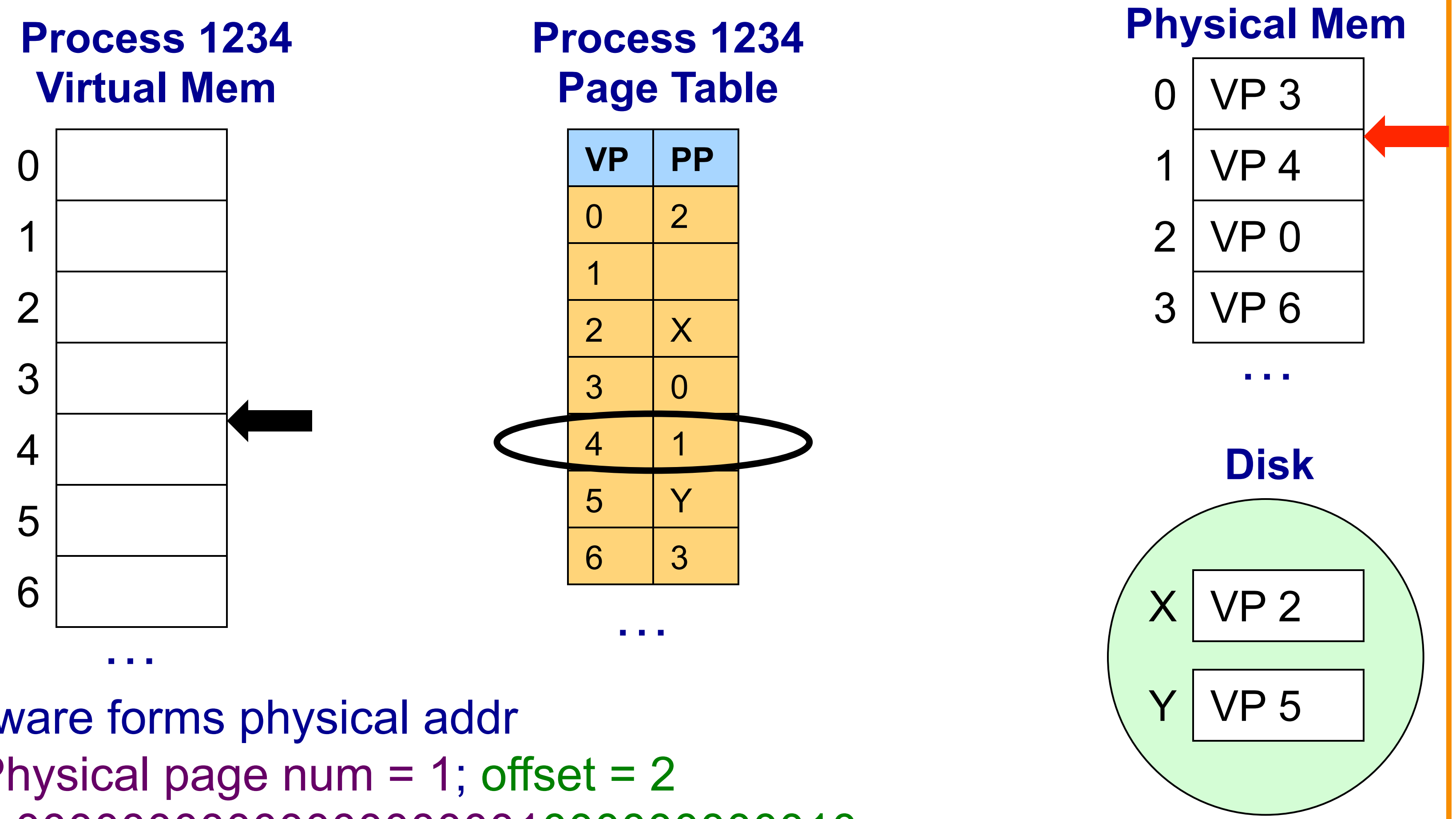
Virtual Memory Example 1 (cont.)



VP	PP
0	2
1	
2	X
3	0
4	1
5	Y
6	3

Hardware consults page table
Hardware notes that virtual page 4 maps to phys page 1
Page hit!

Virtual Memory Example 1 (cont.)



