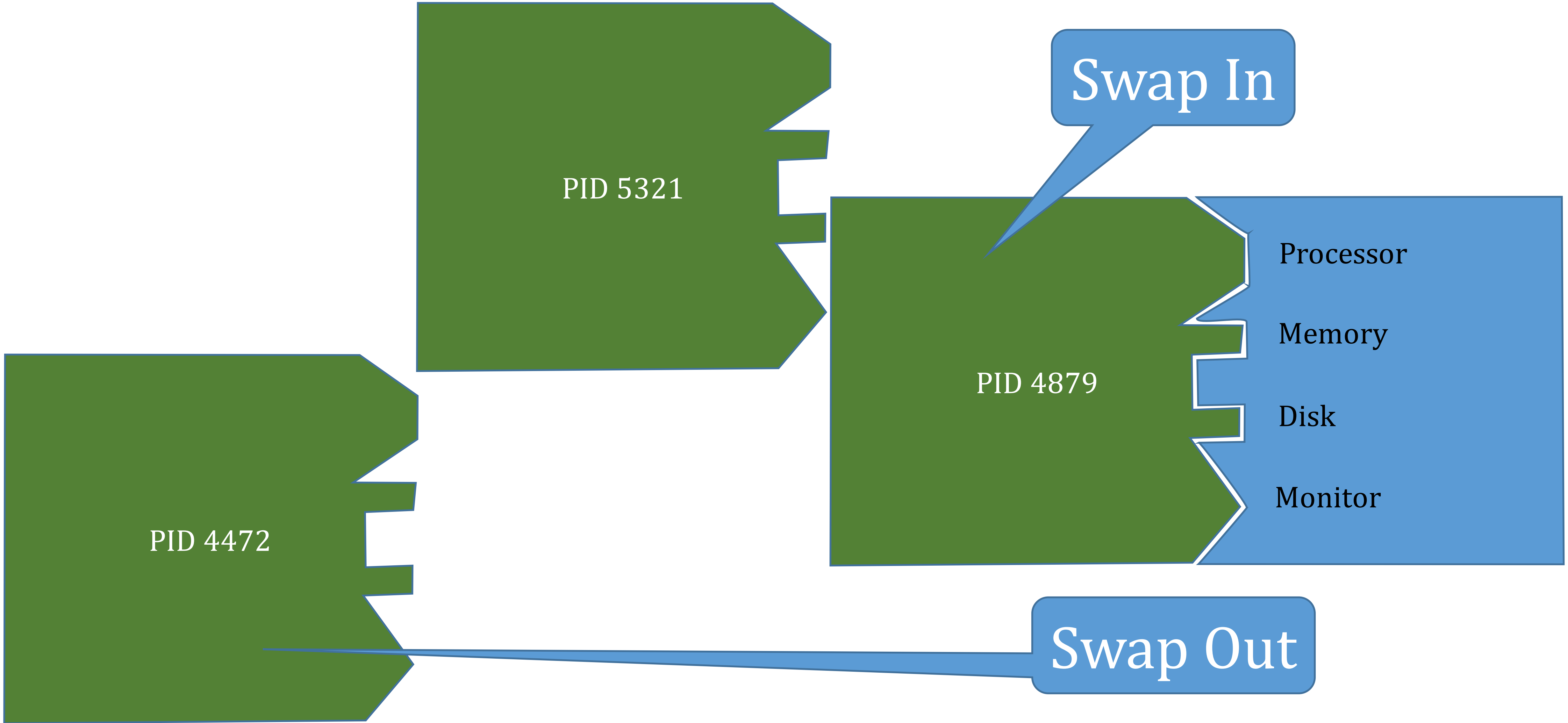
Virtual Memory (Real Memory POV)

Computer Systems Chapter 9.1 - 9.6

Process Resources

- Each process THINKS it owns all machine resources
  - “virtual” processor, virtual memory, virtual keyboard, virtual monitor, virtual disks, virtual network, …
- OS connects VIRTUAL resources to REAL resources

Time Slicing



Swapping Memory

Bad Idea:

Write Swap Out address space from memory to disk

Read Swap In address space from disk to memory

- A 32 bit address space is 4G
- Writing 4G to disk takes ~1G/sec or 4 seconds
- Times slices are MUCH smaller than 1 second
- You would spend 99.9999% of the time reading/writing memory!

Solution: Stay Tuned

Process Attributes

- Logical Control Flow
  - A process executes instructions
  - EIP points to the next instruction to execute
  - After an instruction is fetched, EIP points to the next sequential instruction
  - Control flow instructions modify EIP (jump, call, ret, etc.)
- Address Space
  - Memory starting at address 0x0000 0000 up to 0xFFFF FFFF
  - Contains OS, Code, Heap, Stack, bss, global data, shared libraries, etc.
- Registers / Register Values
- IO resources

Abstract View

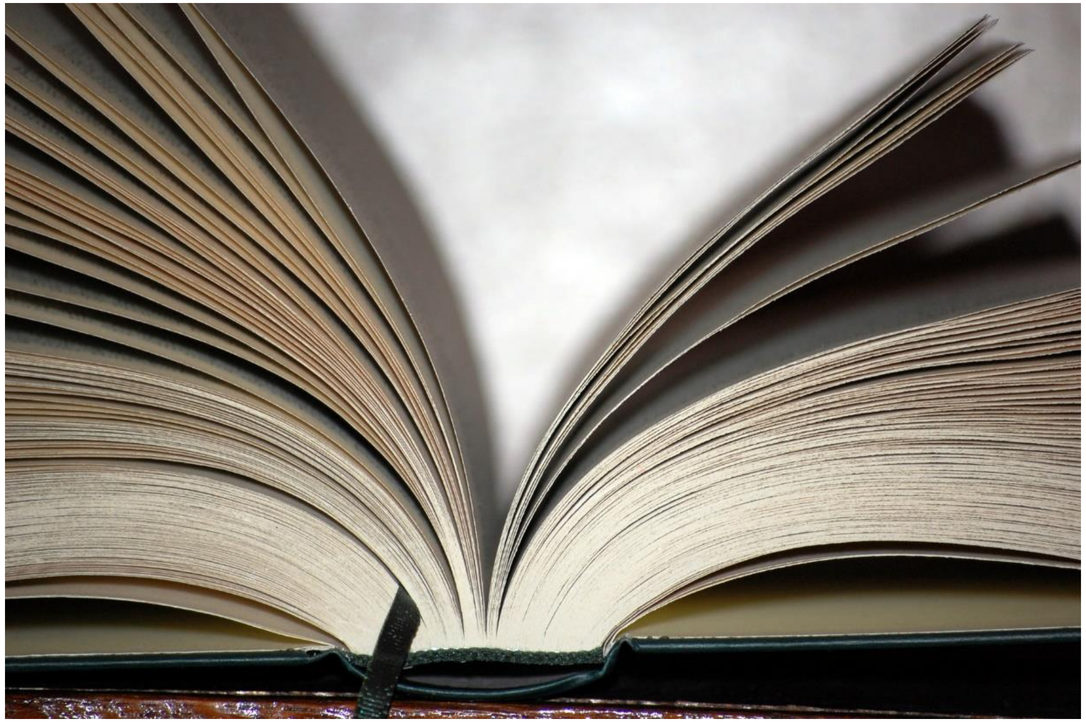
All memory is equally available;

