Questions

1. Consider a (very old!) computer with a hard disk of only 1 GB, main memory of 16 MB, and a cache with 512 blocks of 8 words each. Suppose the hard disk and main memory are partitioned into pages, and suppose each page is 2 KB.

   (a) How many pages are there on the hard disk. How many pages are there in main memory?

   (b) How many bits are in the tag field of each entry of the cache?

2. In lecture 18, I considered two cache designs. Let's consider a third design as well in which there is just one byte at each cache entry (block).

   How does the total number of bits used in the cache vary as we change the block size (in the three cases) ? Assume that, in addition to the data, the cache contains a tag and dirty and valid bits in each entry (15=13+1+1 bits in that example).

3. Suppose you have some fixed number of bits in the cache. Using large blocks gives you better performance since you can take advantage of spatial locality i.e. nearby addresses tend to be accessed at nearby times. (You also get the benefit from the previous question, namely, you use fewer total bits.)

   You might assume you could always improve performance of the cache by making each block wider and having fewer blocks (entries). Is this true?

4. We talked about a dirty bit for the cache. What would a dirty bit be for the page table? How would it be used?

5. Assume a computer has a main memory with 64 MB and a data cache with 256 entries of 4 words each.

   (a) How many KB (kilobytes) of data are stored in the cache ?

   (b) How many bits are in the tag field of each line of the cache?

   (c) Suppose a program makes a sequence of five memory accesses in the order shown below. The numbers are physical byte addresses in main memory.

   ```
   0x1a24380
   0x211f0d0
   0x211f388
   0x212f12c
   0x1c8fad0
   ```

   What are the indices of the cache entries that are used in these memory accesses? Give you answer in hexadecimal.

   (d) What are the contents of the tag fields of these entries after all five memory accesses have been completed ? (Don't show intermediate steps.)

6. (a) Why do entries in the TLB have a ProcessID field, whereas entries in a page table do not?

   (b) TLB entries do not need to be invalidated (valid = 0) when there is a switch from one process to another. The valid bits only need to be set to 0 when a process finishes.

   What about the cache (data or instructions)? Do the valid bits for entries in the cache need to be set automatically to 0 each time the CPU is switched from one process to another?

   Hint (but be careful): the cache does not contain a processID field.

7. When the CPU writes a word into the cache and a write-back policy is used, the dirty bit for that cache entry must be turned on. The dirty bit indicates that the data at this entry is more recent than the data at the corresponding address in main memory.

   (a) Which MIPS instruction(s) can cause the dirty bits of the cache to be turned on?

   (b) Suppose a write-back scheme is used for the (data) cache. If a page fault occurs, then what (if anything) must be done to the cache, prior to handling the page fault?

Solutions

1. First, let's write some of the given values in terms of powers of 2.

   - 1 GB = 2^{30} bytes (hard disk).
   - 16 MB = 2^{24} bytes (main memory).
   - 2 KB = 2^{11} bytes (page)
   - 512 entries of 8 words = $2^9 * 2^3 * 2^2 = 2^{14}$ bytes (cache)

   (a) A page has 2^{11} bytes, so there are $2^{30-11} = 2^{19}$ pages on the hard disk and $2^{24-11} = 2^{13}$ pages in main memory.

   (b) There are 2^{24} bytes in main memory, hence 24 bits for the address of each byte in main memory. The cache only holds bytes that are in main memory. The lower 5 bits are used to access each byte in a block (2^3 words of 2^2 bytes each) and the next lower 9 bits are used to index a block (2^9 blocks). This leaves 24 - 14 = 10 bits for tag.

2. - case 1 (1 byte per block): $2^{17} * (15 + 8) \approx 300,000$
   - case 2 (4 bytes per block): $2^{15} * (15 + 32) \approx 150,000$
   - case 3 (4 words per block): $2^{13} * (15 + 128) \approx 120,000$

   The number of bits decreases from case 1 to case 3. The reason for the decrease is this: as you double the block size, you halve the number of cache lines and hence you halve the number of tags (and dirty and valid bits). The savings comes there.

