



The Memory/Storage Hierarchy and Virtual Memory

Goals of this Lecture



Help you learn about:

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

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

Recall this iClicker Question?



Q: What effect does virtual memory have on the speed and security of processes?

- | | Speed | Security |
|----|-------|-----------|
| A. | | |
| B. | | |
| C. | | no change |
| D. | | |
| E. | | |

Let's see how to make the answer:

Speed	Security
no change	

Agenda



Locality and caching

Typical storage hierarchy

Virtual memory



Storage Device Speed vs. Size

Facts:

- **CPU** needs sub-nanosecond access to data to run instructions at full speed
- **Fast** storage (sub-nanosecond) is small (100-1000 bytes)
- **Big** storage (gigabytes) is slow (15 nanoseconds)
- **Huge** storage (terabytes) is *glacially* slow (milliseconds)

Goal:

- Need many gigabytes of memory,
- but with fast (sub-nanosecond) average access time

Solution: **locality** allows **caching**

- Most programs exhibit good **locality**
- A program that exhibits good locality will benefit from proper **caching**, which enables good **average** performance

Locality



Two kinds of **locality**

- **Temporal** locality
 - If a program references item X now, it probably will reference X again soon
- **Spatial** locality
 - If a program references item X now, it probably will reference item at address $X \pm 1$ 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

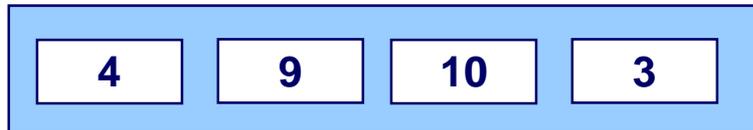
Good locality + proper caching

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

Caching in a Storage Hierarchy



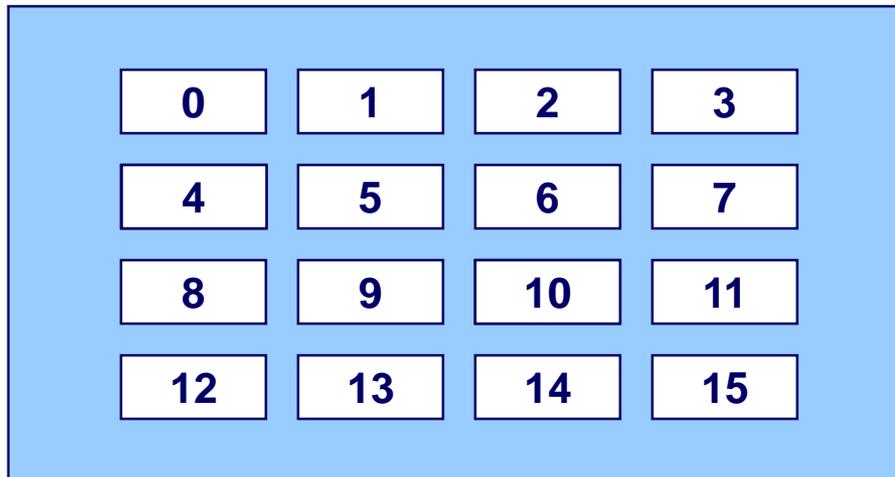
Level k:



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

Blocks copied between levels

Level k+1:



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



Cache Hits and Misses

Cache hit

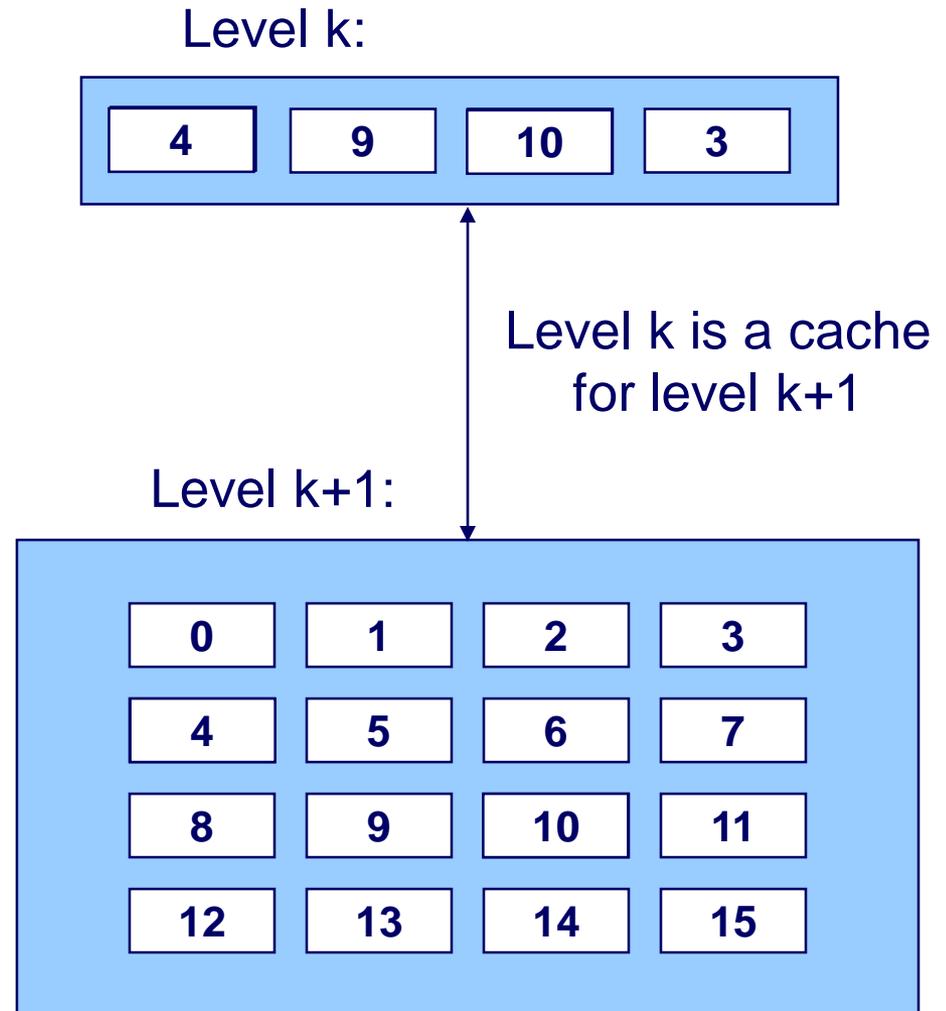
- E.g., request for block 10
- Access block 10 at level k
- Fast!

