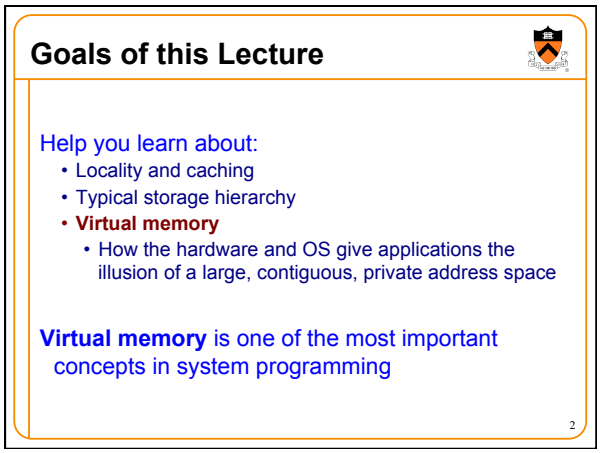
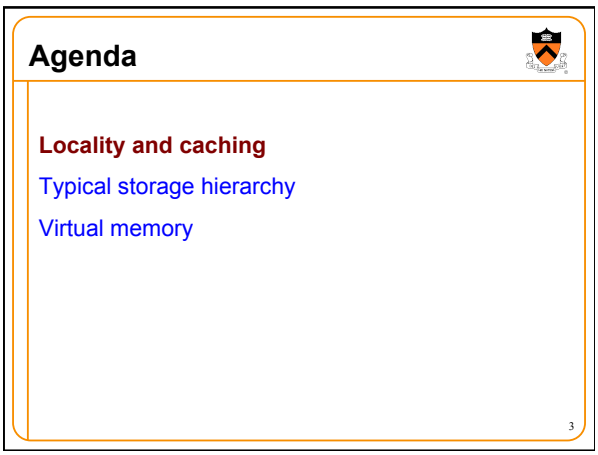
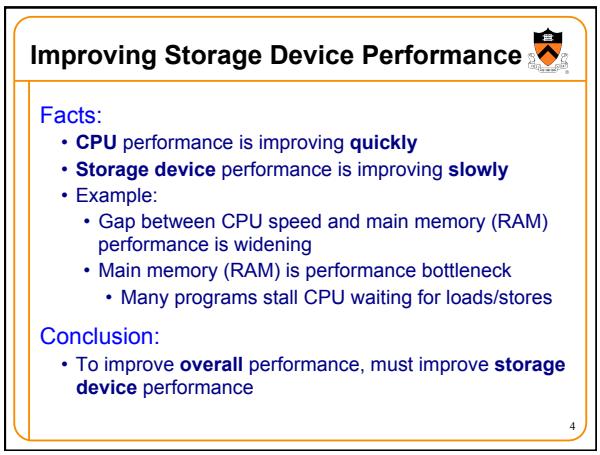
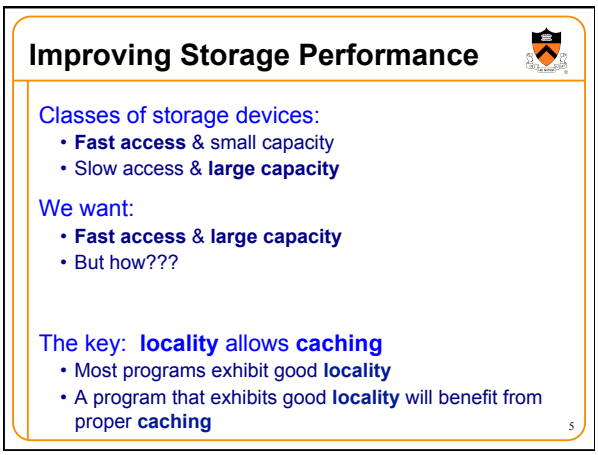
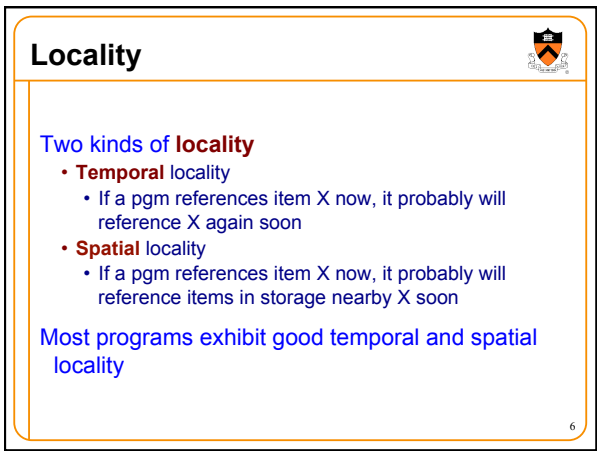