I can address a single byte of memory using an address, which is just a number between 0 and 2^{32} - 1

2^{31}	2^{30}	…	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
b_{31}	b_{30}	…	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}

Pages

- A book consists of many pages
- Each page can contain a fixed amount of text
- Each page has a page number
- You can think of the book as a list of pages



Pages of Memory

- A virtual address space consists of many pages of memory
- Each page contains a fixed amount of data
- Each page has a page “number” or page ID
- You can think of an address space as a list of pages

x86 Stack

%esp
xffff fff4

%esp points at bottom of stack

Memory	
Address	Value
xffff fffc	
xffff fff8	x0000 0004
xffff fff4	x0000 0003
xffff fff0	
xffff ffec	
x0000 0004	
x0000 0000	

Top of stack at high memory

- Memory above %esp is in use
- Memory below %esp is available

Memory Pages

Each page is 4K = 4096 bytes long

$4096=2^{12}$

12 bits is 3 hex digits

	Memory	
Page	Address	Value
xffff f	xffff fffc	
	xffff fff8	x0000 0004
	xffff fff4	x0000 0003
	xffff fff0	
	xffff ffec	
xffff e		
x0000 0	x0000 0004	
	x0000 0000	

Project 2 “Warehouse” terminology

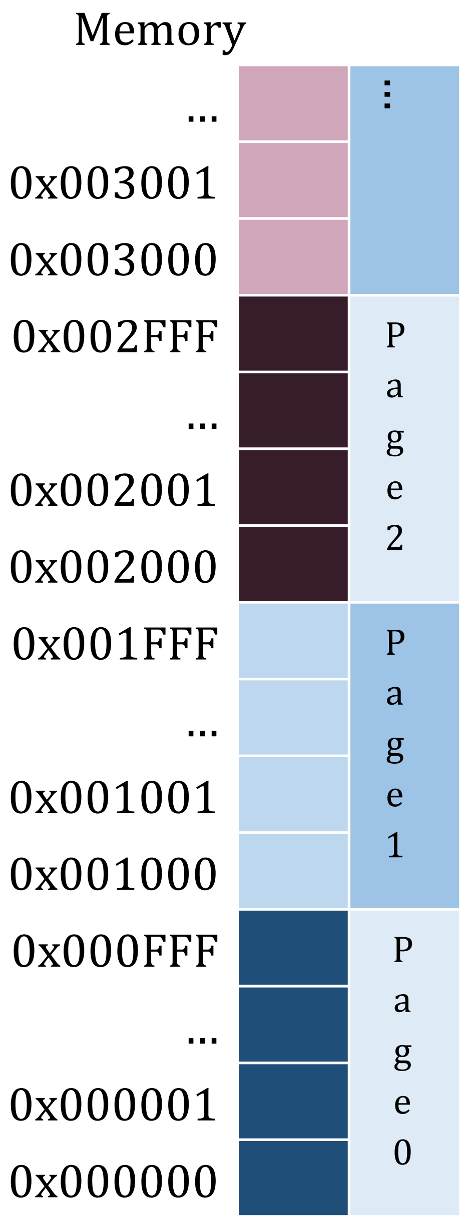
- Each page of memory is like a bin in a warehouse
- Each bin contains a fixed number of bytes (4096)
- To get to a specific byte, first go to the bin in the warehouse
  - Then go to the specific place in that bin to find the byte(s) you need

Page Addresses

- We can divide an address into sub-fields

2^{31}	2^{30}	…	2^{12}	2^{11}	2^{10}	…	2^{2}	2^{1}	2^{0}
b_{31}	b_{30}	…	b_{12}	b_{11}	b_{10}	…	b_{2}	b_{1}	b_{0}
Page ID				Page Offset					

- Equivalent to dividing memory into chunks
  - Chunk size = 2^{12} = 4K = 4096
  - Offset in 4K represented by 3 hex digits
  - For 32 bit addressing, Page ID is 5 hex digits



Example Address Fields

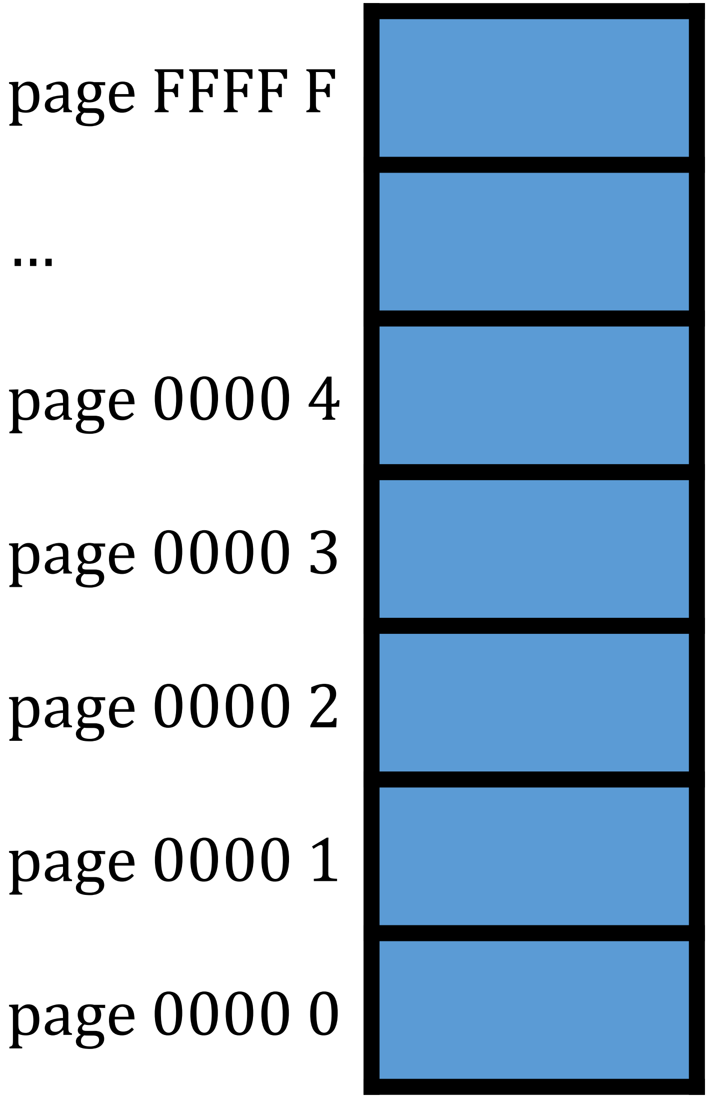
<table>
<tr><td colspan="32">ADDRESS</td></tr>
<tr><td colspan="20">PAGE ID</td><td colspan="12">PAGE OFFSET</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td colspan="4">F</td><td colspan="4">F</td><td colspan="4">F</td><td colspan="4">F</td><td colspan="4">D</td><td colspan="4">0</td><td colspan="4">3</td><td colspan="4">C</td></tr>
</table>

