Liane Yanglian (net ID:xy48),
Compsci 250,
Assignment 5
April 10, 2017,

1. Write-back caches are usually also write-allocate because the two complement each other in their functionalities, specifically reducing writing to memory by reducing trips that the program has to make to update memory. To reduce writing to main memory, when there is a write hit, write-back writes to the block in the cache only and the modified cache block is written to main memory only when the set is too small and the block needs to be replaced; while when there is a write miss, write-allocate brings the block to cache, in hopes that subsequent writes to that block will be captured by the cache, reducing trips to main memory and decreasing read misses. Meanwhile, write-nonallocate does not bring the block from main memory to cache and it writes to main memory, which will defeat the purpose of write-back because write-back aims at reducing writing to memory. As such, write-back and write-allocate work well together as they complement each other such that multiple writes within a block require only a single write to main memory and they use less memory bandwidth as a result.

2. The answer is 3ns, calculations are shown below.
Since, $T_{avgL2} = T_{hitL2} + (\%miss_{M2} * t_{missM2}) = 10 + 0 = 10ns$,
$T_{avg} = T_{hitL1} + (\%miss_{M1} * t_{avgL2}) = 2+10\%*10 = 3ns$.

3.
(a) The answer is 2^{49} virtual pages.
Each process sees 2^{64} bytes, which is equal to $2^{34} * 2^{30}$ B = 2^{34} GB virtual memory. Then, a 32KB page = $2^5 * 2^{10}$ B = 2^{15} B. So the number of virtual pages is 2^{64} B / 2^{15} B = 2^{49}.

(b) The answer is 2^{18} physical pages.
8GB physical memory is $2^3 * 2^{30}$ B = 2^{33} B, so the number of physical pages is 2^{33} B/ ($2^5 * 2^{10}$ B) = 2^{18}.

(c) The answer is 49-bit VPN and 18-bit PPN.
49-bit VPN because there are 2^{49} virtual pages, and 18-bit PPN because there are 2^{18} physical pages.

(d) The answer is 4 bytes (I did this by rounding the PTE size to the nearest power of 2, but according to TA Alex Boldt, the PTE size should be rounded to the nearest number of bytes, in which case, the answer is 3 bytes).
This Is because a PPN is 18-bit, so it needs 3B to hold it, which means the PTE should be 4B, the nearest power of 2.

(e) The answer is 2^{13}. (According to Boldt, for subsequent questions to 1(d), I can assume that the PTE size is 2 bytes for ease of computing, in which case, the answer, the number of PTEs that fit on a page, is 2^{15}B/2 B = 2^{14}.)

The page size is 32KB, which is equal to 2^{15}B, and the size of PTE is 4B. Therefore, the number of PTEs that fit on a page is $2^{15}/2^2 = 2^{13}$.

(f) The answer is 2^{12}.
The machine is 64-bit, which translates into 64-bit pointer, which means 8 bytes. So the number of pointers per page is page size/pointer size = 2^{15}B/2^3B = 2^{12}.

(g) The flat page table would have a size of 2^{51} bytes (again, here if I assume the PTE size is 2B, then the flat page table would have a size of 2^{49}*2B PTE = 2^{50} bytes.)
Given that there are 2^{49} virtual pages, with PTE size being 4B, the size of the flat page table is 2^{49} virtual pages * 4B PTE = 2^{51} bytes.

(h) Both the virtual page offset and physical page offset bits are 110 0001 1011 0100.
The virtual address is 25012_{10} = 0000…0000 0110 0001 1011 0100_2, with the last 15 bits being page offset and the other 49 0's being the VPN. Since the page is always 32KB, the page offset will not change. 15 bits are required for page offset. Therefore, Both the virtual page offset and physical page offset is 110 0001 1011 0100.

(i) No, a TLB miss does not always lead to a page fault. This is because a TLB miss leads to a page fault when the PTE requested is neither in the TLB nor in the page table. However, TLB miss happens when the requested PTE is not in the buffer, but it could still be in the page table/main memory. Therefore, it is possible that a page fault will not occur because I can find the PTE in the page table and put the entry into TLB.

4. My program works, and here is an explanation of the logic. The data structure I used in my code to represent a cache is an array of linked lists, with each array corresponding a set and each linked list corresponding to a frame/block. Please see comments in the code for details.

	Write-back + Write-allocate	Write-through + Write-nonallocate	Load
hit	Write to cache and NOT to main memory; update MRU, dirty bit; print	Write to cache & write to main memory; update MRU; print	Read data bits from cache starting from the index corresponding to the offset, of length offset's size + access size; print
miss	Read block from memory into cache, create a new frame and write to cache; check if LRU should be kicked; if so, check if the LRU is dirty; if	DO NOT bring block to cache; write to main memory; print	Read from memory, create a new frame and write to cache; update MRU; print

	so, update main memory; print		