Computer Systems
Fall 2017
Mid-term exam — Solutions

1. Questions: Answer each of the following questions:
   (a) What is the *translation lookaside buffer*?
   (b) What disadvantage to *FAT*-based file systems is addressed by *inode*-based file systems? How does the latter fix the problem?
   (c) Why do *garbage collectors* exhibit poor *spatial* and *temporal locality*? What effect does this poor locality have on the memory hierarchy?

   Answers:
   (a) The TLB is a small, high-speed hardware cache for page table entries. It is used by the MMU to quickly translation virtual addresses to physical ones.
   (b) When using a *FAT*, the *seek* operation must perform a linear search ($O(n)$, where n is the number of blocks in the file) through the linked list of blocks to find the correct one. The *inode* structure, by organizing the sequence of blocks for each file into a tree, makes it possible to map a specific seek location to a path in the tree to a specific block. Thus, the time to perform the seek is $O(1)$.[1]
   (c) A GC must visit *every live object on the heap*. When visiting each object, either to *copy* or *mark* it, the GC then follows outgoing pointers from that object as quickly as possible. Those pointers may lead anywhere in the heap. Thus, when the GC is running, there is poor sptial locality because the GC follows all available pointers, taking it away from the current region of the address space; and, there is poor temporal locality, since objects are not referenced repeatedly, but quickly used as jump-off points to other objects.
   The effect on the memory hierarchy is to *pollute* the hardware caches, filling them with data unlikely to be used soon by the program, and thus rendering them ineffective.

[1] OK, one could argue that the time is really not $O(1)$ because, as n grows, the depth of the tree—the number of levels of indirection—grows. Whatever function describes this growth, it's very, very slow. Since real *inode* implementations must ultimately fix the maximum depth (e.g., quad-indirect blocks), then there is a worst-case constant number of indirect levels to traverse.

2. QUESTIONS: Consider a system that uses 64-bit addresses and a 64 KB page size. To manage the virtual address spaces, assume a *4-level page table*.
   (a) How would a virtual address be divided into groups of bits to navigate this page table?
   (b) How much space would each allocated portion of this multi-level page table require?

ANSWERS:

(a) The bits of the 64-bit address would be divided like so, where bit `0` is the *least significant*, and bit `63` is the *most significant* of the address:
- `[63-52]`: The *level 0 index*, used to select an entry in the 0^{th}-level (top-level) page table of $2^{12} = 4096$ entries.
- `[51-40]`: The *level 1 index*, used to select an entry into the 1^{st}-level page table.
- `[39-28]`: The *level 3 index*, used to select an entry into the 2^{nd}-level page table
- `[27-16]`: The *level 3 index*, used to select a specific page-table entry at the 3^{rd}-level.
- `[15-0]`: The *offset* into the 64 KB (2^{16} byte) page. There are not used in traversing the page table.

(b) Because each of the four indices is 12 bits, each index chooses from 2^{12} entries per level. Since each entry is itself a 64-bit (8-byte) pointer, then each portion/block/chunk of the multi-level page table is 2^{12} *entries* $\times$ $2^3 \frac{bytes}{entry} = 2^{15}$ *bytes* $= 32$ *KB*. In other words, each chunk requires half of a page.

3. QUESTION: Consider a 16-entry hardware cache that is *4-way set associative* (i.e., there are 4 entries in each set). In this cache, each entry stores a 32-byte cache line (i.e., 4 words given 8 bytes/word).

   How would this hardware cache would use the 64-bit address sought to find the desired word and return it. (Assume that the address's data is, in fact, currently in the cache.)

   ANSWER: The 64-bit address would be divided and used like so:

   - `[63-7]`: The 57-bit *tag*, to be matched against the tags stored in each entry within the selected cache line.
   - `[6-5]`: The 2-bit *set selector*. That is, the hash function that maps an address to a set.
   - `[4-3]`: The 2-bit *word selector* within a cache line. These bits index into the cache line to select an 8-byte portion.
   - `[2-0]`: The 3-bit *byte selector* within the given word. Whole words are returned by the cache, so these bytes are used later by the processor, if needed, to select a specific byte.[2]

[2] I did not expect you to know this detail about selecting bytes within words for only certain operations. I had no expectation that you would mention it.

4. QUESTION: How would you change our Project-1 memory allocator so that it would *coalesce* free blocks? Describe/write/show changes in any data structures, as well as alterations to use those structures by `malloc()` and `free()`.

   ANSWER: Assume a call to `free(ptr)`, where `ptr` provides the base of the programmer-usable block of heap memory. From this block, we must find whether either or both of the adjacent blocks in the address space are free. If so, this block must be coalesced with it/them. Given the original structure, from `ptr`, we can determine the location of the following:

   - At `ptr - sizeof(size_t)`: The *header*, which contains the block's size (not including the header itself.
   - At `ptr + blocksize`: The header of the *next block in the address space*, which gives us the next block's location and, from its header, its size.

   While this information is useful, we would lack the following:

   - The *location of the previous block in the address space.*
   - Whether the *next block is in use or free.*
   - Whether the *previous block is in use or free.*

   In order to find this information, we augment each allocated block with a *footer*—a reserved space, like the header, that appears immediately after the programmer-usable portion of the allocated block.[3] That footer contains a pointer to the block's header.
   Additionally, we will change the encoding of the size in the header. Specifically, because a single block can never take up more than half of the address space, we can use the most significant bit as a *flag* to determine whether the block is *allocated* (bit set to 1) or *free* (bit set to 0).
   Thus, `free()` will be able to:

   - *Find the next block* in the address space, as described above.
   - *Find the previous block* in the address space, by calculating `footer = header - sizeof(void*)`, and following that footer's pointer to reach the previous block's header.
   - *Determine whether the next block is free* by examining the *allocated/free* bit in the next block's header.
   - *Determine whether the previous block is free* by examining the *allocated/free* bit in the previous block's header.

[3] This approach is one of many possible solutions to finding the adjacent blocks. I present it as one example, not as the definitive answer.

First, `free()` must clear the *allocated/free* bit in the current block's header.
If `free()` can coalesce the current block with the next block, then it must find the next block in the free list and remove it, add the next block's size to the current block's size, update the next's block's footer to refer to the current block, and then insert the current block into the free list.
If `free()` can coalesce the current block with the previous block, then it must update its own footer to point to the previous block, and then add its size to the previous block's size. Notice that the current block has not yet been inserted in the free list, and the previous block is already in it, so nothing needs to be removed from that list.
The `malloc()` function needs only to create and assign the footer to each block it creates (when pointer bumping), and to set the *allocated/free* bit on each allocated block's header.