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: “oracle”

- 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
- (can be) 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

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

col-major

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

C uses **row-major** order

- Access in row-major order \Rightarrow good spatial locality
- Access in column-major order \Rightarrow poor spatial locality

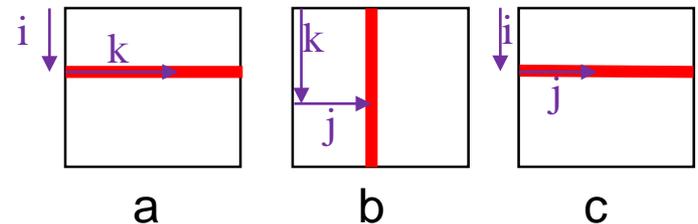
Locality/Caching Example: Matrix Mult



```
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



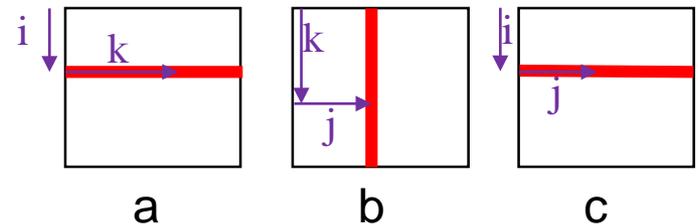
Locality/Caching Example: Matrix Mult