Storage Management
Jennifer Rexford
1


Goals of this Lecture
Help you learn about:
• Locality and caching
• Typical storage hierarchy
• Virtual memory
• How the hardware and OS give applications the illusion of a large, contiguous, private address space
Virtual memory is one of the most important concepts in system programming
2


Agenda
Locality and caching
Typical storage hierarchy
Virtual memory
3


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 CPU waiting for loads/stores
Conclusion:
• To improve overall performance, must improve storage device performance
4


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
5


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
6

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

7

Caching

Cache

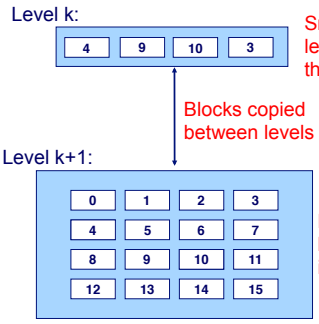
- Fast access, small capacity storage device
- Acts as a staging area for a subset of the items in a slow access, large capacity storage device

Good locality + proper caching

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

8

Caching in a Storage Hierarchy



9

Cache Hits and Misses

Cache hit

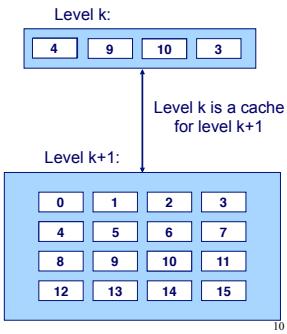
- E.g., request for block 10
- Access block 10 at level k
- Fast!

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



10

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!

11

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

12

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**)

13

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

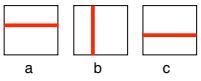
C uses **row-major** order
- Access in row-major order => good spatial locality
- Access in column-major order => poor spatial locality

14

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

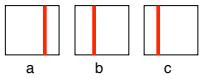


15

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

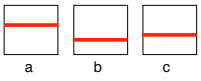


16

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



17

Agenda

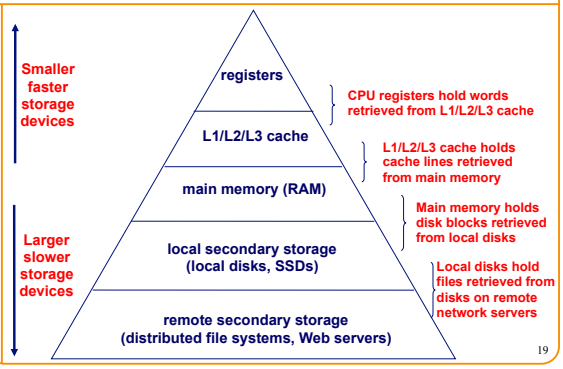
Locality and caching

Typical storage hierarchy

Virtual memory

18

Typical Storage Hierarchy



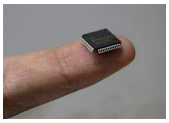
19

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



20

Typical Storage Hierarchy

Local secondary storage: disk drives
- **Latency**: ~100,000 cycles
  - 1000 times slower than main memory
  - 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

21

Typical Storage Hierarchy

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

22

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

23

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

24

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	64 bytes
Main memory page	4KB (4096 bytes)
Disk block	4KB (4096 bytes)
Disk transfer block	4KB (4096 bytes) to 64MB (67108864 bytes)

25

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

26

Agenda

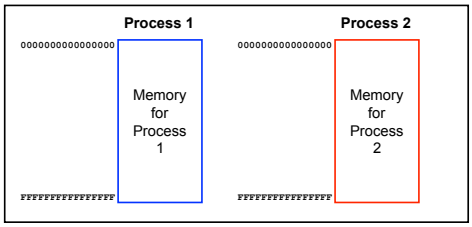
Locality and caching

Typical storage hierarchy

Virtual memory

27

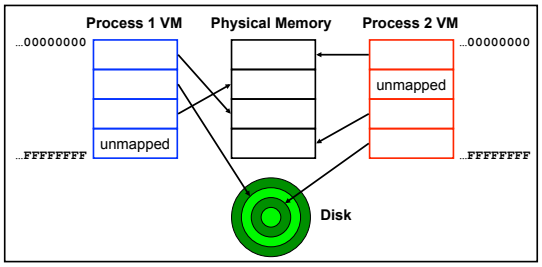
Main Memory: Illusion



Each process sees main memory as
Huge: 2^{64} = 16 EB (16 exabytes) of memory
Uniform: contiguous memory locations from 0 to 2^{64}-1

28

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

29

Virtual & Physical Addresses

Question

- How do OS and hardware implement virtual memory?

