CS 351: Systems Programming

Fall 2019 Final Exam

Instructions:

- This exam is closed-book, closed-notes. Calculators are not permitted.
- For numbered, multiple-choice questions, fill your answer in the corresponding row on the "bubble" sheet.
- For problems that require a written solution (labeled with the prefix "WP"), write your answer in the space provided on the written solution sheet. Please write legibly and clearly indicate your final answer.
- Turn in the exam question packet, bubble sheet, and written solution sheet separately.
- Good luck!

Concepts. (20 points):

Please choose the *single best* answer to each of the following questions.

1. Which access pattern demonstrates good *spatial* locality?

   (a) stepping through elements in a one-dimensional array with a large stride
   (b) successively accessing elements in neighboring columns within a two-dimensional array
   (c) successively accessing elements in neighboring rows within a two-dimensional array
   (d) repeatedly accessing a global variable

2. Which approach to software-level cache optimization is appropriate when the working set of data is too large to fit in the cache (i.e., resulting in *capacity misses*)?

   (a) blocking
   (b) improvements to spatial locality
   (c) improvements to temporal locality
   (d) better leveraging registers

3. Which aspect of memory is directly managed by user-level code, and so is best optimized by the compiler?

   (a) mapping virtual to physical pages
   (b) translating virtual to physical addresses
   (c) mapping variables to registers
   (d) deciding what values to store in the cache

4. What is the primary reason for incorporating the translation lookaside buffer (TLB)?

   (a) to improve data cache hit rates
   (b) to offload the responsibility for translating virtual addresses from the MMU to the kernel
   (c) to reduce the amount of internal fragmentation arising due to paging
   (d) to avoid accessing the page table in main memory

5. What action is likely to closely follow a *page fault*?

   (a) the MMU triggers a segmentation fault
   (b) the kernel loads a page into memory and updates the page table
   (c) the segmentation registers are updated to reflect updating mappings
   (d) the page table is defragmented to make room for a new page table entry

6. In lower-level caches (i.e., those further away from the CPU), it is likely that:

   (a) cache complexity is reduced, allowing for higher miss rates but faster hit times
   (b) cache complexity is increased, allowing for higher hit rates but slower hit times
   (c) block sizes are reduced, in order to allow for improved cache utilization
   (d) virtual addresses are used for lookup instead of physical addresses

7. What is a significant downside to segmentation, as a virtual memory implementation, that is addressed by paging?

   (a) internal fragmentation due to segment overhead
   (b) the inability to implement memory protection
   (c) external fragmentation due to inconsistent segment sizes
   (d) inefficient segmented-to-linear address translation

8. The primary benefit to using an *explicit free list* over an *implicit list* is that:

   (a) only free blocks need be searched to find a fit
   (b) it makes it possible to perform `free` in O(1) time
   (c) it makes it possible to perform immediate coalescing
   (d) best fit becomes a $O(\log(N))$ operation, where N is the number of free blocks

9. In C, the lack of runtime type information and the presence of pointers means that a garbage collection facility would need to be *conservative*, which means ...

   (a) it might not free up all unreachable (allocated) memory
   (b) it might accidentally mark free blocks as allocated
   (c) it would be unable to free blocks found in a circular data structure
   (d) it would need to assume all currently allocated blocks are reachable

10. Which of the following operations would not typically be permitted on a *character* device (as opposed to a *block* device)?

    (a) reading
    (b) writing
    (c) seeking / random access
    (d) buffering

Cache Hit/Miss Rates (12 points):

Consider the following function, which takes the dimension `D` of two square arrays to process, and pointers to the arrays:

```
void procArr(int D, int A[D][D], int B[D][D]) {
  int i, j, res;
  res = 0;
  for (i=0; i<D; i++) {
    for (j=0; j<D; j++) {
      res += A[j][i] + B[i][j];
    }
  }
}
```

For the following questions, only consider memory accesses to the arrays in `procArr` — i.e., assume that all other variables are mapped to registers — and that `int`s are 4 bytes wide.

WP1 (a). Given a direct-mapped data cache with 8-byte blocks and 4 total lines, indicate which accesses in the argument arrays A and B with dimensions `D=2` are hits, misses, and miss-evictions by marking each cell on the worksheet with either `H` (hit), `M` (miss), or `E` (miss-eviction).

Arrays A and B begin at memory addresses `0x601000` and `0x602008`, respectively. Assume that the cache starts out empty.

WP1 (b). Repeat part (a) using a direct-mapped cache with 16-byte blocks and 4 total lines, and array dimensions `D=3`.

This time, assume arrays A and B begin at memory addresses `0x601000` and `0x602020`, respectively.

VM Structures (8 points):

Consider a system that uses 40-bit virtual addresses and has 8GB (2^{33} bytes) of physical DRAM installed. Paging is used as to implement virtual memory, and the page size is 512KB (2^{19} bytes). Memory is byte-addressed, and the word size is 8 bytes.

11. What is the total number of pages in a process's virtual address space?

    (a) 2^{19} pages
    (b) 2^{21} pages
    (c) 2^{28} pages
    (d) 2^{33} pages

12. Given single-level page tables with word-sized page table entries, how large is the size of each per-process page table?

    (a) 2^{19} bytes (512KB)
    (b) 2^{20} bytes (1MB)
    (c) 2^{24} bytes (16MB)
    (d) 2^{28} bytes (256MB)

13. Given a two-level page table setup, where the first-level page directory contains 1K (2^{10}) word-sized pointers to second-level page tables, what is the minimum amount of physical memory required for a process's paging structures?