```
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



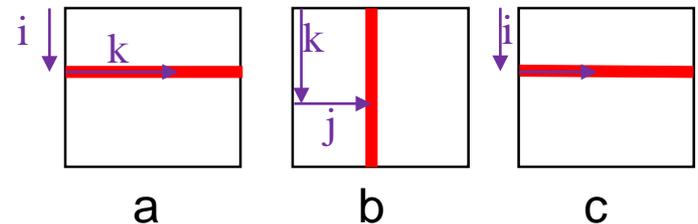
Locality/Caching Example: Matrix Mult



```
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

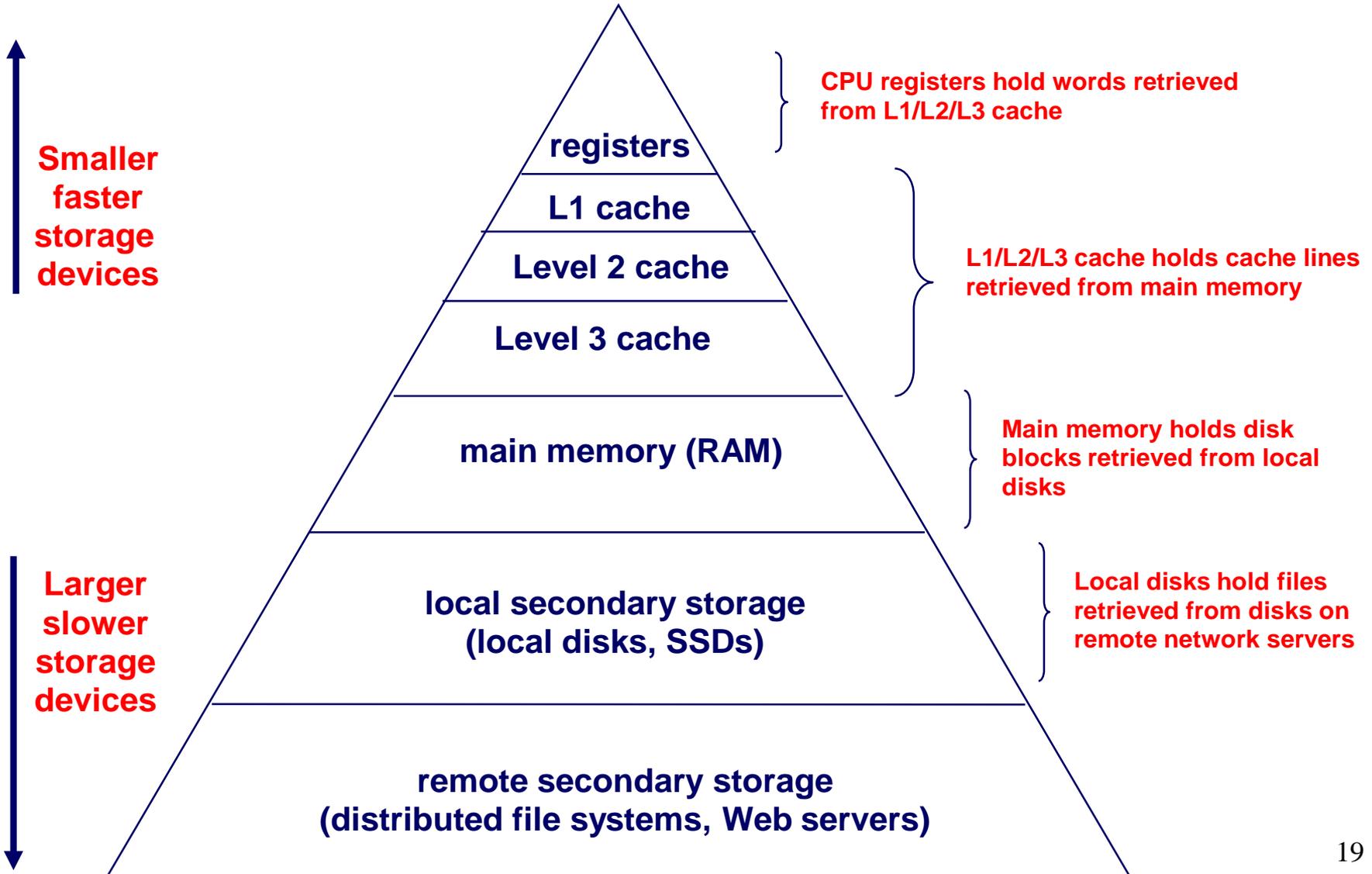


Locality and caching

Typical storage hierarchy

Virtual memory

Typical Storage Hierarchy



Typical Storage Hierarchy



Factors to consider:

- Capacity
- Latency (how long to do a read)
- Bandwidth (how many bytes/sec can be read)
 - Weakly correlated to latency: reading 1 MB from a hard disk isn't much slower than reading 1 byte
- Volatility
 - Do data persist in the absence of power?

Typical Storage Hierarchy



Registers

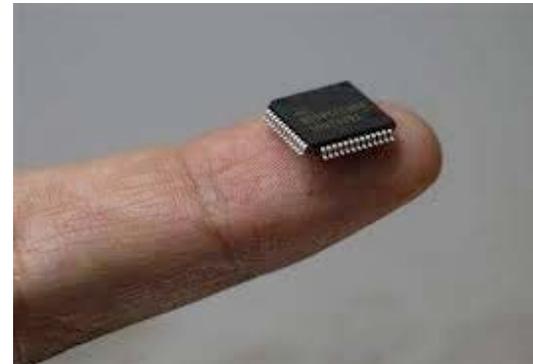
- **Latency:** 0 cycles
- **Capacity:** 8-256 registers (31 general purpose registers in AArch64)

L1/L2/L3 Cache

- **Latency:** 1 to 40 cycles
- **Capacity:** 32KB to 32MB

Main memory (RAM)

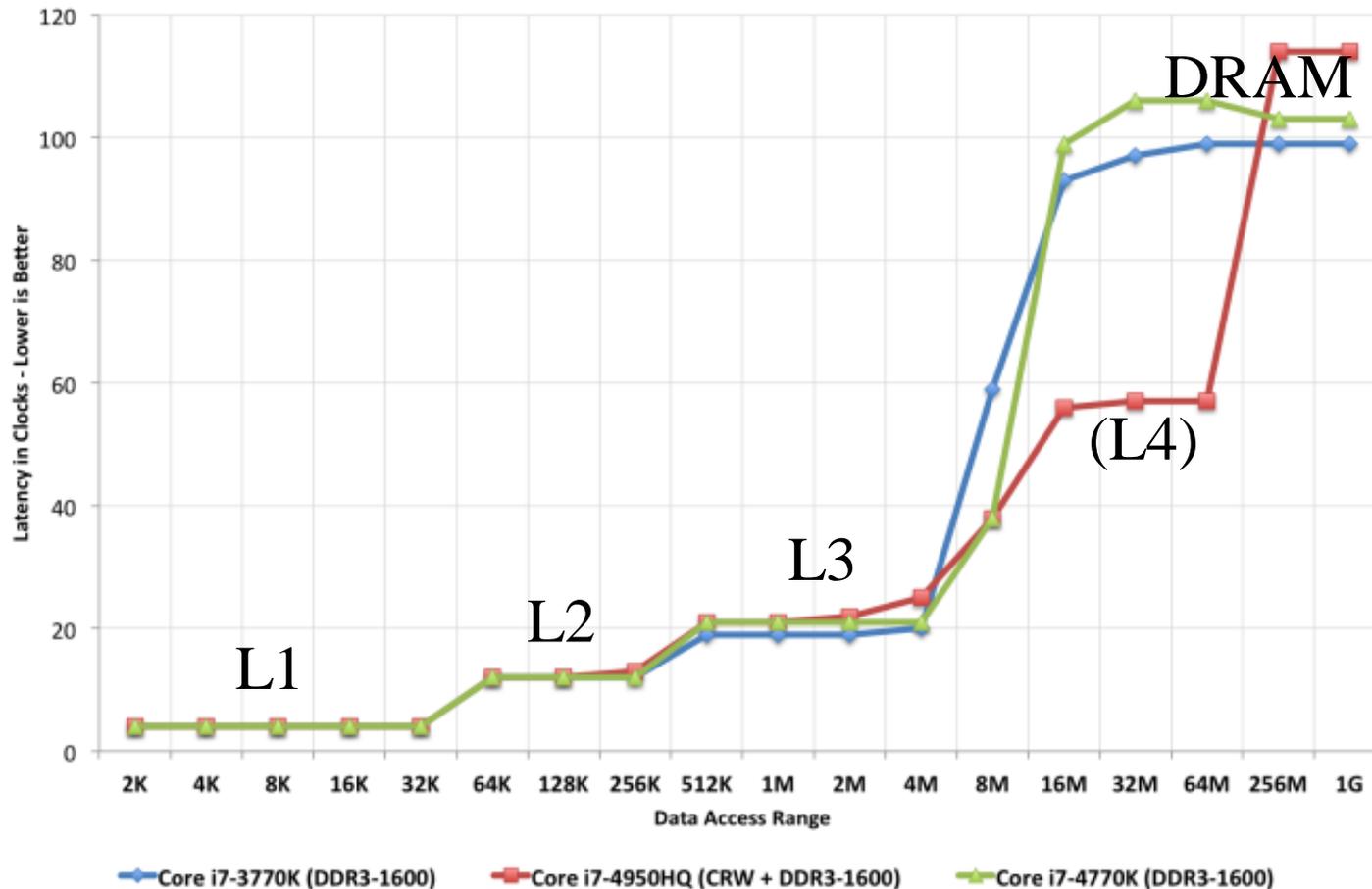
- **Latency:** ~ 50-100 cycles
 - 100 times slower than registers
- **Capacity:** GB



Cache / RAM Latency



Memory Latency vs. Access Range (Sandra 2013 SP3)



1 clock = $3 \cdot 10^{-10}$ sec

Typical Storage Hierarchy



Local secondary storage: disk drives

- Solid-State Disk (SSD):
 - Flash memory (nonvolatile)
 - **Latency:** 0.1 ms (~ 300k cycles)
 - **Capacity:** 128 GB – 2 TB
- Hard Disk:
 - Spinning magnetic platters, moving heads
 - **Latency:** 10 ms (~ 30M cycles)
 - **Capacity:** 1 – 10 TB



ComputerHope.com



Disks



HDD



1 ms

SSD

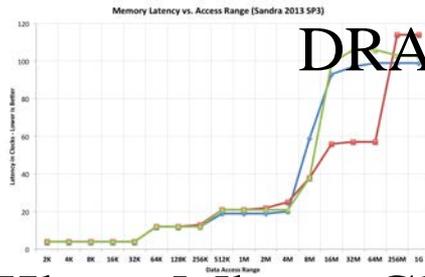


ComputerHope.com

1 μ s

DRAM

1 ns



Kb

Mb

Gb

Tb

Typical Storage Hierarchy