Answer (part 1)

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

30

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

FC010 Virtual & Physical Addresses

virtual addr | virtual page num | offset |

52 bits 12 bits

physical addr | physical page num | offset |

On FC010:

- Each offset is 12 bits
  - Each page consists of 2^{12} bytes
- Each virtual page number consists of 52 bits
  - There are 2^{52} virtual pages
- Each virtual address consists of 64 bits
  - There are 2^{64} bytes of virtual memory (per process)

FC010 Virtual & Physical Addresses

virtual addr | virtual page num | offset |

52 bits 12 bits

physical addr | physical page num | offset |

22 bits 12 bits

On FC010:

- Each offset is 12 bits
  - Each page consists of 2^{12} bytes
- Each physical page number consists of 22 bits
  - There are 2^{22} physical pages
- Each physical address consists of 34 bits
  - There are 2^{34} (16G) bytes of physical memory (per computer)

Page Tables

Question

- How do OS and hardware implement virtual memory?

Answer (part 2)

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

Page Tables (cont.)

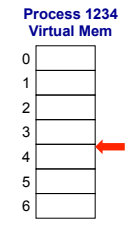
Page Table for Process 1234

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



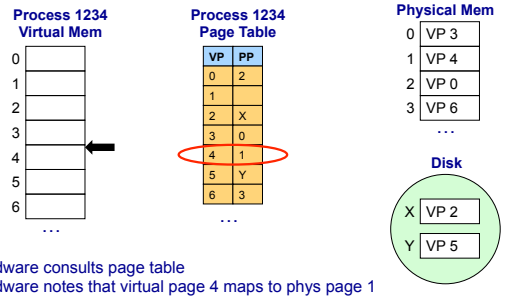
Process 1234 accesses mem at virtual addr 16386

16386 =

…00000000000000000100000000000010$_B$ =

Virtual page num = 4; offset = 2

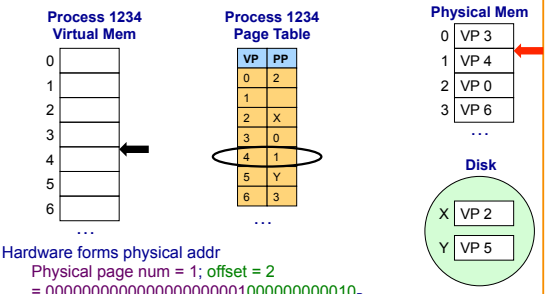
Virtual Memory Example 1 (cont.)



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

37

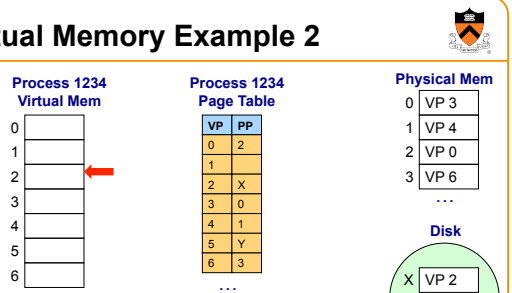
Virtual Memory Example 1 (cont.)



Hardware forms physical addr
Physical page num = 1; offset = 2
= 00000000000000000000010000000000$10_B$
= 4098
Hardware fetches/stores data from/to phys addr 4098

38

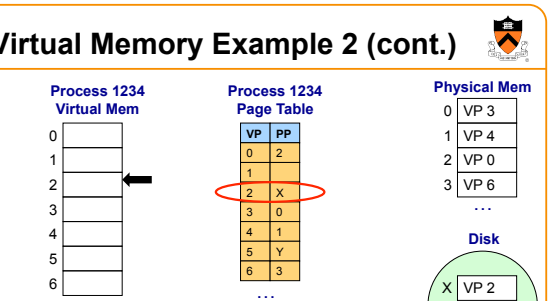
Virtual Memory Example 2



Process 1234 accesses mem at virtual addr 8200
8200 =
…00000000000000000010000000001000$_B$ =
Virtual page num = 2; offset = 8

39

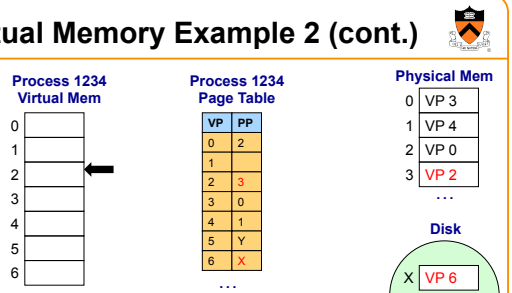
Virtual Memory Example 2 (cont.)



