Board Notes on Virtual Memory

Part A: Why Virtual Memory?

- Let's user program size exceed the size of the physical address space
- Supports protection
 - o Don't know which program might share memory at compile time.

Consider the following:



- Above:
 - Assume 4KB pages therefore, think about "groups of 2¹² pieces of data"
- Usually, virtual address space is *much* greater than physical address space
 - (Mapping allows code with virtual address to run on any machine.)

Part B:

How do we translate a Virtual Address to a Physical Address

(or alternatively, "How do we know where to start looking in memory?")

- Good analogy: It's like finding what cache block a physical address maps to.

Example:

 What if 32-bit virtual address (2³² virtual addresses), 4KB pages (like above), 64 MB of main memory (2²⁶ physical addresses)

How is this mapping done?

VPN (Virtual Page Number)	OFFSET
PFN (Physical Frame Number)	OFFSET

How do we do VPN \rightarrow PFN mapping?

- Leverage structure called page table
- To make analogy to cache, "data" = PFN
- To make analogy to cache, also have valid, dirty bits
- -

- If no valid mapping, get page fault:
 - Try to avoid
 - o Involves lots of disk traffic
 - o Placement in memory done fully associative, LRU to minimize
 - Placement = some extra overhead, but small percent and worth it to avoid M CC penalty

Offset still the same because we go down the same distance

More specifically:

The process works like this...



Even more specifically...

- The page table is stored in memory
- The beginning of the page table is stored in the page table register
- OS knows where PT for each program begins; interfaces with architecture to find



Part C:

How big is the page table?

- Page table can actually become pretty big...
- Example #1:
 - 4 KB pages
 - Therefore need 2¹² (or 12 bits of offset)
 - (Offset does same thing that it does in cache block not just picks page entry)
 - 32-bit virtual address
 - 32 12 = 20 bits of VPN
 - 4 Byte / page table entry
 - Holds LRU status, valid, dirty, PFN (~32 bits)
- Therefore, 2^{20} entries in page table, each ~ 4 bytes each \rightarrow 4 Mbytes
 - o Not as big as memory, but what about cache?

Another Example...

- Assume
 - Virtual address = 64 bits
 - o 4 KB pages
 - 4 bytes/page

	VPN (52 bits)	Offset (12 bits)
--	---------------	------------------

- PT would be:
 - $4.5 \times 10^{15} \times 4 \sim 10^{16}$ bytes \rightarrow 10 petabytes!
- Soluton(s):
 - o Multi-level, inverted page tables you'll learn about in OS

Part D:

Is page table / virtual address translation slow?

- It can be \rightarrow have maybe 2 references / translation
 - Solution: TLB = "Translation Lookaside Buffer" • Fast cache for page table

What does the TLB look like?

- It's a really small, fully associative cache

Virtual Page #	Physical Frame #	Dirty	LRU	Valid

- 1. All of the virtual page numbers would be searched for a match
- 2. The physical frame number is the data that is supplied
- 3. The physical frame number is concatenated with an offset to form a physical address



No

Cache miss

stall

Cache hit?

Yes

Deliver data to CPU

No

TLB miss

stall

Cache/buffer

memory write

Where is the TLB on the critical path?

Yes

Replace

page from

disk

Part E: Example 1:

Assume a machine with the following characteristics:

- The CPU supplies a 64 bit virtual address.
- The 64 bit virtual address must be translated into a 64 bit physical address.
- This single core machine has <u>2 levels of cache</u>.
- The clock rate is 1 GHz. -

Question A:

The page table has 2³² entries. Determine the physical address associated with the virtual address using portions of the memory state of the machine provided below. (Addresses hex.)

Page Table Register: AAAA 0000 0000 0000 Virtual Address: 0000 BBBB 0000 AAAA

	Address (least significant to most significant)	Data
1	0000, 0000, 0000, 0000	FEDC, BA98, 7654, 3210
2	0000, 0000, 1111, 1111	0123, 4567, 89AB, CDEF
3	0000, 0000, AAAA, 0000	0000, 0000, 3333, 7777
4	0000, 0000, DDDD, 9988	2222, 7777, 8888, 4444
5	0000, 000F, 0000, 0000	E1E1, 0E1E, 10E1, E100
6	0000, BBBB, 0000, AAAA	0000, 0000, EEEE, FFFF
7	0000, BBBB, AAAA, AAAA	0000, 0000, 9999, 2222
8	000A, AAA0, 0000, 0000	0000, 0000, 2222, 2222
9	0BBB, AAAA, 0000, AAAA	0000, 0000, 5555, 4444
10	AAAA, 0000, 0000, 0000	0000, 0000, BBBB, CCCC
11	AAAA, 0000, 0000, BBBB	0000, 0000, FFFF, EEEE
12	AAAA, BBBB, BBBB, AAAA	1111, 0000 AAAA, 0000
13	BBBB, AAAA, 0000, 0000	0000, 0000, 9999, 1111
14	BBBB, BBBB, BBBB, AAAA	0000, 0000, 9999, 8888
15	CCCC, CCCC, DDDD, DDDD	8888, 9999, 0000, 1111
16	FFFF, CCCC, DDDD, AAAA	1111, 1111, 0000, AAAA

Goto memory location

Get:

Physical address is:

+

AAAA 0000 0000 0000 0000 BBBB AAAA 0000 0000 BBBB 0000 0000 FFFF EEEE FFFF EEEE 0000 AAAA

Question B:

The physical address generated above is then sent to a direct mapped, L1 cache. The L1 cache has the following characteristics:

- The cache has 4096 blocks.
- There are 256, 32-bit words in each block. Addresses are to the word.
- The cache can hold 4 MB (i.e. 4,194,304 bytes) of data.

Using the physical address found in Part A, fill in the following table:

Index	OAA
Offset	AA
Tag	FFFF EEEE 000

Physical address: FFFF EEEE 0000 AAAA

4096 Blocks $\rightarrow 2^{12} \rightarrow 12$ bits of index $\rightarrow 0AA$

256 words/block \rightarrow 2⁸ words/block \rightarrow 8 bits of offset \rightarrow AA

Therefore there are 64 - 12 - 8 = 44 bits of tag \rightarrow FFFF EEEE 000

Question C:

Assume that you have a sequence of 3 virtual addresses that you need to convert to physical addresses. The first virtual address takes 3 nanoseconds to translate to a physical address. The second virtual address takes 300,000 nanoseconds to translate to a physical address. The third virtual address takes 100 nanoseconds to translate to a physical address. For each virtual address, *briefly* comment on the critical path of translation.

3 ns	TLB hit + cache hit
300000 ns	TLB miss + page fault
100 ns	(1) TLB Miss + Page Table Hit OR
	(2) TLB Hit + Cache Miss

Part E: Example 2:

Assume that the 1st three lines of code for Microsoft Word are as follows – really!

Address 1:	10000	# static variable 10000
Address 2:	20000	# static variable 20000
Address 3:	Load R17 Address 1	# Load 10000 into R17

Thus, we want to load the data at address 1 of the Microsoft Word assembly code into register #17.

Question A:

Using a modern superscalar machine's datapath as context (with two levels of on-chip cache), list all of the steps involved with loading this initial value stored in the program code written by Microsoft into physical register #17. You have just turned your machine on and are loading Microsoft Word for the first time.

TLB Miss, Page Table Lookup, TLB Update, L1 \$ Miss, L2 \$ Miss, Memory Reference, L2 Update, L1 Update, data into R17.

<u>Question B:</u> Provide a rough estimate of the amount of time that this would take.

Seconds!