Remote secondary storage (a.k.a. “the cloud”)

- **Latency:** tens of milliseconds
 - Limited by network bandwidth
- **Capacity:** essentially unlimited



Storage Hierarchy & Caching Issues



Issue: Block size?

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

Device	Block Size
Register	8 bytes
L1/L2/L3 cache line	128 bytes
Main memory page	4KB or 64KB
Disk block	512 bytes to 4KB
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 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



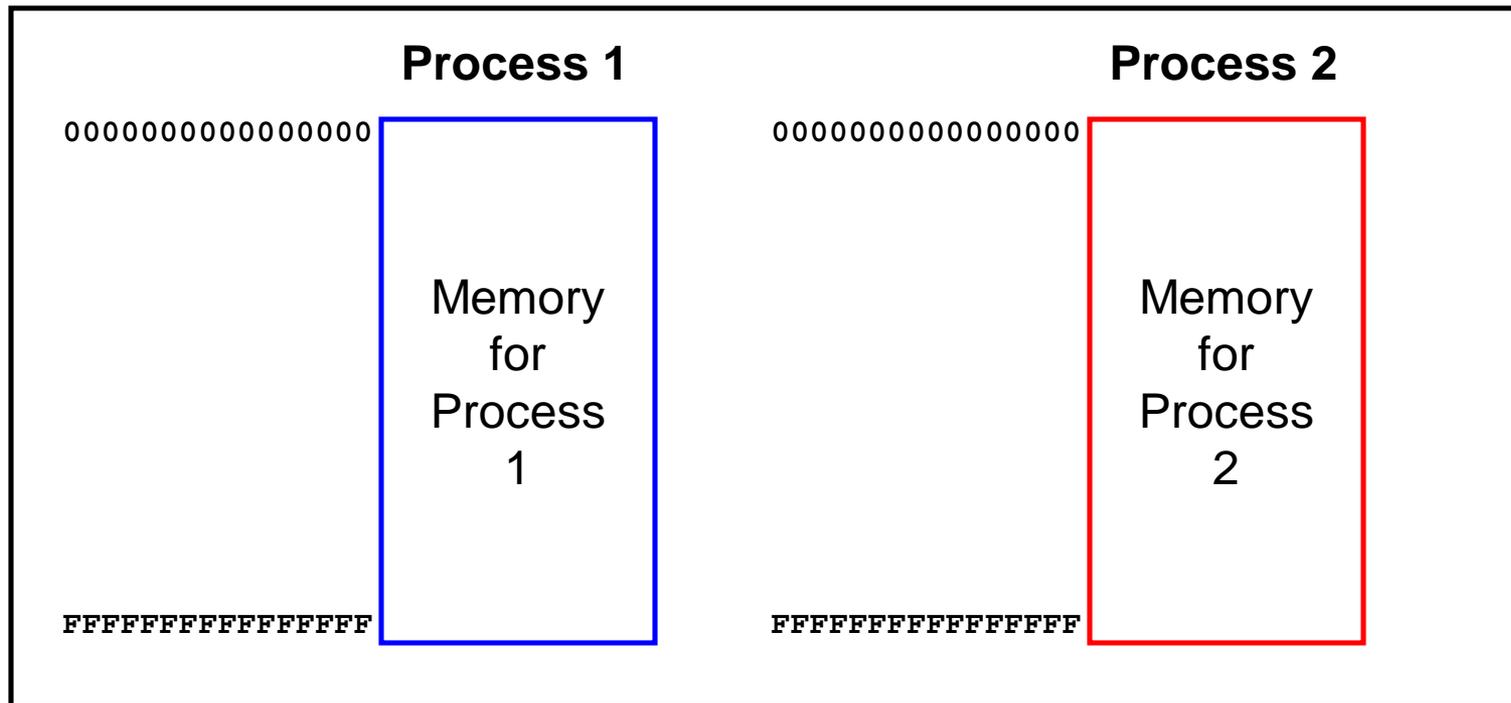
Locality and caching

Typical storage hierarchy

Virtual memory



Main Memory: Illusion

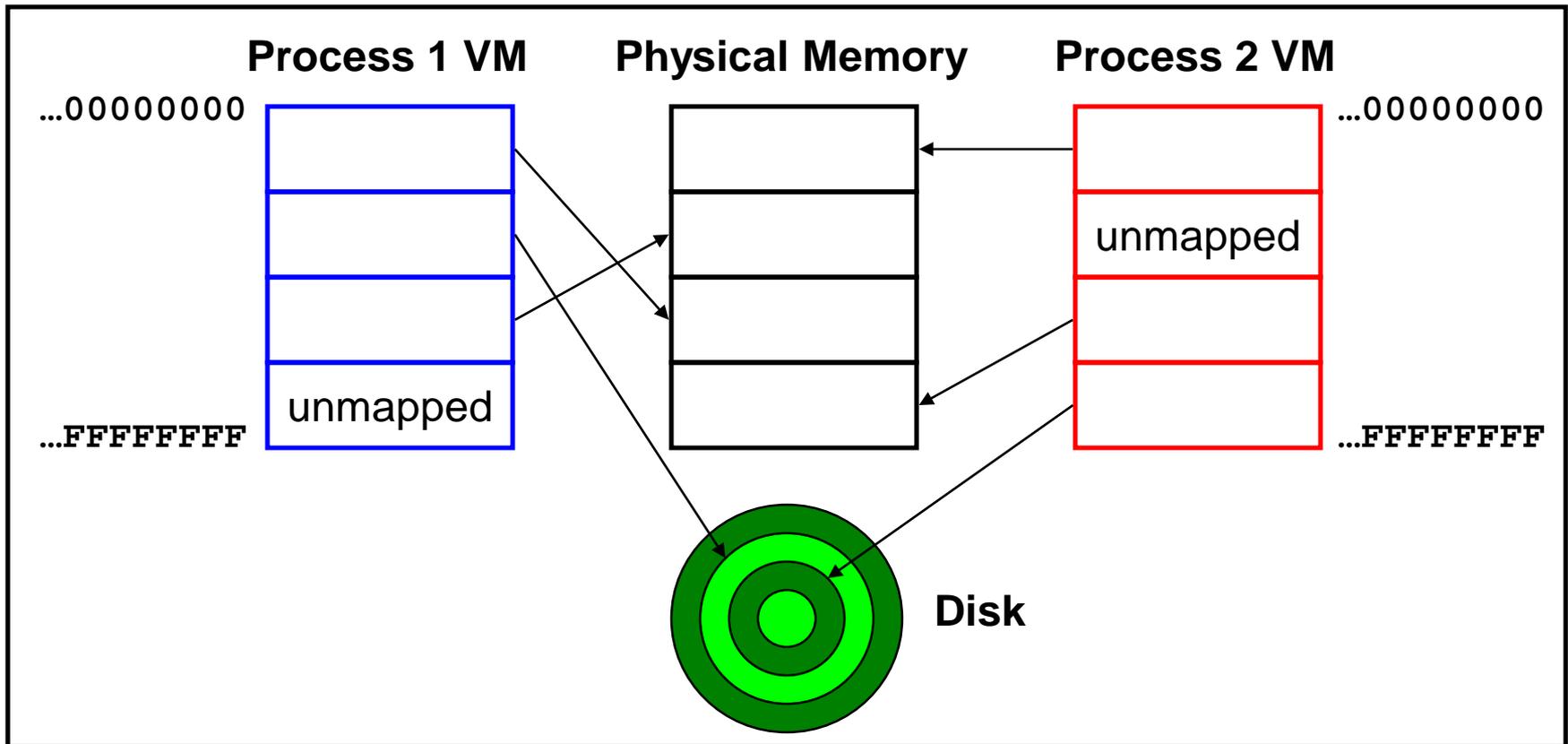


Each process sees main memory as

Huge: $2^{64} = 16 \text{ EB}$ (16 exabytes) of memory $\approx 10^{19}$

Uniform: contiguous memory locations from 0 to $2^{64}-1$

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

virtual page num	offset
------------------	--------

- 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

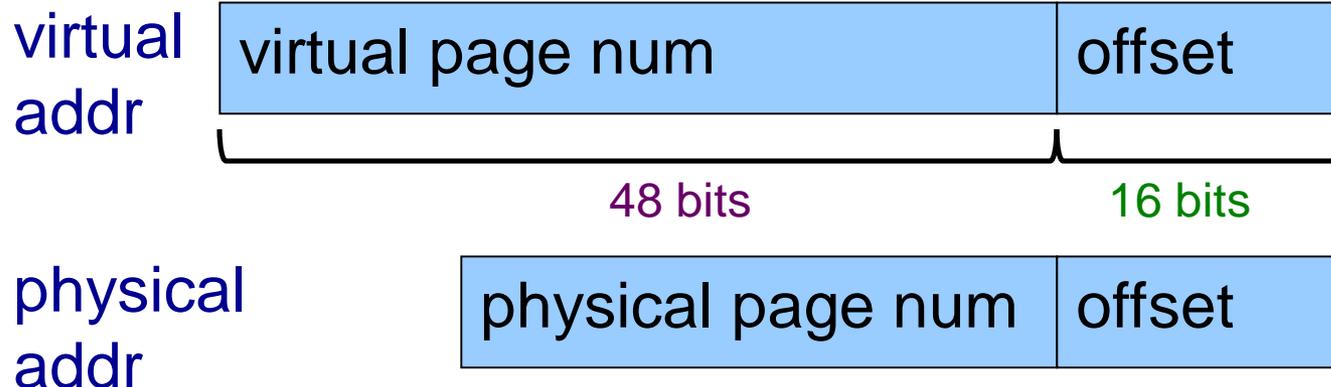
physical page num	offset
-------------------	--------

- 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

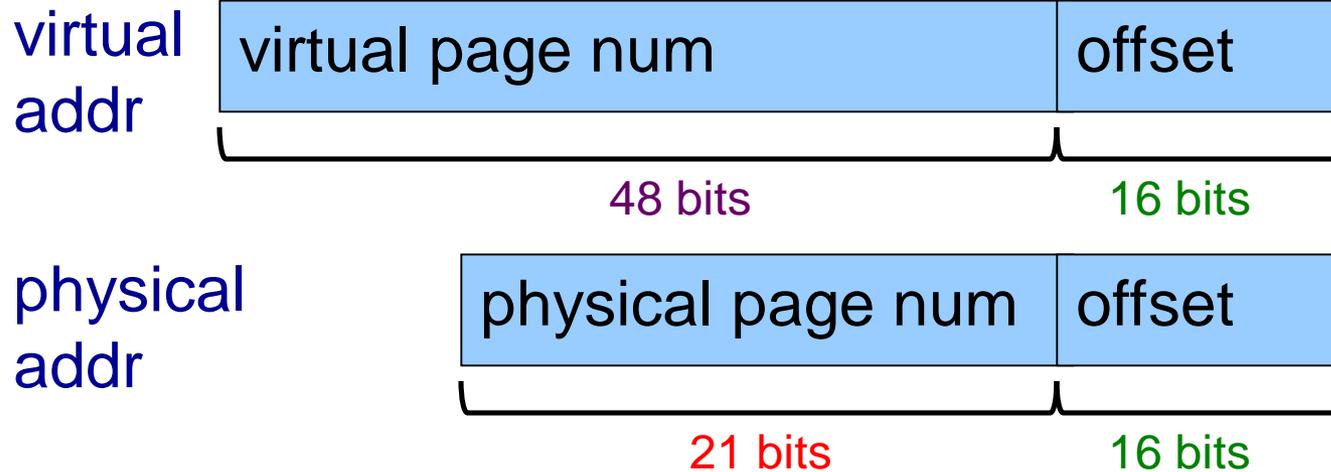
CourseLab Virtual & Physical Addresses



On AArch64:

- Each virtual address consists of 64 bits
 - There are 2^{64} bytes of virtual memory (per process)
- Each offset is either 12 or 16 bits (determined by OS) – 16 bits on armlab
 - Each page consists of 2^{16} bytes
- Each virtual page number consists of $64 - 16 = 48$ bits
 - There are 2^{48} virtual pages

CourseLab Virtual & Physical Addresses



On ArmLab:

- Each physical address consists of 37 bits
 - There are 2^{37} (128G) bytes of physical memory (per computer)
- With 64K pages, each offset is 16 bits
 - Each page consists of 2^{16} bytes
- Each physical page number consists of $37 - 16 = 21$ bits
 - There are 2^{21} physical pages

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**

0	
1	
2	
3	
4	
5	
6	

...

**Process 1234
Page Table**

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

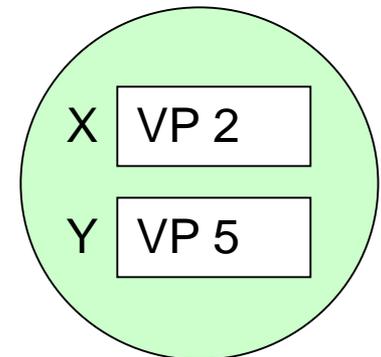
...

Physical Mem

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

...

Disk



Process 1234 accesses mem at
virtual addr 262146 (= 0x40002)

 **iClicker Question coming up . . .**

iClicker Question

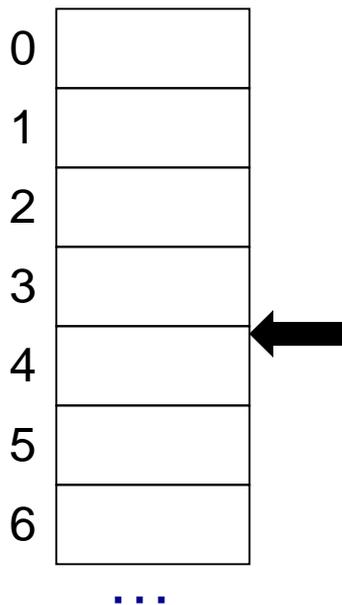
Q: For virtual address 262146 (= 0x40002), what is the virtual page number and offset within that page?

- A. Page = 4, offset = 2
- B. Page = 0x40 = 64, offset = 2
- C. Page = 0x400 = 1024, offset = 2
- D. Page = 2, offset = 4
- E. Page = 2, offset = 0x400 = 1024

Virtual Memory Example 1 (cont.)



Process 1234
Virtual Mem



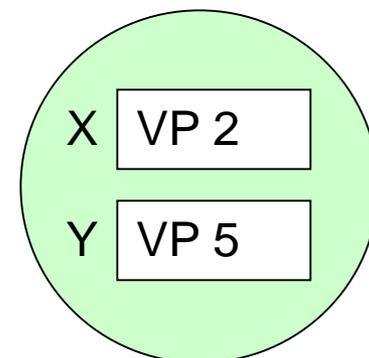
Process 1234
Page Table

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
...	

Disk



Hardware consults page table

Hardware notes that virtual page 4 maps to phys page 1

Page hit!

▶ iClicker Question

Q: For virtual address 262146 (= 0x40002),
what is the corresponding physical address?

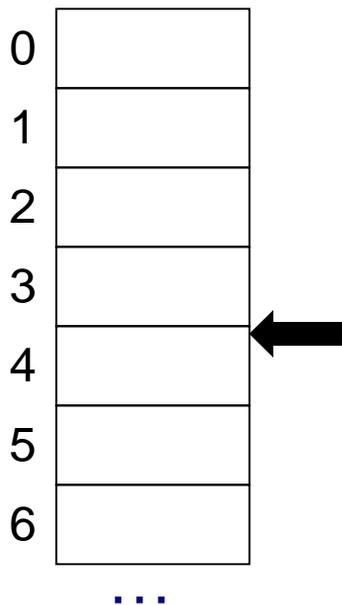
- A. 0x140002
- B. 0x41002
- C. 0x10002
- D. 0x10000
- E. 0x2

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

Virtual Memory Example 1 (cont.)



Process 1234
Virtual Mem