Memory Address: 0xFFFF D03C

Page: 0xFFFF D

Page Offset: 0x03C

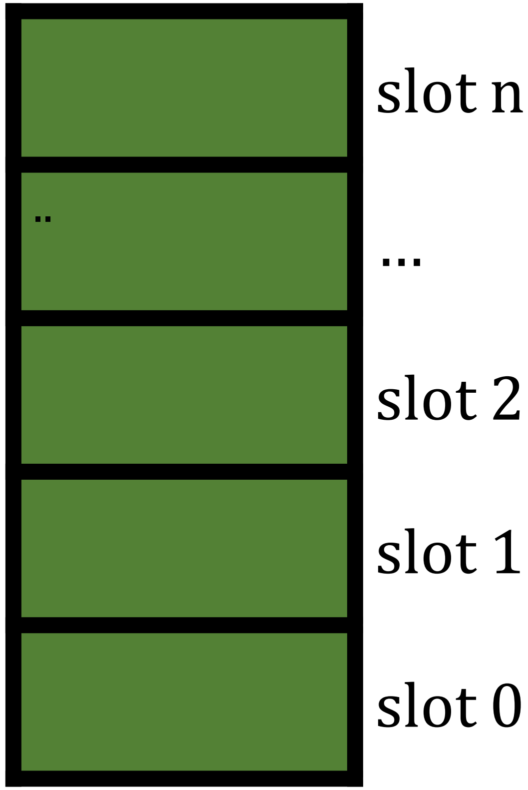
Address Space/Warehouse



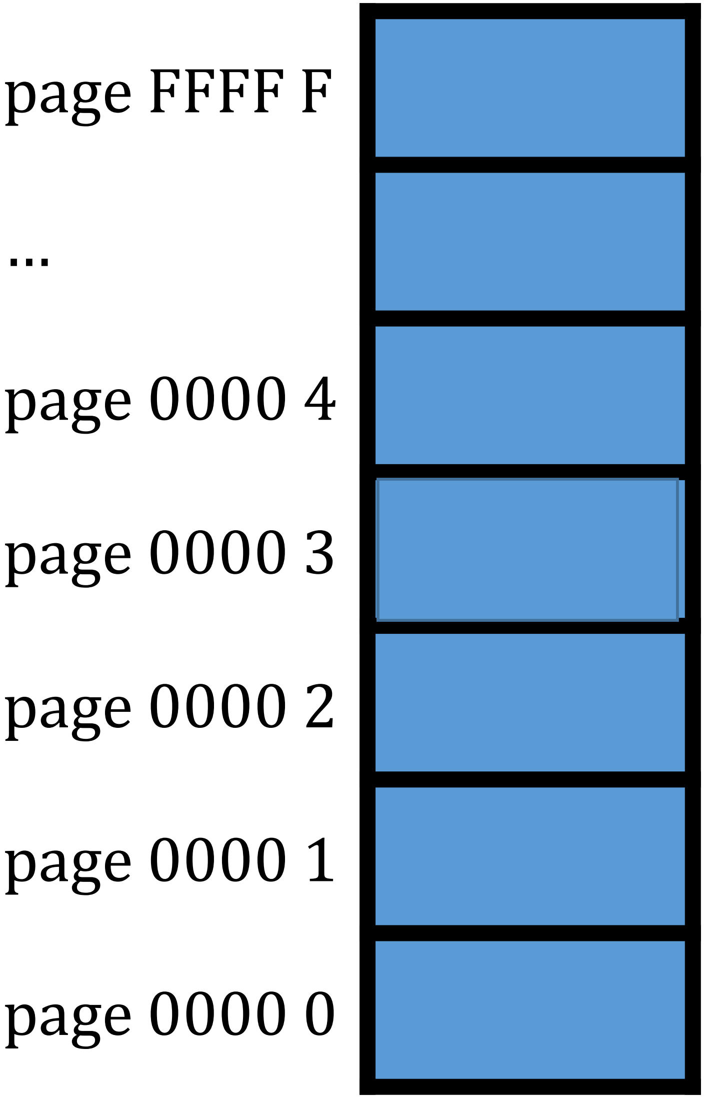
- List of pages
- Top is page 0xFFFF F
- Bottom is page 0x0000 0

Real Memory / Workbench

- Actual RAM in hardware
- Array of Page Slots
- Each slot is one page (4K) big
- Typically, << slots than pages