Hardware forms physical addr

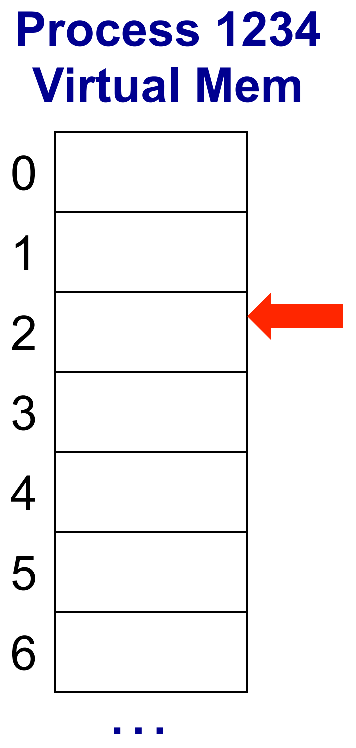
Physical page num = 1; offset = 2

= $00000000000000000001000000000010_B$

= 4098

Hardware fetches/stores data from/to phys addr 4098

Virtual Memory Example 2



Process 1234
Page Table

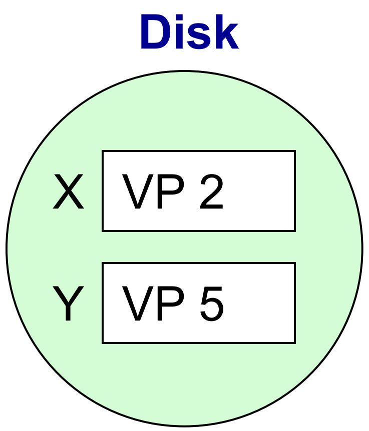
VP	PP
0	2
1	
2	X
3	0
4	1
5	Y
6	3

. . .

Physical Mem

0	VP 3
1	VP 4
2	VP 0
3	VP 6

. . .



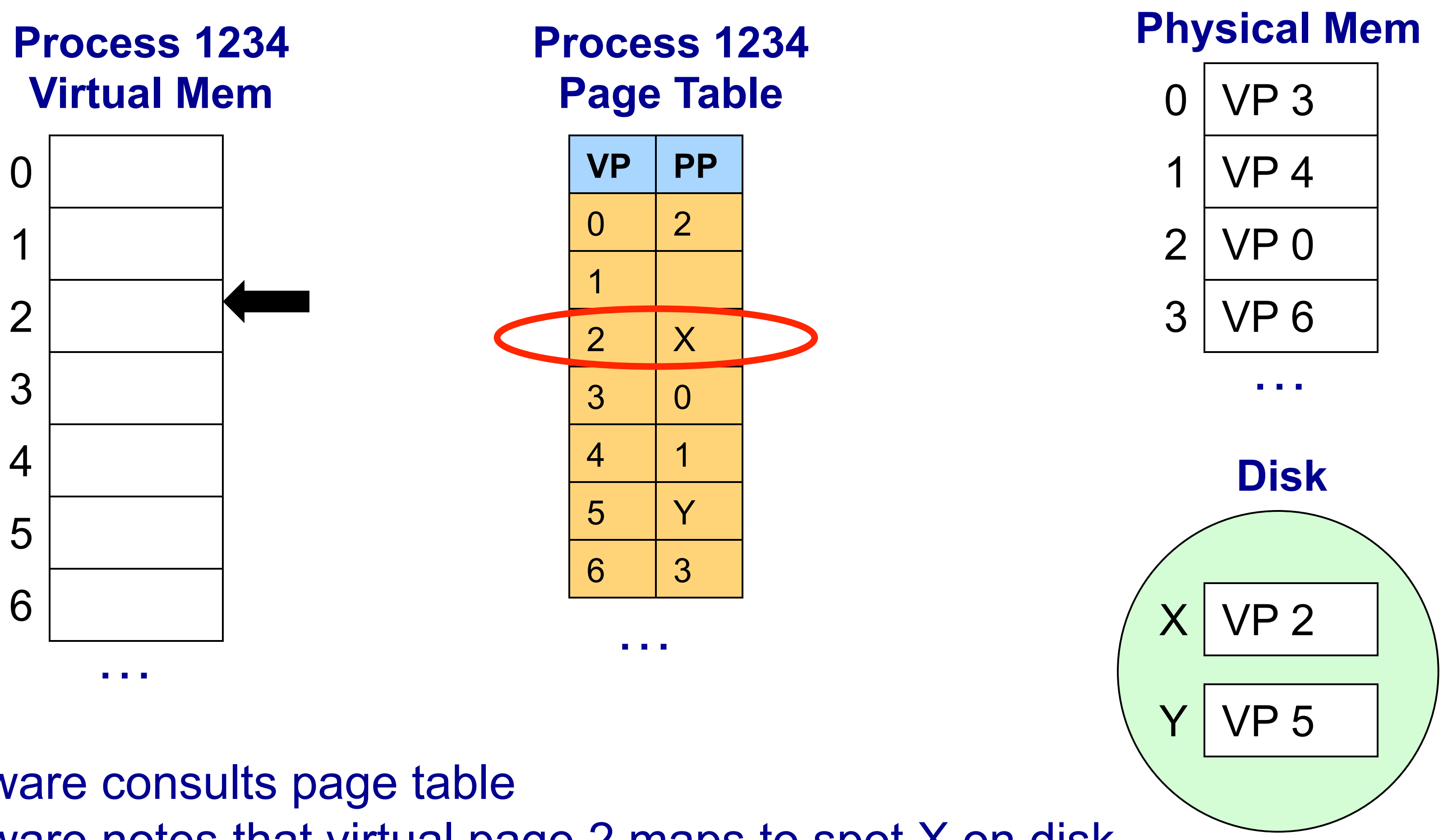
Process 1234 accesses mem at virtual addr 8200