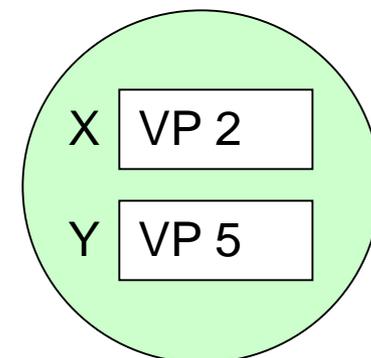
Process 1234
Page Table

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
...	

Disk



Hardware forms physical addr

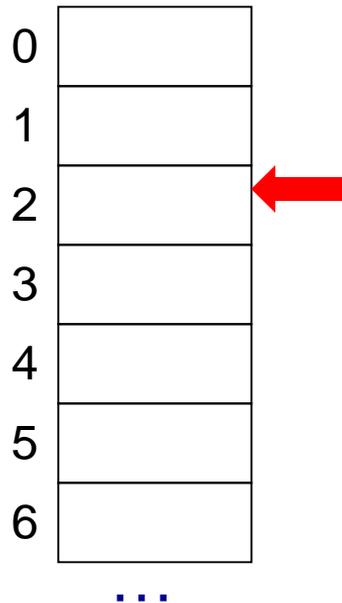
Physical page num = 1; offset = 2
= 0x10002
= 65538

Hardware fetches/stores data from/to phys addr 65538



Virtual Memory Example 2

Process 1234
Virtual Mem



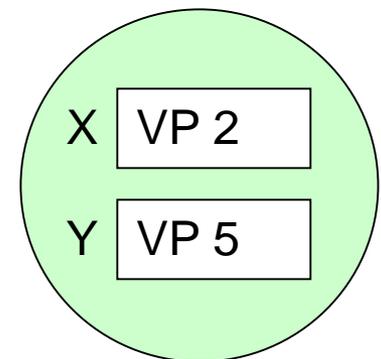
Process 1234
Page Table

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
...	

Disk



Process 1234 accesses mem at virtual addr 131080

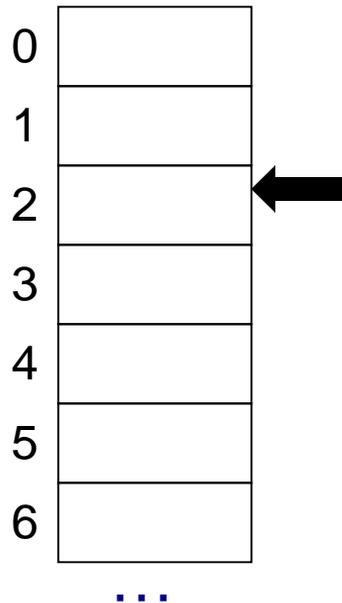
$$131080 = 0x20008 =$$

Virtual page num = 2; offset = 8

Virtual Memory Example 2 (cont.)



Process 1234
Virtual Mem



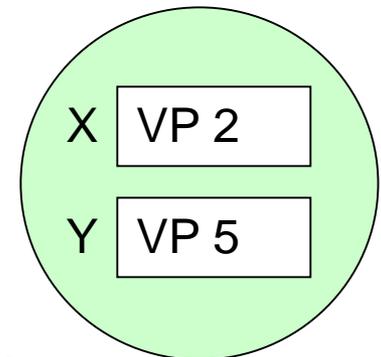
Process 1234
Page Table

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
...	

Disk



Hardware consults page table

Hardware notes that virtual page 2 maps to spot X on disk

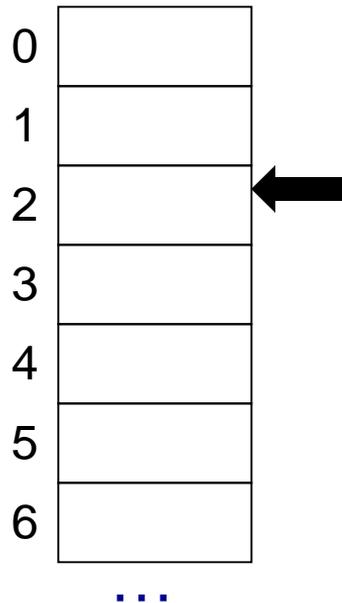
Page miss!

Hardware generates **page fault**

Virtual Memory Example 2 (cont.)



Process 1234
Virtual Mem



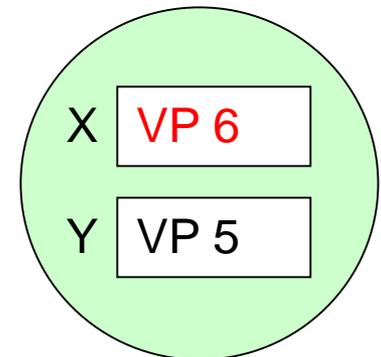
Process 1234
Page Table

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



OS gains control of CPU

OS swaps virtual pages 6 and 2

This takes a long while (disk latency); run another process for the time being, then eventually...

OS updates page table accordingly

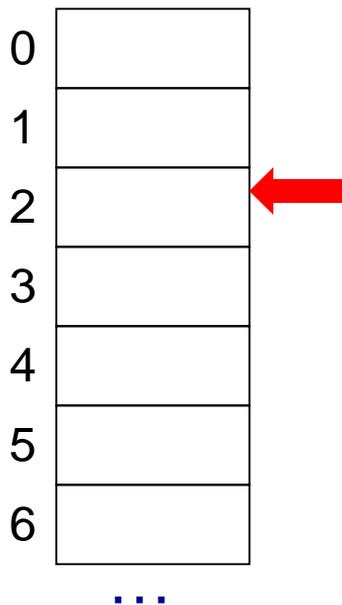
Control returns to process 1234

Process 1234 re-executes **same instruction**

Virtual Memory Example 2 (cont.)



Process 1234
Virtual Mem



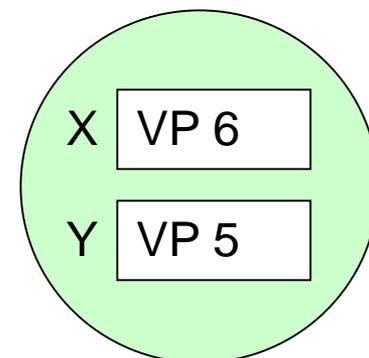
Process 1234
Page Table