    (a) 2^{10} bytes (1KB)
    (b) 2^{13} bytes (8KB)
    (c) 2^{16} bytes (64KB)
    (d) 2^{20} bytes (1MB)

14. Consider a process with 128MB of data stored in its heap region. If each page in said region contains, on average, 256KB of data (payload), approximately how much heap memory is being wasted due to internal fragmentation?

    (a) 1MB
    (b) 32MB
    (c) 64MB
    (d) 128MB

Address Translation (10 points):

The questions on the next page are based on the cache and memory setup described below:

- Memory is byte addressable
- Virtual addresses are 28 bits wide
- Physical addresses are 22 bits wide
- The page size is 8K bytes (2^{13} bytes)
- The TLB is 2-way set associative, with 8 total lines
- The cache is 4-way set associative, with 8-byte blocks and 8 total lines

The current TLB, cache, and partial page table contents are given here:

TLB			
Index	Tag	Valid	PPN
0	0BA3	0	0E6
	1A2D	0	1CA
1	03B2	1	0AA
	0300	0	1B1
2	1E10	0	014
	061D	1	1C0
3	0F5F	1	08C
	13A0	1	07C

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	198	1	0A	1DE	1
01	10F	0	0B	1EB	1
02	0B1	1	0C	046	1
03	054	1	0D	18E	1
04	0EC	1	0E	1AE	0
05	141	1	0F	10C	0
06	0BE	1	10	178	0
07	165	0	11	01D	1
08	09B	1	12	1AD	0
09	0CB	0	13	1F7	1

Cache										
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0	3C33B	0	93	47	78	C6	41	80	18	A9
	08C15	1	0E	EE	DF	E5	C7	5B	4B	84
	077AD	1	0B	5A	96	AA	FD	EC	D2	2A
	06DF8	0	96	17	11	90	CA	4E	EA	05
1	24AAA	1	6D	7C	5C	14	9F	50	84	AB
	08815	0	4F	3B	4E	34	FB	39	B7	08
	1145D	0	1B	4D	21	25	ED	1D	B7	FD
	0F849	1	46	03	0D	DB	B5	76	1A	9C

And for your convenience, here's a decimal ↔ hex ↔ binary lookup table:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

WP2 (a). Using the diagrams on the worksheet, indicate where the following fields are to be found in the virtual and physical addresses (if they exist).

- Virtual address fields:

  VPO : The virtual page offset
  VPN : The virtual page number
  TLBI : The TLB index
  TLBT : The TLB tag

- Physical address fields:

  PPO : The physical page offset
  PPN : The physical page number
  CO : The cache block offset
  CI : The cache set index
  CT : The cache tag

WP2 (b). Translate the virtual address **0x9D0649E**.

WP2 (c). Translate the virtual address **0x0018152**.

For parts **(b)** and **(c)**, fill in all possible fields in the provided diagrams and tables. If a page fault occurs, leave the physical-address related fields empty.

DMA Metrics (8 points):

Given the following sequence of calls to `malloc` and `free`:

```
void *p1, *p2, *p3, *p4, *p5, *p6;
p1 = malloc(512);
free(p1);
p2 = malloc(512);
free(p2);
p3 = malloc(512);
p4 = malloc(512);
free(p3);
p6 = malloc(512);
```

15. What is the maximum aggregate payload?

    (a) 512 bytes (b) 1024 bytes

    (c) 2048 bytes (d) 3072 bytes

16. Given a final heap size of 4096 bytes, what is the peak memory utilization?

    (a) 100% (b) 50%

    (c) 25% (d) 15%

Next, consider an implicit-list based dynamic memory allocator with the following properties:

- blocks and payloads are word-aligned, with a word size of 8 bytes
- each block has a header and footer word used to store the block size & allocated bit
- the amount of space available in any given block for payload must be at least 50% of the block size.

17. What is the minimum block size?

    (a) 16 bytes (b) 24 bytes

    (c) 32 bytes (d) 40 bytes

18. Given a payload request of 100 bytes, what is the minimum size of a block that can satisfy the request *and* be split into an allocated and free block?

    (a) 116 bytes (b) 120 bytes

    (c) 144 bytes (d) 152 bytes

Implicit List DMA (12 points):

For the following two written problems, consider an implicit-list based allocator and memory architecture with the following properties:

- words are 4-bytes in width, and blocks and payload are word-aligned
- every block has word-sized boundary tags (each corresponding to a `size_t`) that store the size of the block and an allocated bit
- immediate coalescing is performed, and blocks are split during allocation whenever doing so results in two blocks with non-zero payload
- when splitting, the "lower" part of a block is allocated
- a first-fit search is used during allocation

WP3. (6 points) Starting with the leftmost heap depicted on the worksheet, and moving from left to right, show the updated header/footer values on the heap after each successive call to `malloc`, `free`, and `realloc`.

WP4. (6 points) Implement the function `void *try_split(void *bp, size_t size)` which is passed a pointer to the start of a free block and a request (payload) size. The free block is guaranteed to be large enough to accommodate the request. If possible, the function will split the block into two separate free blocks, the first of which can accommodate the request with minimum internal fragmentation.

If the split is successful the function will return a pointer to the second block, otherwise it will return `NULL`.

The following definitions are given to help work with aligned values:

```
#define ALIGNMENT 4
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~3L)
#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
```

Your implementation should not make use of any external macros nor functions. (Sanity check: our implementation is 10 lines of code.)