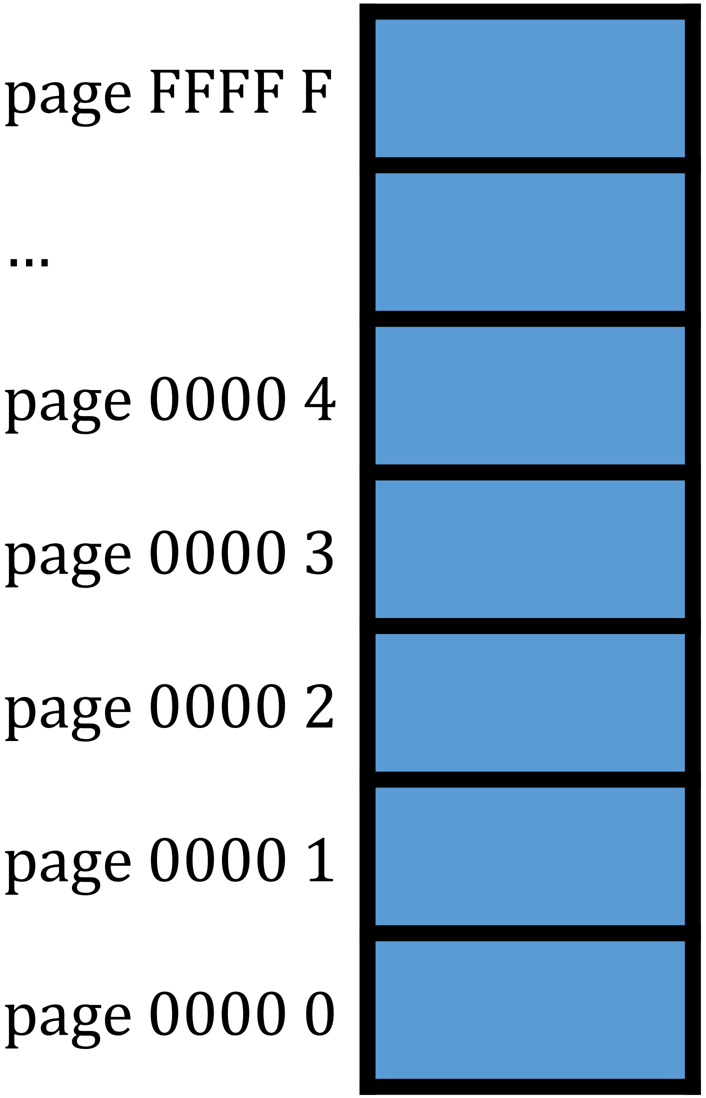


Page Swap In / Get bin from warehouse



Copy a page from address space into real memory

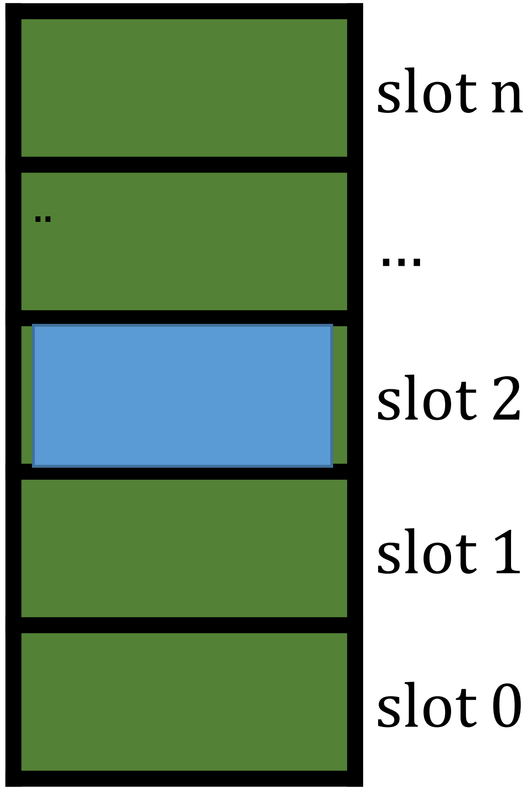
Page Table



Page	Slot
0000 3	2

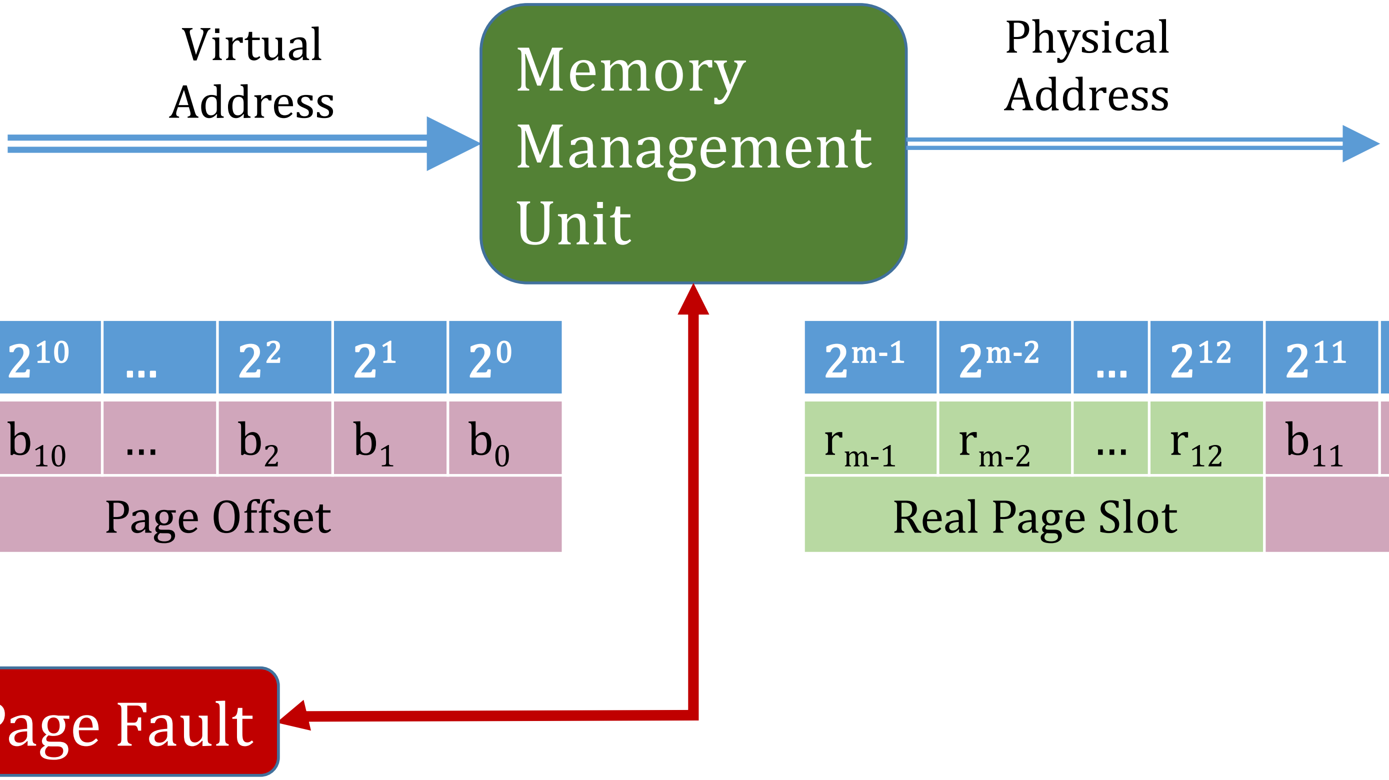
Track which pages are in which slots

Updated when PageSwap In occurs



Virtual Memory Shell Game

Page	Slot
0000 3	2

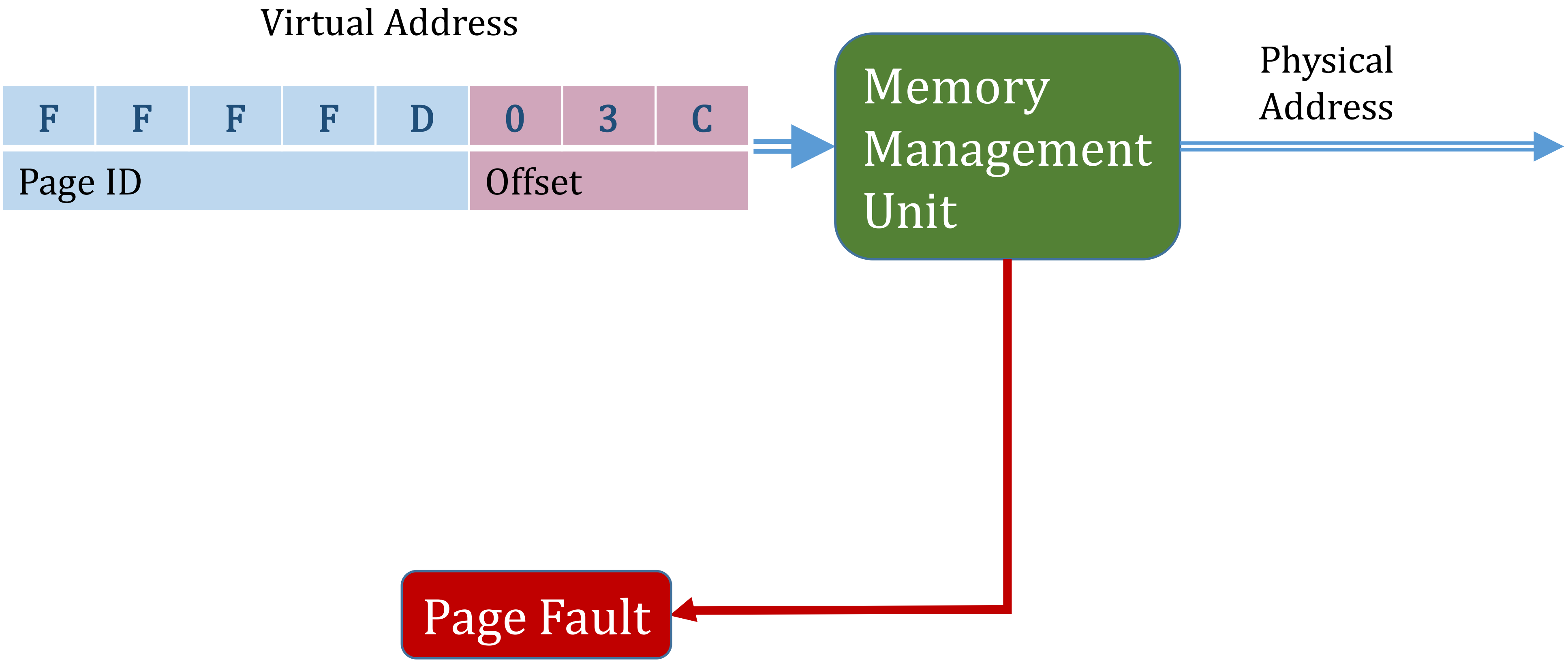


2^{31}	2^{30}	…	2^{12}	2^{11}	2^{10}	…	2^{2}	2^{1}	2^{0}
b_{31}	b_{30}	…	b_{12}	b_{11}	b_{10}	…	b_2	b_1	b_0
Page ID				Page Offset					

2^{m-1}	2^{m-2}	…	2^{12}	2^{11}	2^{10}	…	2^{2}	2^{1}	2^{0}
r_{m-1}	r_{m-2}	…	r_{12}	b_{11}	b_{10}	…	b_2	b_1	b_0