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



Process 1234 accesses mem at virtual addr 131080

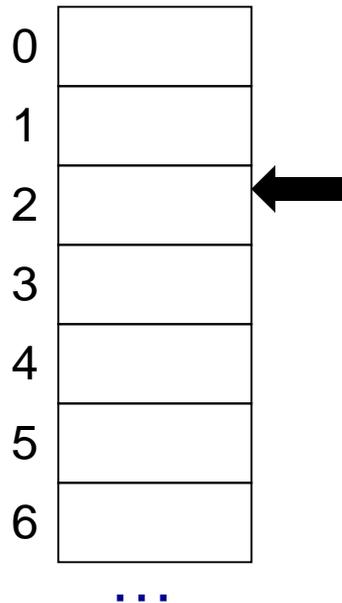
$$131080 = 0x20008 =$$

Virtual page num = 2; offset = 8

Virtual Memory Example 2 (cont.)



Process 1234
Virtual Mem



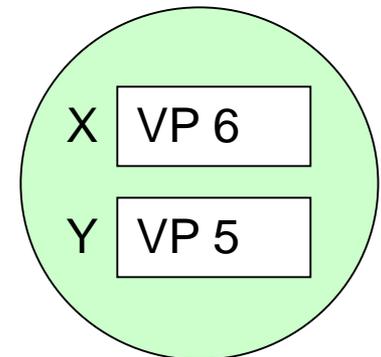
Process 1234
Page Table

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



Hardware consults page table

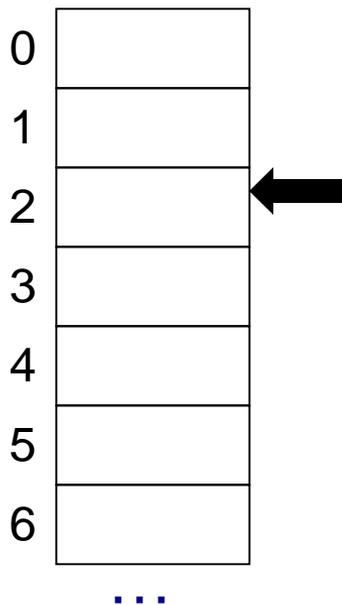
Hardware notes that virtual page 2 maps to phys page 3

Page hit!

Virtual Memory Example 2 (cont.)



Process 1234
Virtual Mem



Process 1234
Page Table

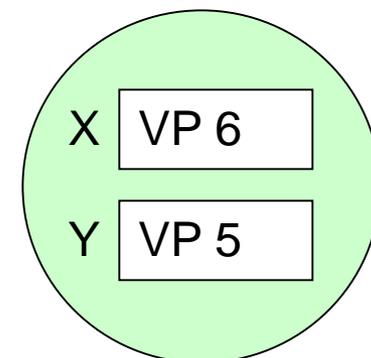
VP	PP
0	2
1	
2	3
3	0
4	1
5	Y
6	X
...	

Physical Mem

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

...

Disk



Hardware forms physical addr

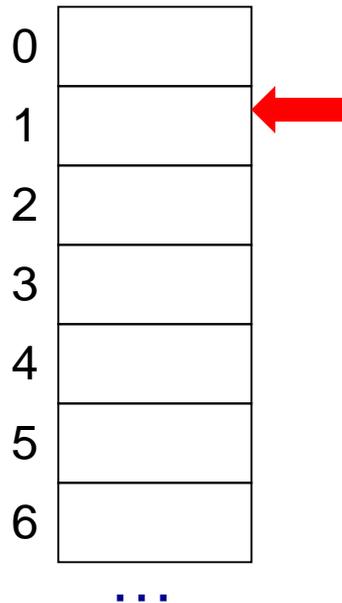
Physical page num = 3; offset = 8
= 0x30008
= 196622

Hardware fetches/stores data from/to phys addr 196622



Virtual Memory Example 3

Process 1234
Virtual Mem



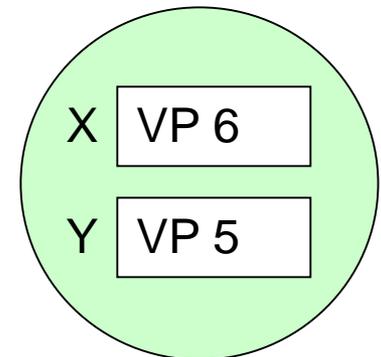
Process 1234
Page Table

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



Process 1234 accesses mem at virtual addr 65545

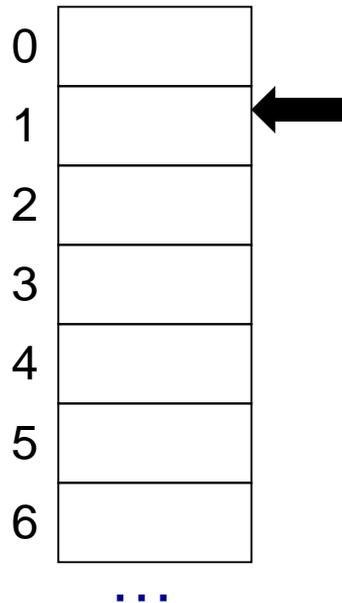
$$65545 = 0x10009 =$$

Virtual page num = 1; offset = 9

Virtual Memory Example 3 (cont.)



Process 1234
Virtual Mem



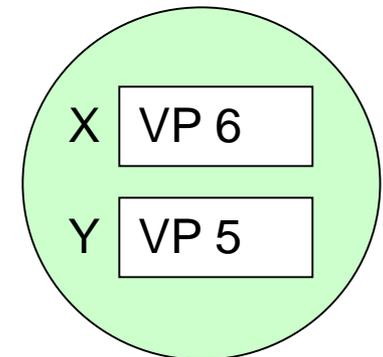
Process 1234
Page Table

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



Hardware consults page table

Hardware notes that virtual page 1 is unmapped

Page miss!

Hardware generates **segmentation fault** (*Signals* lecture!)

