Lecture 14 Virtual Memory Exercises

Q1:
Given the same main memory configuration as **Lecture 12 Exercises**:

- 2 memory channels
- 4 DRAM DIMMS (2 DIMMS per channel)

Each DIMM has:

- 1 rank
- 8 chips per rank
- 8 bits per column
- 4 banks per chip
- 32,768 rows per bank
- 1,024 columns per bank
- 8-byte bus

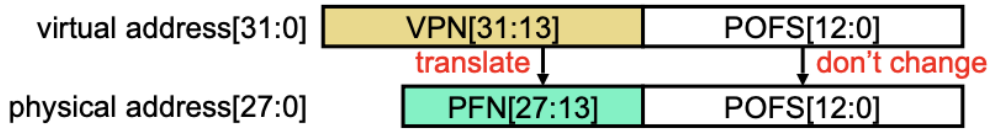
Also assume

- The physical address space has 1M (i.e., 1048576) pages (physical frames)
- Virtual addresses have 64 bits

What is the size of a page? What is the maximum number of pages in the virtual address space?

Solution:
Based on Lecture 12 Exercises, the total physical memory size is 4GBytes
So, page size = 4GBytes / 1048576 = 4096 Bytes

Think about the example address translation in the Lecture 14 notes:



Based on the assumption of this question, bits in page offset = log2(4096) = 12 bits
Therefore, bits in virtual page number = 64 – 12 = 52 bits
Maximum pages in virtual address space = 2^52 = 4503599627370496 Bytes

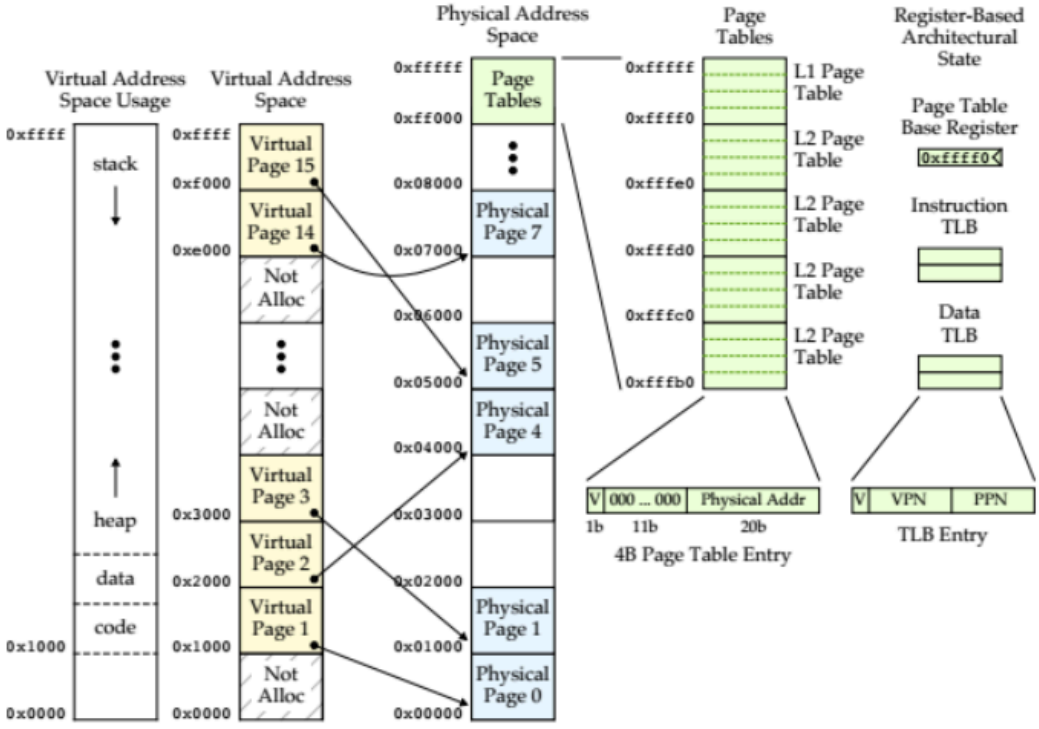
Q2:
As discussed in Lecture 14 video, a two-level page table is a space-efficient way to translate virtual addresses to physical addresses. The L1 page table entries point to L2 page tables, and the L2 page table entries point to the corresponding page in physical memory. The virtual address is used to "walk" the page table. Some bits of the virtual address are used to index into

the L1 page table, different bits of the virtual address are used to index into the L2 page table, and finally the page offset bits are used to index into the physical page.