Real Page Slot				Page Offset					

Page Fault – Page not in Page Table!

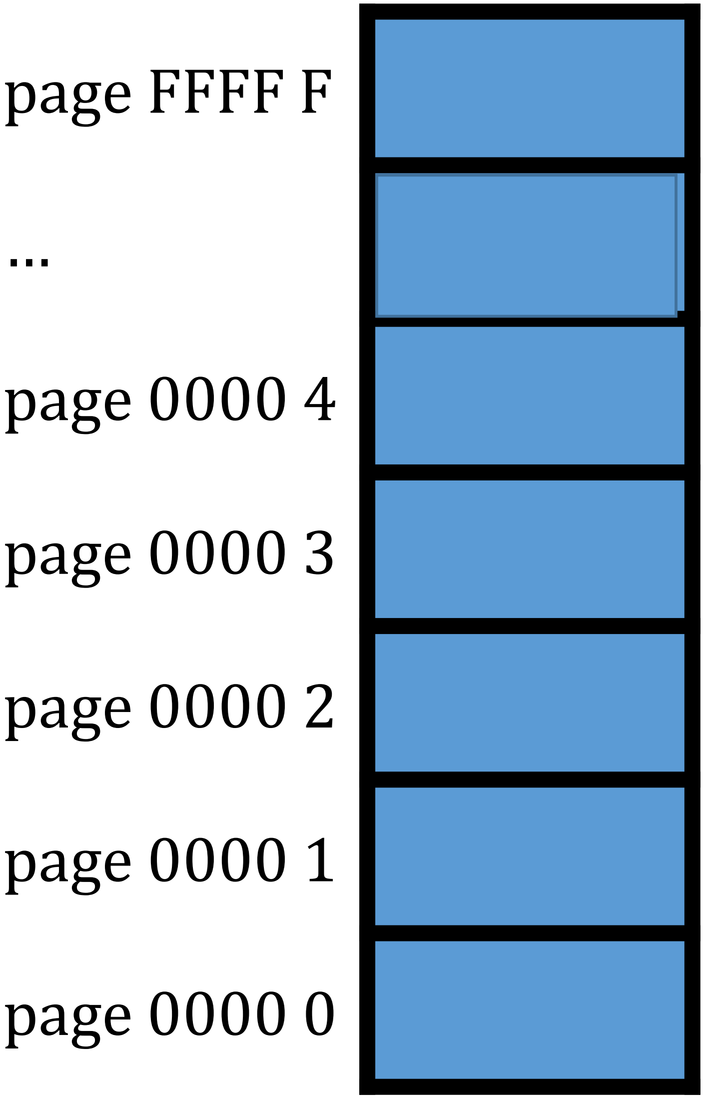
Page	Slot
0000 3	2



Page Fault – bin not on workbench

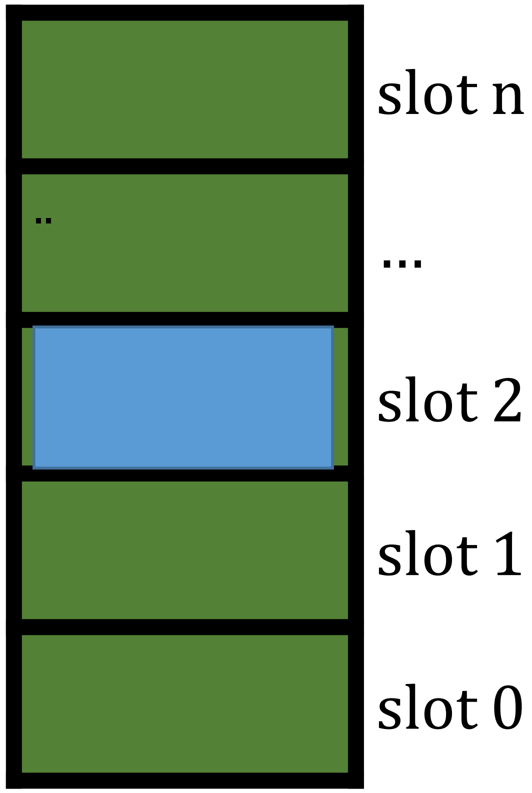
- Signals a page fault interrupt to the process/instruction requesting that address
- Process goes idle until page fault is resolved
- OS swaps page in to get that page into real memory
- Swap In updates the page table
- Signals “resume” when swap in is complete
- Process again becomes active
- Processing resumes with the currently executing instruction!
    - Current instruction had not yet completed.