8200 =

$00000000000000000001000000001000_B$ =

Virtual page num = 2; offset = 8

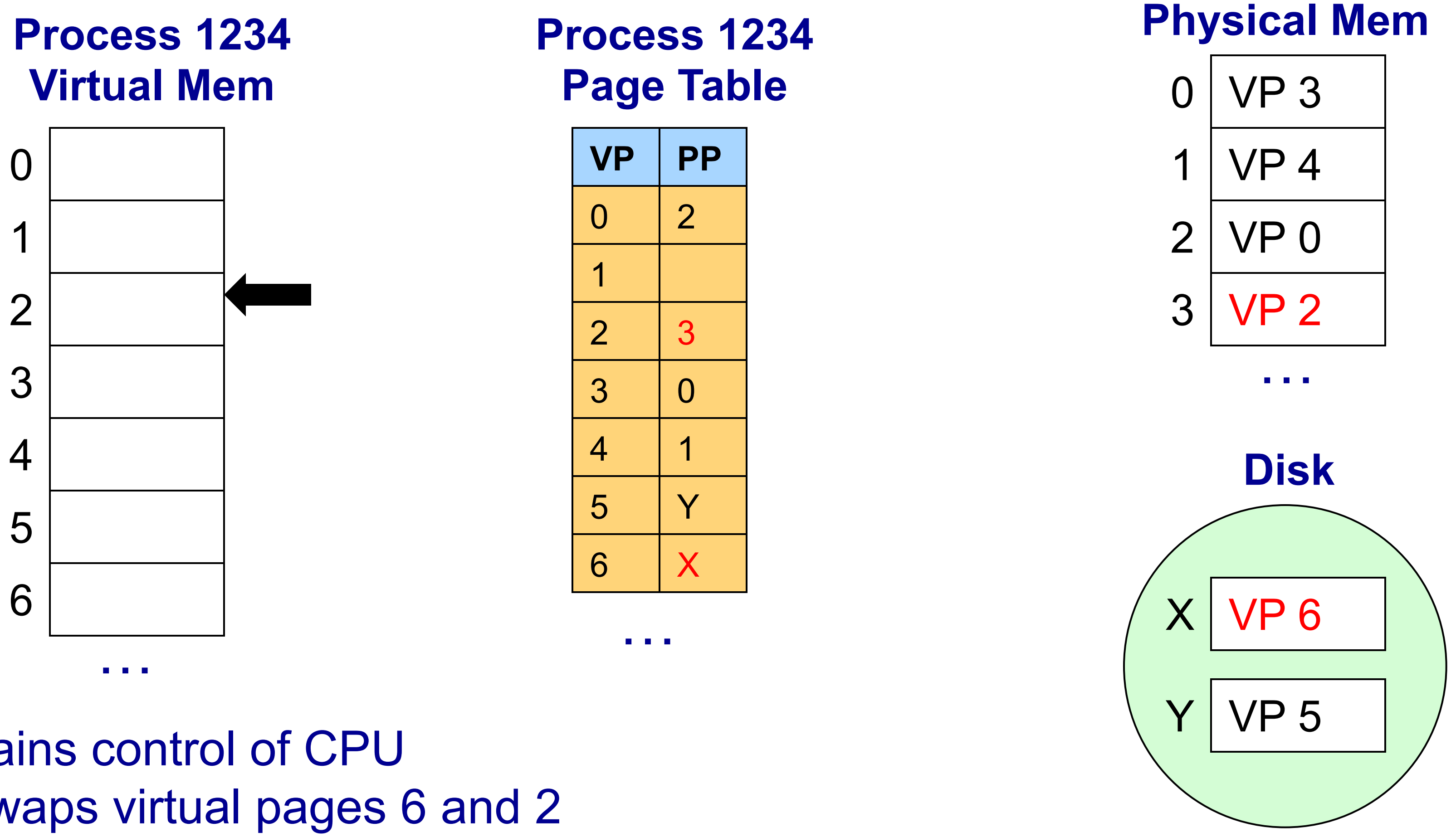
Virtual Memory Example 2 (cont.)