In this problem, we will be exploring a small-scale page-based memory translation system that uses 4 KB pages and a two-level page-table. We will assume that all addresses are byte addresses, virtual addresses are 16 bits (i.e., we have 64KB of virtual memory), and physical addresses are 20 bits (i.e., we have 1MB of physical memory). We have more physical memory than virtual memory to enable multiple programs to be resident in physical memory at the same time. *While these small memory spaces are not realistic, they will help simplify the problem.*

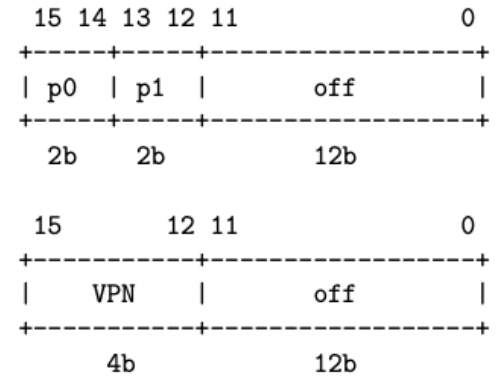
Assume program A was running for some amount of time, was context swapped by the operating system so that program B could run, and is now being context swapped back onto the processor. This means that some amount of physical memory has already been allocated to program A and the two-level page table is already initialized and stored in physical memory. However, since a context swap flushes the TLB, all entries in the TLB are now invalidated.

The Figure below shows the state of the system when we restart execution of program A. Note that only 5 virtual pages have been allocated to program A; the remaining 11 virtual pages are unallocated. The L1 page table and each L2 page table has 4 entries. The page tables are stored at the very top of the physical memory address space. The L1 page table starts at address 0xffff0 and the L2 page tables are directly below the L1 page table. All page-table entries (PTEs) are assumed to be 4 bytes: one valid bit, 11 bits that are always zero, and 20 bits for a physical address. A page-table base register is already initialized to point to the base of the L1 page table.



Q2A: Which bits of the virtual address are used for: (a) the page offset, (b) the virtual page number (VPN), (c) indexing into the L1 page table, and (d) indexing into the L2 page table?

Solution:
Page offset is 12 bits, index into L1 and L2 page tables are both 2 bits, and the VPN is 4 bits. Address mappings shown below:



Here, p0 and p1 are used as indexes for referencing the L1 and L2 page table entries, respectively.

Q2B (Challenging problem, not required):
The L1 page table has four PTEs, and there are four L2 page tables each with four PTEs for a total of 20 PTEs. These page tables are stored in physical memory. **Fill in the following table, which shows the contents of physical memory where the page tables for program A reside.** We have provided one page-table entry for the L1 page-table to get you started.

	Page Table Entry	
Physical Addr of PTE	Valid Bit	Physical Address
0xffffc		
0xffff8		
0xffff4		
0xffff0	1	0xfffb0
0xfffec		
0xfffe8		
0xfffe4		
0xfffe0		
0xfffdc		
0xfffd8		
0xfffd4		
0xfffd0		
0xfffcc		
0xfffc8		
0xfffc4		
0xfffc0		
0xfffbc		
0xfffb8		
0xfffb4		
0xfffb0		

As an aside, we probably should not have pre-allocated all five page tables! As you will see, only a subset of these page tables actually need to be allocated, so by pre-allocating all five pages we have mitigated the key advantage of a two-level page table compared to a one-level page table. Please note that if all of the PTEs in a L2 page table are invalid, then there should not be a valid PTE entry in the L1 page table pointing to this L2 page table. In other words, let's try and capture the idea that we would not really need to allocate L2 page tables for which all entries are invalid.

Solution:

	Page-Table Entry	
Paddr of PTE	Valid	Paddr
0xffffc	1	0xfffe0
0xffff8	0	
0xffff4	0	
0xffff0	1	0xfffb0
0xfffec	1	0x05000
0xfffe8	1	0x07000
0xfffe4	0	
0xfffe0	0	
0xfffdc	0	
0xfffd8	0	
0xfffd4	0	
0xfffd0	0	
0xfffcc	0	
0xfffc8	0	
0xfffc4	0	
0xfffc0	0	
0xfffbc	1	0x01000
0xfffb8	1	0x04000
0xfffb4	1	0x00000
0xfffb0	0	

Contents of Physical Memory with Page Tables

Hint: For example, if program A wants to access Virtual Page 2, i.e., virtual address is 0x2000, then based on the address mapping in Q2A, p0 = 00 and p1 = 10. How to determine which L1 and L2 PTEs are used and the value stored in those PTEs? Steps:

- p0 points to the L1 page table entry (PTE) with an index of 00, i.e, the first PTE in the L1 page table. Based on the figure, the physical address of this PTE is 0xffff0. As this PTE is accessed, its valid bit is set to 1.
- As this is the first PTE in the L1 page table, it will point to the starting address of the first L2 page table. This starting physical address of the first L2 page table is 0xfffb0. Therefore, the value stored in the first L1 PTE is 0xfffb0.
- We then use p1 to index into this L2 page table. As p1 = 10, it indicates we need to index into the L2 PTE with an index of 10, i.e., the third PTE in this L2 page table. What is the physical address of this PTE? Because the size of each PTE is 4B, the address of

the third PTE = 0xfffb0 + 4 + 4 = 0xfffb8. As the PTE with the address of 0xfffb8 is accessed, its valid bit is set to 1.

- What is the value stored in this third L2 PTE? The physical address of the Virtual Page 2. Based on the figure, Virtual Page 2 is mapped to Physical Page 4 and the physical address is 0x04000. So 0x04000 is the value that is stored in this third L2 PTE.

Q2C (Challenging problem, not required):
A two-level page table requires two additional memory accesses for every instruction or data memory request. A translation-lookaside buffer (TLB) can be used to cache translations and provide single-cycle mappings between virtual to physical addresses. Each TLB entry includes a valid bit, virtual page number (VPN), and physical page number (PPN). TLBs are usually flushed on a context swap. This is one step in implementing memory protection. Flushing the TLB prevents one program from accidentally using an old translation in the TLB to access physical memory allocated to a different program. Unfortunately, this results in TLB misses when a program restarts execution.

We will assume that program A was in the middle of copying a large amount of data from the stack to the heap when it was context swapped. Now that program A is restarting, it will continue copying the data from the stack to the heap. This results in the following address stream:

```
0xeff4, 0x2ff0, 0xeff8, 0x2ff4, 0xeffc, 0x2ff8, 0xf000,
0x2ffc, 0xf004, 0x3000, 0xf008, 0x3004, 0xf00c, 0x3008
```

We will be focusing on a two-entry, fully associative TLB exclusively for data memory accesses (i.e., instruction memory accesses use a different TLB). Assume the TLB uses a least-recently used replacement policy. **Create a table similar to the one shown below which shows the state of the TLB during the given sequence of data memory request transactions.**

Fill in the VPN and page offset for each transaction before updating the virtual page number (VPN) and physical page number (PPN) of each TLB entry after each transaction. Indicate which accesses result in a TLB miss or hit. Indicate the total number of memory accesses for each transaction (i.e., include any accesses to the page tables and the actual access corresponding to the memory transaction). Include the total number of TLB misses and the TLB miss rate in the table. *You only need to fill in elements in the table when the value changes! Remember that the TLB entries should always reflect the state of the TLB before the corresponding transaction on that row executes!*

To get you started, we have filled in the table for the first transaction. Use a dash (–) to indicate an invalid TLB entry (all TLB entries are initially invalid).

Transaction Address	VPN	Page Offset	m/h	Total Num Mem Accesses	TLB Way 0		TLB Way 1	
					VPN	PPN	VPN	PPN
0xeff4	0xe	0xff4	m	3	–	–	–	–
0x2ff0	...				0xe	0x07		
0xeff8	...							
Number of Misses =								
Miss Rate =								

TLB Contents Over Time

Solution:

Transaction Address	VPN	Page Offset	m/h	Total Num Mem Accesses	TLB Way 0		TLB Way 1	
					VPN	PPN	VPN	PPN
0xeff4	0xe	0xff4	m	3	–	–	–	–
0x2ff0	0x2	0xff0	m	3	0xe	0x07		
0xeff8	0xe	0xff8	h	1			0x2	0x04
0x2ff4	0x2	0xff4	h	1				
0xeffc	0xe	0xffc	h	1				
0x2ff8	0x2	0xff8	h	1				
0xf000	0xf	0x000	m	3				
0x2ffc	0x2	0xffc	h	1	0xf	0x05		
0xf004	0xf	0x004	h	1				
0x3000	0x3	0x000	m	3				
0xf008	0xf	0x008	h	1			0x3	0x01
0x3004	0x3	0x004	h	1				
0xf00c	0xf	0x00c	h	1				
0x3008	0x3	0x008	h	1				
Number of Misses =		4						
Miss Rate =		0.29						

TLB Contents Over Time