Swap In



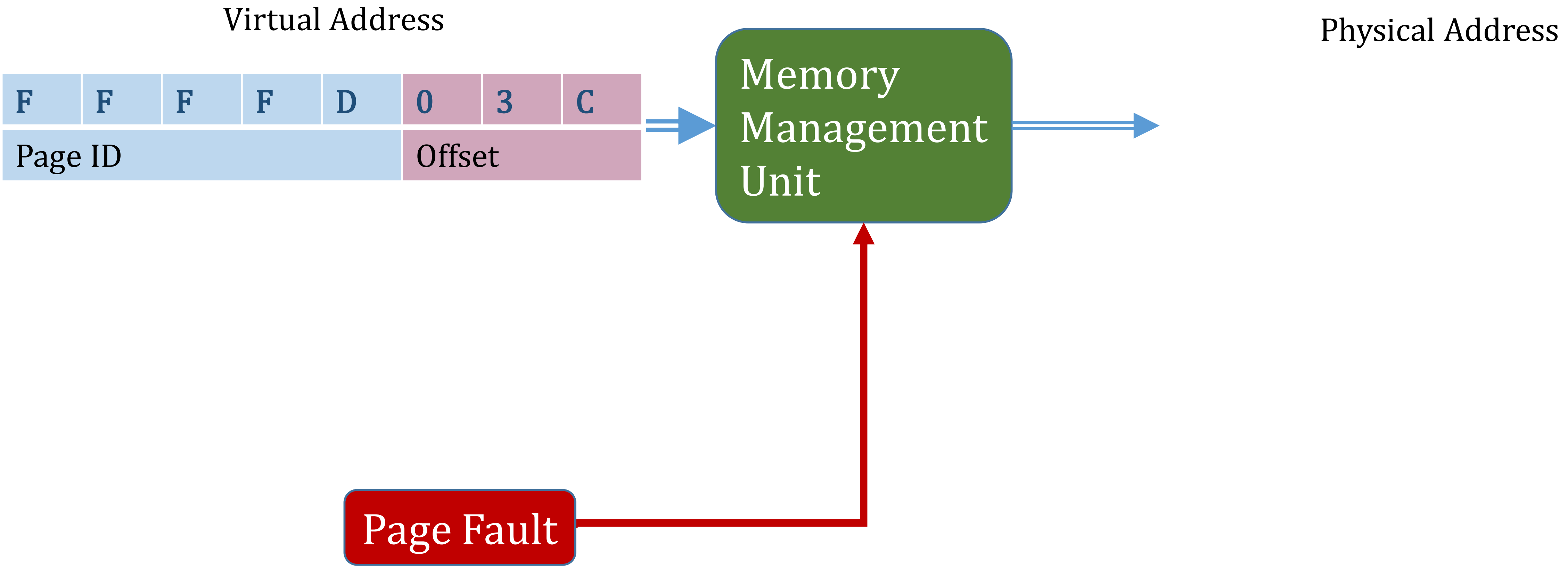
Copy new page to real memory and update page table