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

40

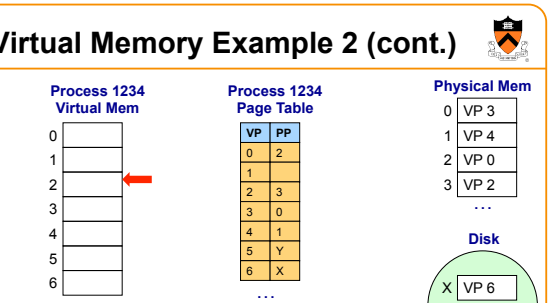
Virtual Memory Example 2 (cont.)



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**

41

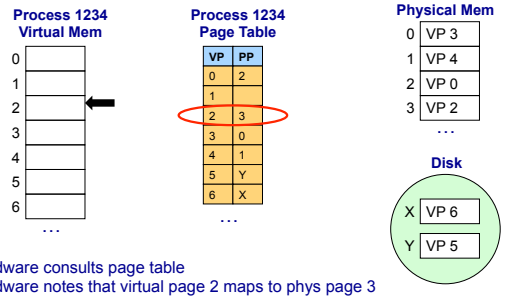
Virtual Memory Example 2 (cont.)



Process 1234 accesses mem at virtual addr 8200
8200 =
…00000000000000000010000000001000$_B$ =
Virtual page num = 2; offset = 8

42

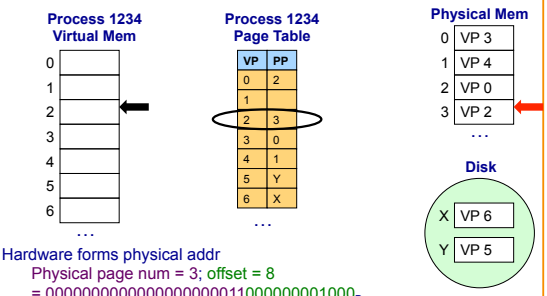
Virtual Memory Example 2 (cont.)



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

43

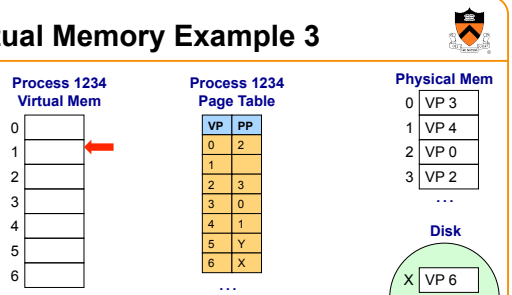
Virtual Memory Example 2 (cont.)



Hardware forms physical addr
Physical page num = 3; offset = 8
= 00000000000000000011000000001000$_B$
= 12296
Hardware fetches/stores data from/to phys addr 12296

44

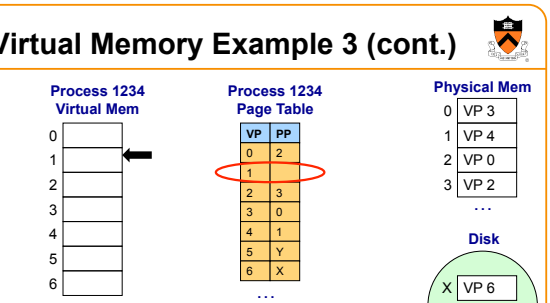
Virtual Memory Example 3



Process 1234 accesses mem at virtual addr 4105
4105 =
…00000000000000000001000000001001$_B$ =
Virtual page num = 1; offset = 9

45

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!)

46

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

47

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…

48

Storing Page Tables (cont.)

Note 1
- Page tables are accessed frequently
- Likely to be cached in L1/L2/L3 cache

Note 2
- X86-64 architecture provides special-purpose hardware support for virtual memory…

49

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!!!

50

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

51

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

52

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

53

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 `0xbfffffff` 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

54

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

55

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

56

Measuring Memory Usage

On FC010 computers:

```
$ ps l
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0 42579  7694 20201  20   0 168040 14328 signal T    pts/0      0:00 emacs -nw
0 42579  7695 20201  20   0  11260   928 -      R+   pts/0      0:00 ps l
0 42579 20201 20200  20   0  24032  2036 wait   Ss   pts/0      0:00 -bash
```

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

57

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

58