3. Unfortunately not. Once the blocks in the cache become too wide and the number of the blocks becomes too few, the win that you get from spatial locality starts to become a loss in that there are few blocks that are represented in the cache. For example, suppose the cache had a single block of 16 KB, say 4K instructions. Now imagine a MIPS program that repeatedly calls a small function (a few instructions) but the code of the function (the callee) is more than 16 KB away in the program address space from the code that *calls* that function. If there were only a single block in the cache, then code for the caller and the code for the callee would not both fit into the (single block) cache at the same time. This means you would get misses. (If the the caller and callee were both small, then using many smaller blocks would be better, since both caller and callee could fit into the cache at the same time.) Moreover, having to fill such a huge block would be very expensive.

4. A dirty bit for the page table would indicate whether the page in main memory has been written to (from a write back from the cache), since that main memory page was first brought in from the hard disk. (Not all pages come originally from the hard disk though. Some are created as a program runs e.g. when the stack grows into a new page.) The dirty bit would be useful in case we needed to replace that page (to make room for another one, during a page swap. If the page dirty bit is 1, then we indeed need to write that page back to the disk. If the page dirty bit is 0, then the page was never been written to since it was brought into main memory (and hence that page wouldn't need to be written back).

5. (a) $256 \times 16 = 4096$ bytes, or 4 KB

   (b) 64 MB requires 26 bits of address space, which are broken into a 2 bit "byte offset", a 2 bit "word offset", a tag, and an 8 bit cache index (256 entries in the cache). The tag is thus 14 bits (26 - 2 - 2 - 8 = 14).

   (c) The byte offset and word offset use bits 0,1 and 2,3 respectively. Thus, the cache index is determined by the next 8 bits (4-11), or equivalently, by the hex digits 1 and 2 (where least significant is hex digit 0). The cache indices are 0x38, 0x0d, 0x38, 0x12, 0xad. Note that two of these (0x38) are the same and so there are only four different cache lines that are accessed.

   (d) The tag field is upper part of the address, beyond the cache index. (It is 4 hex digits.) The contents of the tag fields after all five accesses are:

```
index   tag
38      211f  (note that the first tag, 1a24, was overwritten)
0d      211f
12      212f
ad      1c8f
```

6. (a) There is only one TLB and it contains entries from many different page tables. So each entry needs a field to say which process it belongs to, i.e. which page table.

   Entries in a page table don't need a process ID since each process has its own page table, so the ProcessID would be the same for each entry in a single page table (so there's no need for the PID field).

   (b) No, the cache does not need to be emptied when the operating system switches to a new process. As long as the valid bit is on, the entry in the cache can be assumed to correspond to a block in main memory (the block being determined by the cache index and by the tag).

   You might think that the answer is yes, arguing that cache entries are useless to the new program since they hold information that is not needed. (Hence, the cache should be emptied.) There is some truth to this argument. However, if process A runs, then B, then A again, then there may be lots of cache entries for process A that are still valid the second time A is called. So, in general, there is no reason why the cache needs to be emptied.

   When a process finishes though, the cache entries for that process should be made invalid.

   So what about my tricky hint about the cache not containing a ProcessID field (PID)? The point of the PID is to disambuiguate different processes, which is necessary since processes have the same 32 bit address space. But there is no need to explicitly distinguish the physical address of different processes, since each process typically uses its own set of physical pages. (Moreover, one sometimes might want two processes to use the same physical page, e.g. if they are calling the same code from a library, or if they are sharing data.) This was mentioned in the lecture notes, but I repeat it here because you might only been watching the lectures themselves.

7. (a) `sw, sb, swc1`

   (b) Before the page is swapped out of main memory, the CPU needs to go through the cache and write back *all* blocks whose dirty bit is 1 *and* whose valid bit is 1 *and* whose physical address belongs to the page to be swapped out. It then needs to set the valid bit to 0 for those blocks.