Page	Slot
0000 3	2
FFFF D	1



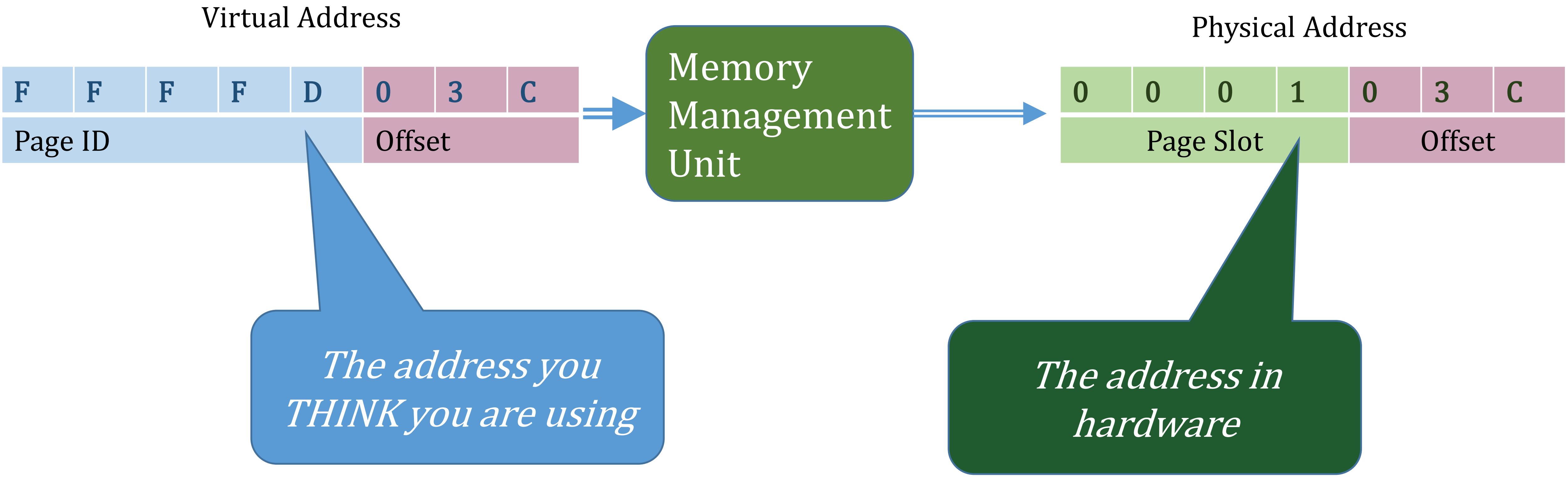
Swap In Complete

Page	Slot
0000 3	2
FFFF D	1



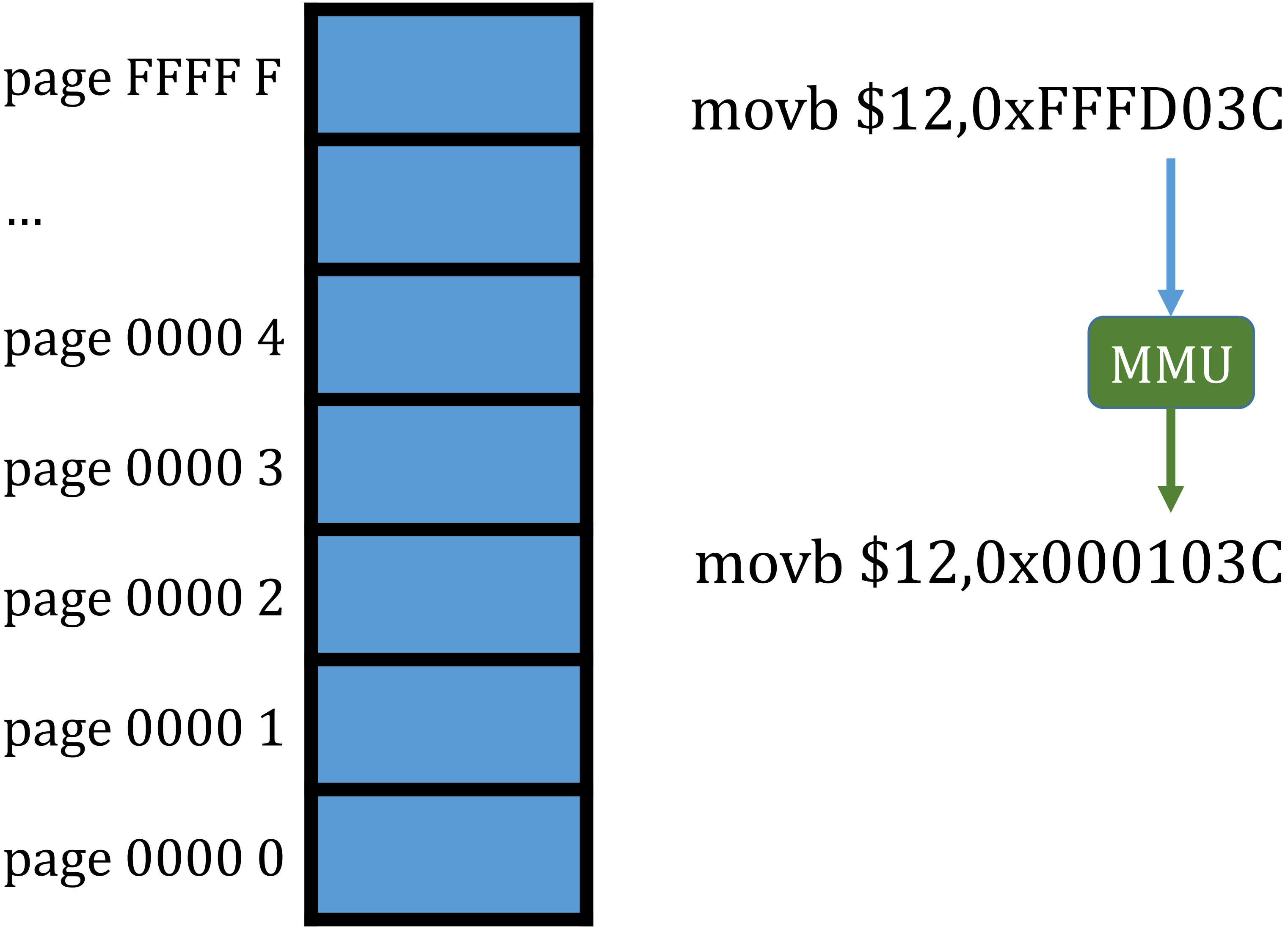
Virtual Memory Shell Game

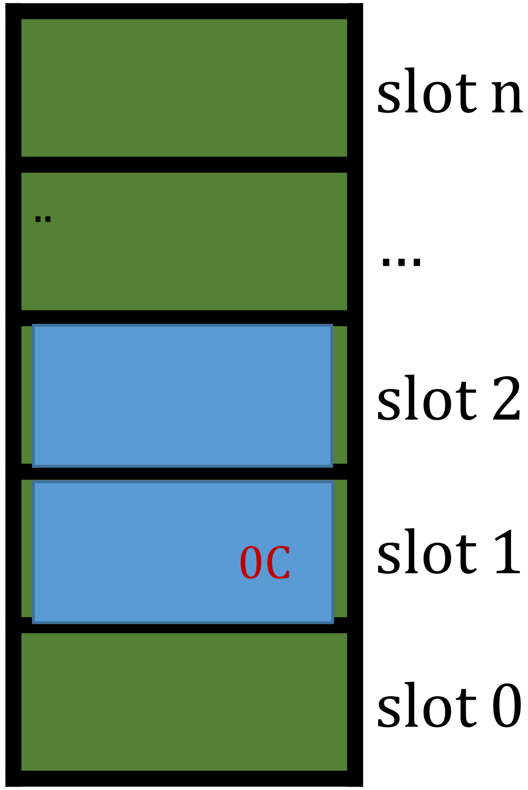
Page	Slot
0000 3	2
FFFF D	1



Memory Write

Page	Slot	Dirty
0000 3	2	0
FFFF D	1	1



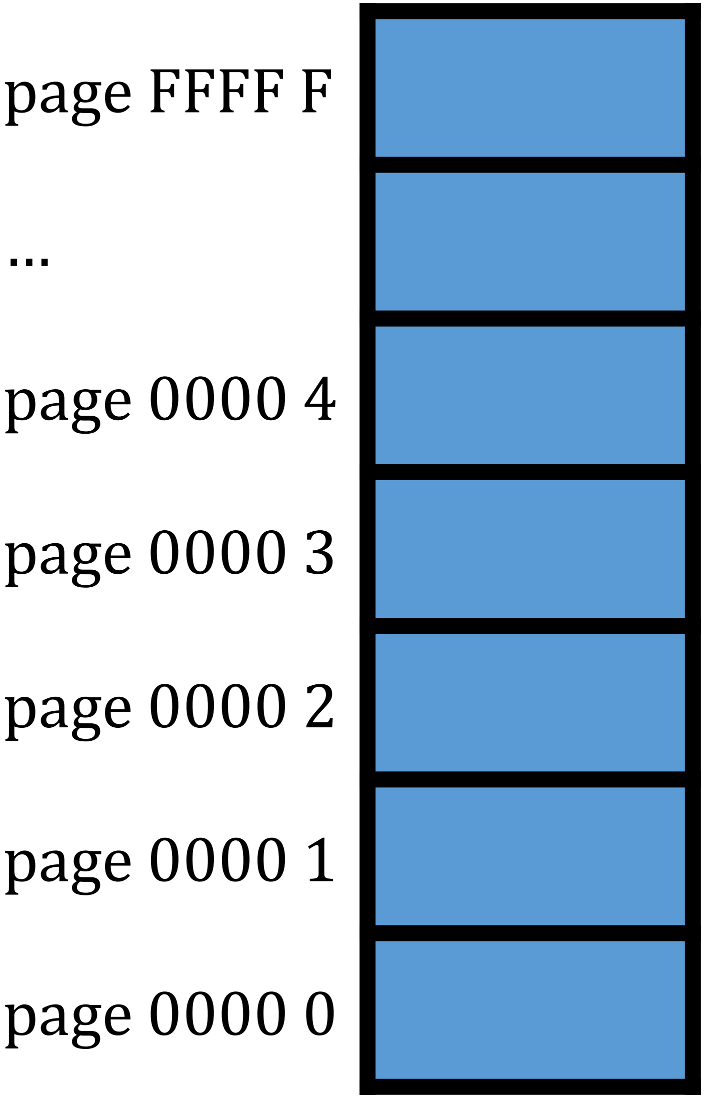


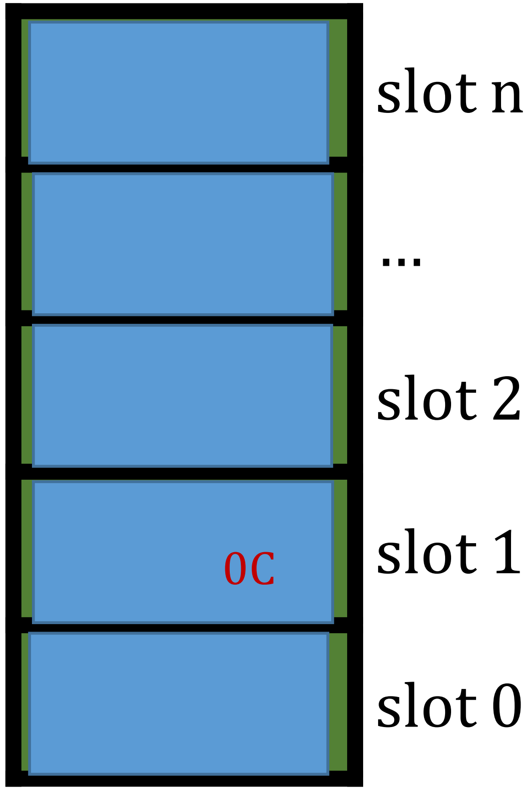
Page Table Dirty Bit

- Keeps track of whether the page in real memory is exactly the same as the page in virtual memory
- As soon as we write to memory in a page, that page becomes dirty