VP	PP
0	2
1	
2	X
3	0
4	1
5	Y
6	3

Hardware consults page table
Hardware notes that virtual page 2 maps to spot X on disk
Page miss!
Hardware generates **page fault**

Virtual Memory Example 2 (cont.)



Process 1234 Page Table

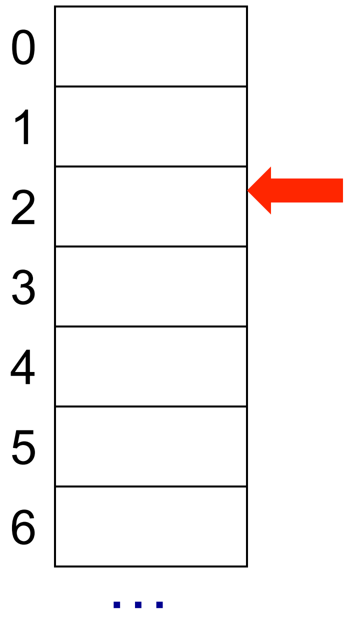
VP	PP
0	2
1	
2	3
3	0
4	1
5	Y
6	X

…

OS gains control of CPU
OS swaps virtual pages 6 and 2
OS updates page table accordingly
Control returns to process 1234
Process 1234 re-executes **same instruction**

Virtual Memory Example 2 (cont.)

Process 1234 Virtual Mem



Process 1234 Page Table

VP	PP
0	2
1	
2	3
3	0
4	1
5	Y
6	X

…

Physical Mem

0	VP 3
1	VP 4
2	VP 0
3	VP 2

…

Disk

X	VP 6
Y	VP 5

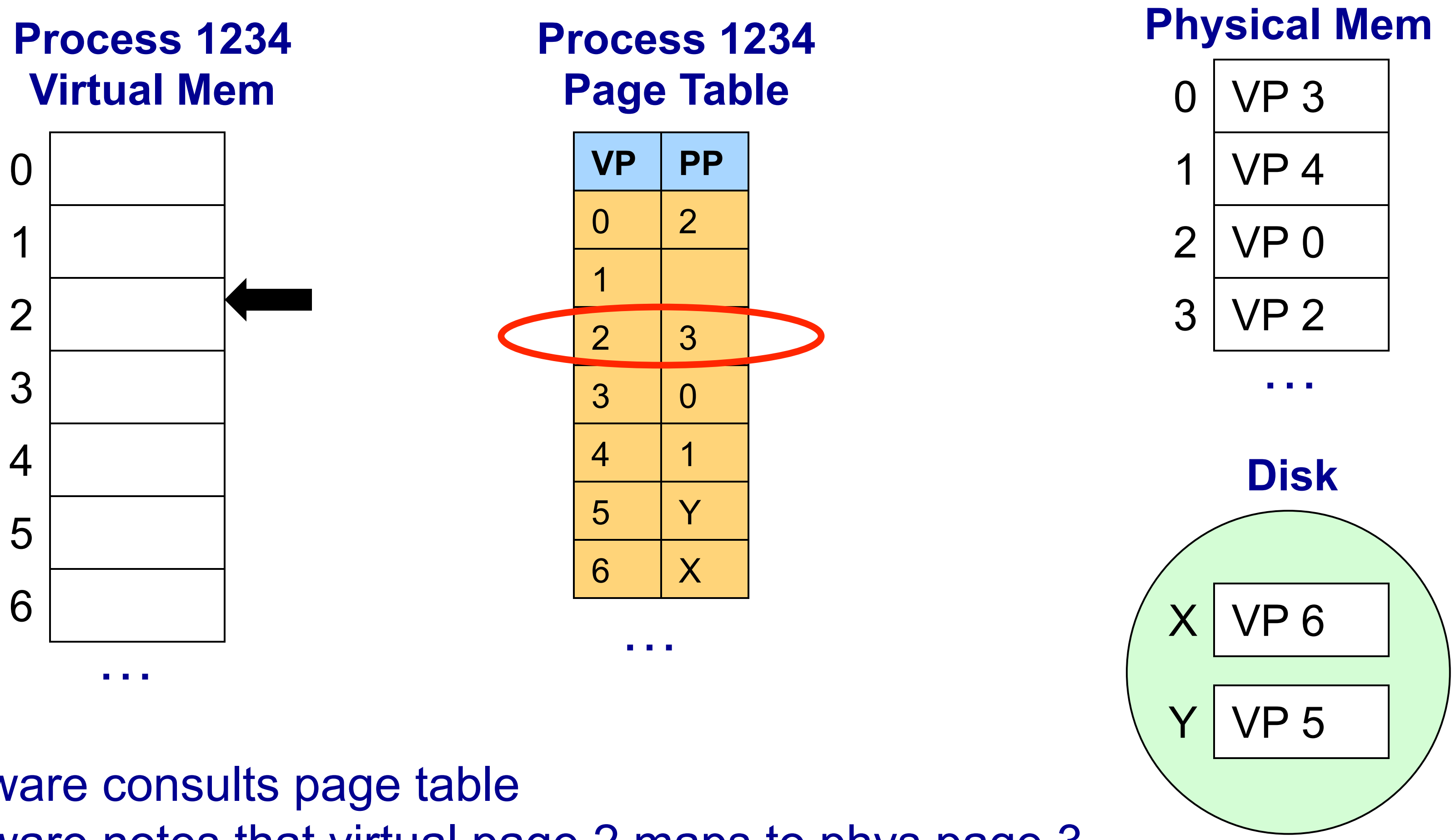
Process 1234 accesses mem at virtual addr 8200