OS gains control, (probably) kills process



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

- It hurts! So don't do that, then!
- 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

- Conceptually, 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

- Modern hardware (including ARM) 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 \Rightarrow no need to consult page table in L1/L2/L3 cache or memory
 - Miss \Rightarrow 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!!!

Recall this iClicker Question?



Q: What effect does virtual memory have on the speed and security of processes?

- | | Speed | Security |
|----|-------|-----------|
| A. | | |
| B. | | |
| C. | | no change |
| D. | | |
| E. | | |

That's why the real answer is:

Speed	Security
no change	

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 pointer to the page table** for process X, restore contents of registers **and pointer to the 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 `0x400000`
- 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

```
$ ps l
F  UID  PID  PPID  PRI  NI  VSZ  RSS  WCHAN  STAT  TTY  TIME  COMMAND
0 42579 9655 9696 30 10 167568 13840 signal TN pts/1 0:00 emacs -nw
0 42579 9696 9695 30 10 24028 2072 wait SNs pts/1 0:00 -bash
0 42579 9725 9696 30 10 11268 956 - RN+ pts/1 0:00 ps l
```

VSZ (virtual memory size): virtual memory usage
RSS (resident set size): physical memory usage
(both measured in kilobytes)

Summary



Locality and caching

- Spatial & temporal locality
- Good locality \Rightarrow 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 operating systems and computer hardware