Eventually All Slots are Filled

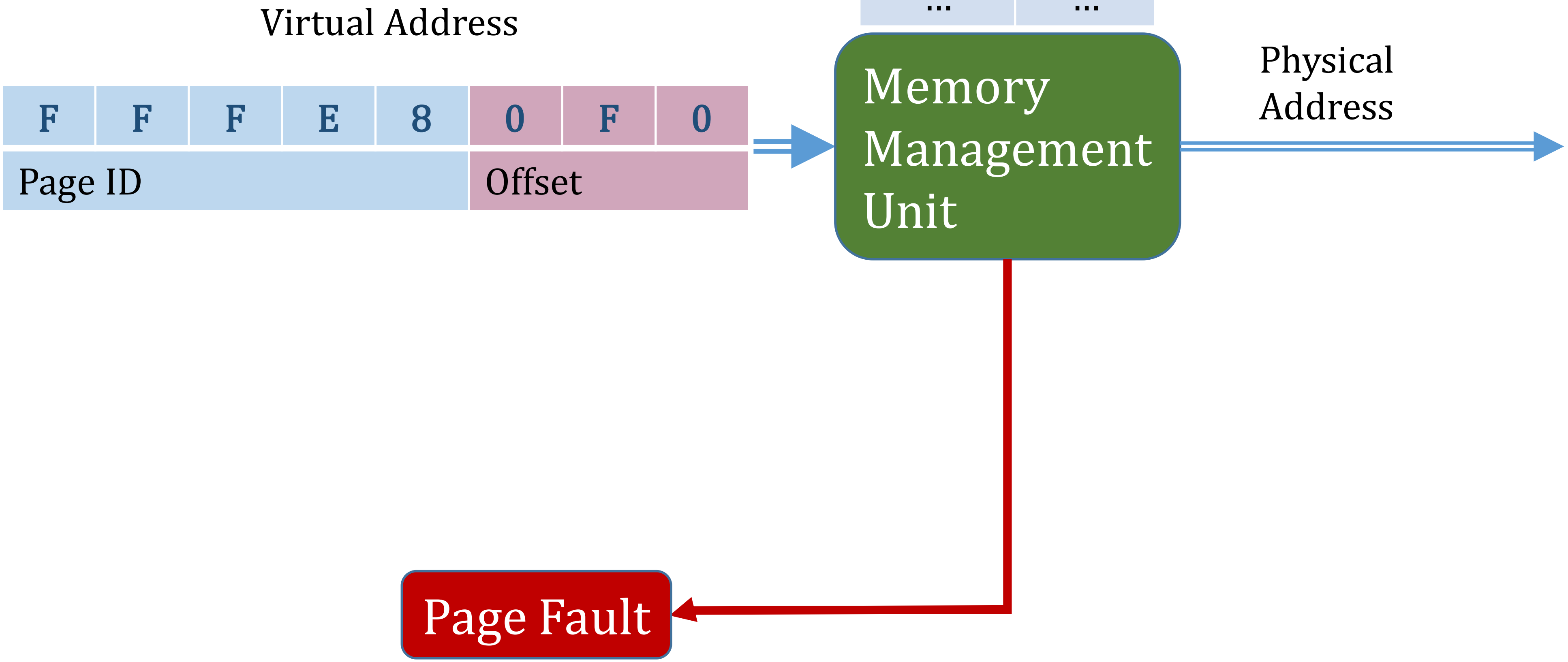
Page	Slot	Dirty
0000 3	2	0
FFFF D	1	1
…	…	…





Page Fault – Page not in Page Table!

Page	Slot
0000 3	2
FFFF D	1
…	…



Page Swap Out / Workbench is full

- Need to make room in real memory for a new slot
- Need to choose a "victim"… a page already in a real memory slot that can be sacrificed
- Victim Choice is the MOST important algorithm in terms of performance!!!!!!
  - Don't swap out the page that the next instruction needs!
- If a real memory slot is dirty, need to update virtual memory with updated contents

Page Ejection Algorithms

- How do we choose which page to swap out?
  - Don’t want to swap out a page we are about to use again!
  - Hard to predict what future memory requests will be.
- Locality
  - Future memory requests will be near current memory requests
  - Future memory requests will be near recent memory requests
- Least Recently Used (LRU)
  - Eject the page referenced the longest time ago
  - True LRU is expensive (updated every memory access)
  - Random – cheap but often causes higher miss rate

Random Page Ejections

- Throw out a page chosen at random
- Cheap to implement
- Probability of selecting the best page is 1/#slots
- Probability of selecting the worst page is 1/#slots

Page Ejection by “era”

- When a page is referenced, set “touched” to true
- When a page ejection is required
  - Find first page for which “touched” is false
  - If no such pages exist, set all pages “touched” flag to false (new “era”)
- Advantages / Disadvantages
  - Relatively cheap and fast
  - Occasionally (once an era) requires reset (Full page table update)
  - Early in the era, occasional “bad” choice
  - Late in the era, close to LRU
  - Cheap version of LRU

Linked List LRU

- Each page table entry has a “next” and “prev” pointer
- At memory reference, put page table entry at list head
- At page ejection time, eject page at list tail

- Advantages / Disadvantages
  - Requires list update on every memory reference (expensive)
  - Allows search of page table in LRU order (head to tail)
  - True LRU algorithm – best theoretical hit rates

Paging Performance

- Virtual memory usually kept on disk
- Reading from disk is about 100x slower than reading from RAM
- Every Page Swap requires disk read (and maybe write)
- Performance depends on # Page Faults / time
- Typically measured as “Page Hit Rate”

$$Page\ Hit\ Rate = \frac{\#\ Memory\ Access\ in\ Real\ Memory}{\#\ Memory\ Accesses}$$

Page Hit Rate

- Often achieve page hit rate of 99.99+%
- The higher the page hit rate, the closer virtual memory speeds are to real memory speeds
- If the page hit rate gets too slow, we start “thrashing”
  - Spend more time swapping pages in and out than doing real work
- The more real memory, (more page slots) the higher the page hit rate.

Page Hit Rates – Reading Code

- For instructions
  - Most instruction fetch is just a couple of bytes from previous %eip
  - If the average instruction length is 4 bytes, page fault for every 1K instructions for sequential code
- Branches
  - Local branches… branches to locations within the page
    - If loop around 50% of your code, and execute 100 times, then page fault every 50K+ instructions
  - Far branches… branches outside the page
    - Cause page fault
    - Very rare

Page Hit Rate – Stack Space

- Function references local data, and parameters in stack frame or nearby stack frame
- Stack frame most likely in a single page or at least a small set of pages
  - Compiler rounds stack to even boundary to ensure single page
- Stack frame page(s) swapped in when function starts (or already there if there is room in the prev. frame's page)
- Sometimes get a page miss when function is called
- Page misses very rare otherwise!

Page Hit Rate – Heap Space

- Much more likely to get page misses on dynamically allocated memory
  - Dynamically allocated memory is often larger than local variables
  - No guarantee that dynamically allocated memory is near other dynamically allocated memory

Why Virtual Memory?

- Enables very large virtual address spaces
  - RAM is expensive
  - Disk space is cheap
- Enables “sleep” mode (write dirty pages, and empty real memory)
- Enables per-page Memory Protection (Permissions in page table)
- Enables “memory mapped” IO (more later)
- Enables independent virtual address space for each process
- Enables fast process swap in / swap out (more later)