8200 =

00000000000000000100000000001000$_B$ =

Virtual page num = 2; offset = 8

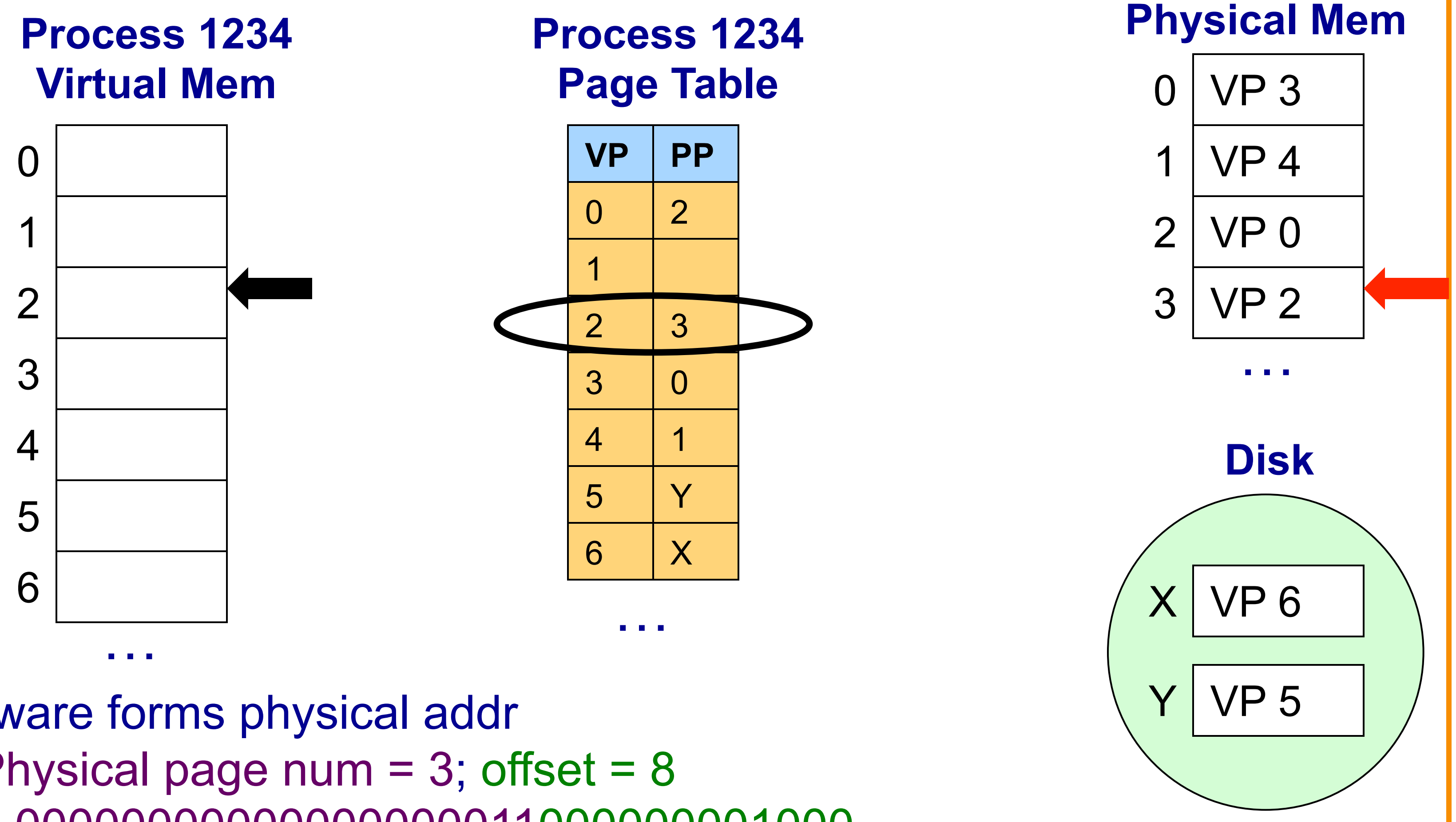
Virtual Memory Example 2 (cont.)



VP	PP
0	2
1	
2	3
3	0
4	1
5	Y
6	X

Hardware consults page table
Hardware notes that virtual page 2 maps to phys page 3
Page hit!

Virtual Memory Example 2 (cont.)



Process 1234 Page Table

VP	PP
0	2
1	
2	3
3	0
4	1
5	Y
6	X

...

Hardware forms physical addr

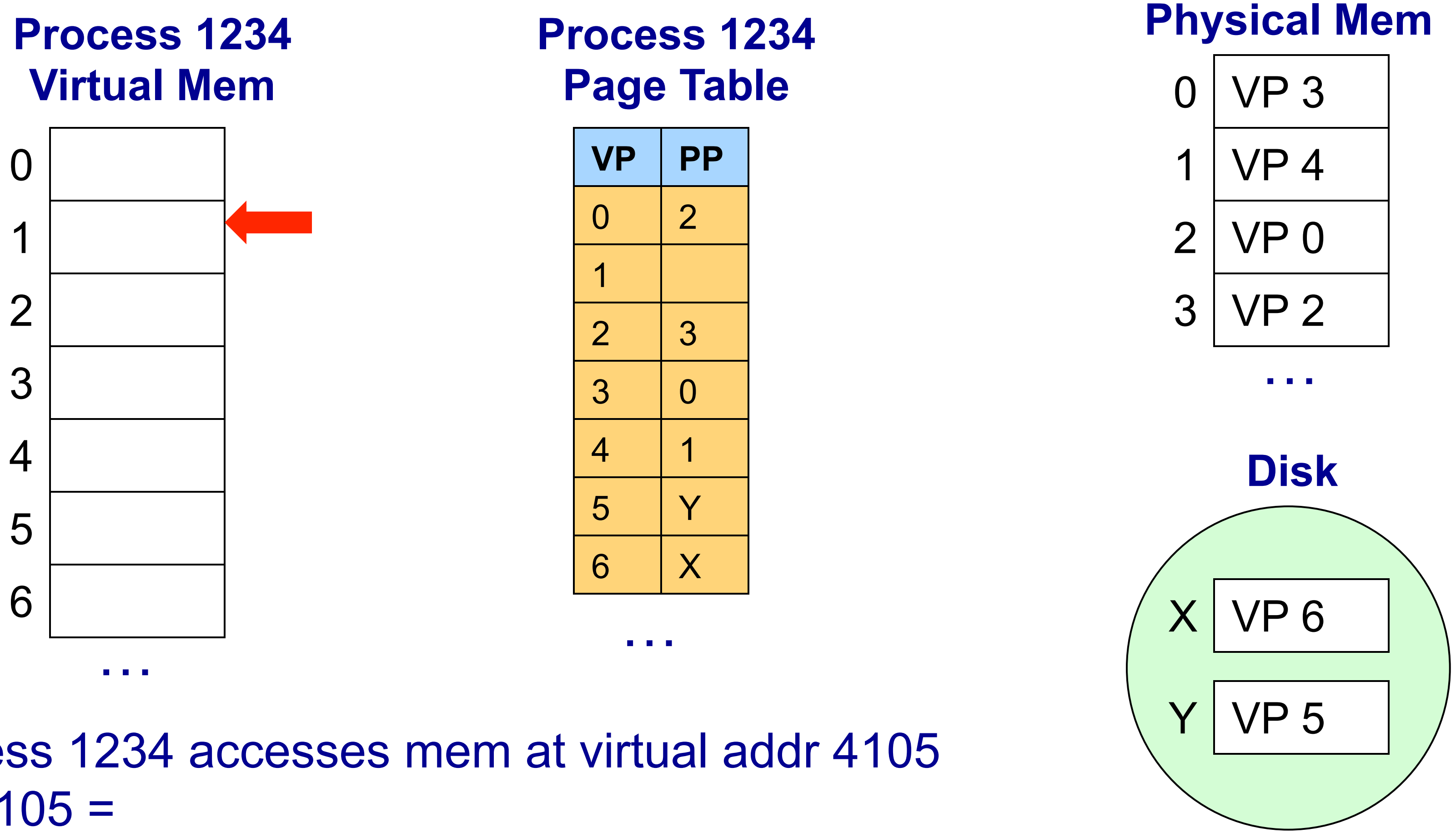
Physical page num = 3; offset = 8

= $00000000000000000011000000001000_B$

= 12296

Hardware fetches/stores data from/to phys addr 12296

Virtual Memory Example 3



VP	PP
0	2
1	
2	3
3	0
4	1
5	Y
6	X

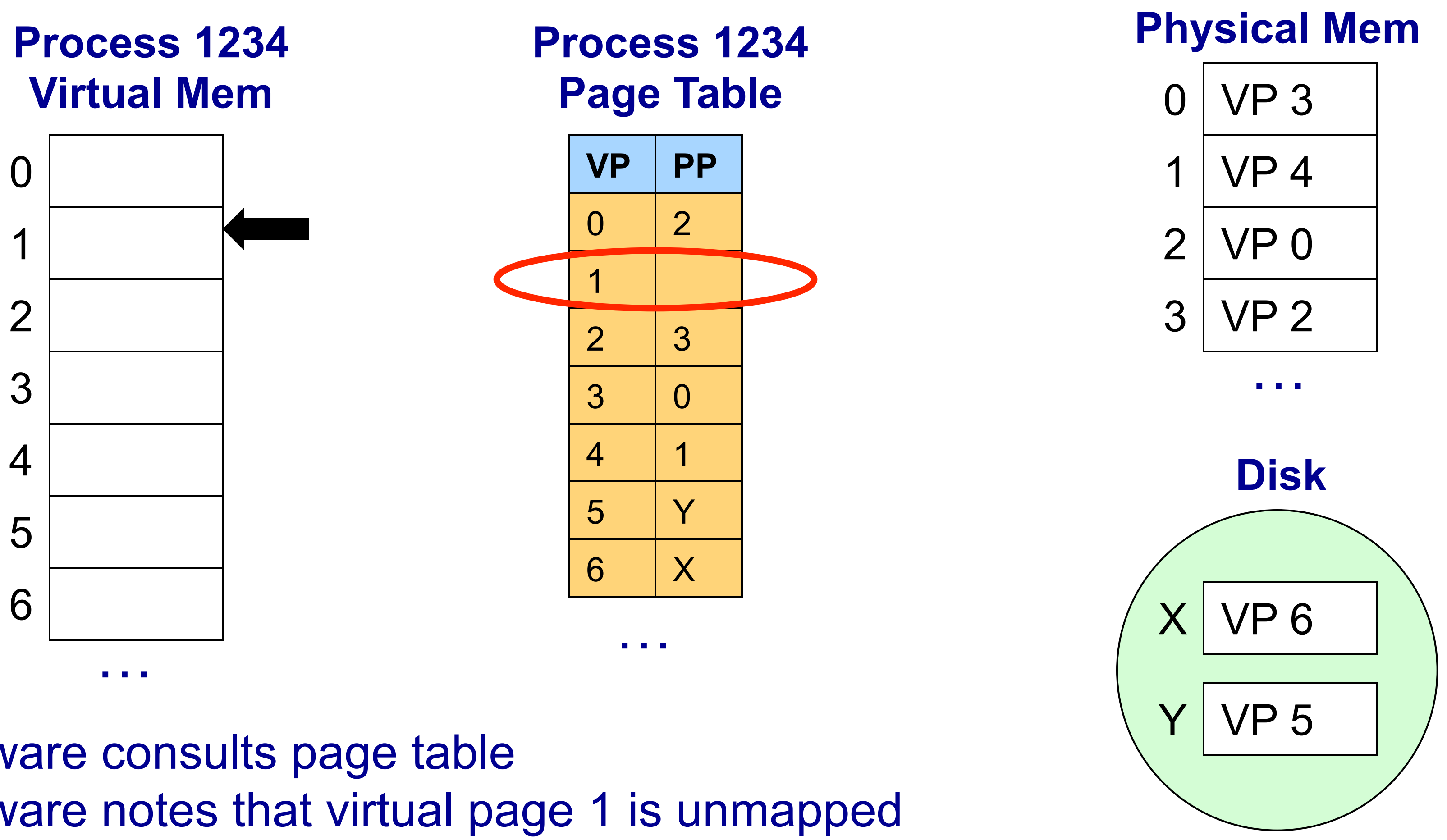
Process 1234 accesses mem at virtual addr 4105

4105 =

$00000000000000000001000000001001_B$ =

Virtual page num = 1; offset = 9

Virtual Memory Example 3 (cont.)



Hardware consults page table
Hardware notes that virtual page 1 is unmapped
Page miss!
Hardware generates **segmentation fault**
(See ***Signals*** lecture for remainder!)

Storing Page Tables

Question

- Where are the page tables themselves stored?

Answer

- In main memory

Question

- What happens if a page table is swapped out to disk???!!!

Answer

- OS is responsible for swapping
- Special logic in OS “pins” page tables to physical memory
  - So they never are swapped out to disk

Storing Page Tables (cont.)

Question

- Doesn’t that mean that each logical memory access requires **two** physical memory accesses – one to access the page table, and one to access the desired datum?

Answer

- Yes!

Question

- Isn’t that inefficient?

Answer

- Not really…

Storing Page Tables (cont.)

Note 1

- Page tables are accessed frequently
- Likely to be cached in L1/L2/L3 cache

Note 2

- IA-32 architecture provides special-purpose hardware support for virtual memory…

Translation Lookaside Buffer

Translation lookaside buffer (TLB)

- Small cache on CPU
- Each TLB entry consists of a page table entry
- Hardware first consults TLB
    - Hit => no need to consult page table in L1/L2/L3 cache or memory
    - Miss => swap relevant entry from page table in L1/L2/L3 cache or memory into TLB; try again
- See Bryant & O'Hallaron book for details

Caching again!!!

Aside: Segmentation

In the early days (before the mid-1950s)

- Programmers incorporated storage allocation in their programs
- … whenever the total information exceeded main memory

Segmentation

- Programmers would divide their programs into "segments"
- Which would "overlay" (i.e., replace) one another in main memory

Pros

- Programmers are intimately familiar with their code
- And can optimize the layout of information in main memory

Cons

- Immensely tedious and error-prone
- Compromises the portability of the code

Additional Benefits of Virtual Memory

Virtual memory concept facilitates/enables many other OS features; examples…

Context switching (as described last lecture)

- **Illusion**: To context switch from process X to process Y, OS must save contents of registers **and memory** for process X, restore contents of registers **and memory** for process Y
- **Reality**: To context switch from process X to process Y, OS must save contents of registers **and virtual memory** for process X, restore contents of registers **and virtual memory** for process Y
- **Implementation**: To context switch from process X to process Y, OS must save contents of registers **and page table** for process X, restore contents of registers **and page table** for process Y

Additional Benefits of Virtual Memory

Memory protection among processes

- Process's page table references only physical memory pages that the process currently owns
- Impossible for one process to accidentally/maliciously affect physical memory used by another process

Memory protection within processes

- Permission bits in page-table entries indicate whether page is read-only, etc.
- Allows CPU to prohibit
  - Writing to RODATA & TEXT sections
  - Access to protected (OS owned) virtual memory

Additional Benefits of Virtual Memory

Linking

- Same memory layout for each process
    - E.g., TEXT section always starts at virtual addr `0x08048000`
    - E.g., STACK always grows from virtual addr `0x0bffffff` to lower addresses
- Linker is independent of physical location of code

Code and data sharing

- User processes can share some code and data
    - E.g., single physical copy of stdio library code (e.g. printf)
- Mapped into the virtual address space of each process

Additional Benefits of Virtual Memory

Dynamic memory allocation

- User processes can request additional memory from the heap
  - E.g., using `malloc()` to allocate, and `free()` to deallocate
- OS allocates *contiguous* virtual memory pages…
  - … and scatters them *anywhere* in physical memory

Additional Benefits of Virtual Memory

Creating new processes

- Easy for "parent" process to "fork" a new "child" process
  - Initially: make new PCB containing copy of parent page table
  - Incrementally: change child page table entries as required
- See ***Process Management*** lecture for details
  - **`fork()`** system-level function

Overwriting one program with another

- Easy for a process to replace its program with another program
  - Initially: set page table entries to point to program pages that already exist on disk!
  - Incrementally: swap pages into memory as required
- See ***Process Management*** lecture for details
  - **`execvp()`** system-level function

Measuring Memory Usage

On nobel computers:

```
$ ps l
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0 42579 13082 13081  20   0 112712  2016 wait   Ss   pts/0      0:00 -bash
0 42579 13305 13082  20   0 156916 13684 signal T    pts/0      0:00 emacs -nw
0 42579 13517 13082  20   0  11272   892 -      R+   pts/0      0:00 ps l
```

VSZ (virtual memory size): virtual memory usage
RSS (resident set size): physical memory usage

Summary

Locality and caching

- Spatial & temporal locality
- Good locality => caching is effective

Typical storage hierarchy

- Registers, L1/L2/L3 cache, main memory, local secondary storage (esp. disk), remote secondary storage

Virtual memory

- Illusion vs. reality
- Implementation
  - Virtual addresses, page tables, translation lookaside buffer (TLB)
- Additional benefits (many!)

Virtual memory concept permeates the design of modern operating systems and computer hardware