CSE351 Spring 2010 – Final Exam (9 June 2010)

Please read through the entire examination first! We designed this exam so that it can be completed in 100 minutes and, hopefully, this estimate will prove to be reasonable.

There are 5 problems for a total of 200 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided. If you need more space (you shouldn't), you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. Do NOT use any other paper to hand in your answers. If you have difficulty with part of a problem, move on to the next one. They are mostly independent of each other.

The exam is CLOSED book and CLOSED notes. Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Name: _____

ID#: _____

Problem	Max Score	Score
1	40	
2	35	
3	40	
4	35	
5	50	
TOTAL	**200**	

1. Pointers, Addresses, and Values (40 points)

A memory has the following contents (in our typical little-endian format):

Variable	Address	Bytes
A	0x08000100	04 01 00 08
B	0x08000104	14 01 00 08
C	0x08000108	ff ff ff fe
D	0x0800010c	ff ff ff ff
E	0x08000110	00 00 00 00
fp	0x08000114	01 00 00 00
G	0x08000118	33 35 31 00
ptr	0x0800011c	00 00 00 00

Given the following declarations:

```
int *A, *B, *ptr;
int C, D, E, G;
typedef struct foo { int i[2]; char * aString; } foo;
foo *fp;
```

Fill in columns for the address (in hex) that is changed in each statement and the value (in hex) to which it is changed. Assume that the statements are executed in sequences and changes in one line propagate to following lines.

C Statement	Address (hex)	Value (hex)
`ptr = &A;`	*0x0800011c*	*0x08000100*
`ptr = A;`	*0x0800011c*	*0x08000104*
`ptr = (int *) *A;`	*0x0800011c*	*0x08000114*
`ptr = (int *) *ptr;`	*0x0800011c*	*0x00000001*
`fp = (foo *) &C;`	*0x08000114*	*0x08000108*
`fp->aString = (char *) &G;`	*0x08000110*	*0x08000118*
`fp->i[1] = (int) *fp->aString;`	*0x0800010c*	*0x00000033*
`fp = (foo *)(((char *)fp) + 12);`	*0x08000114*	*0x08000114*

2. Caches (35 points)

You are trapped inside of a Programming Testing Facility, where a disembodied robotic voice named MemCaFE (Memory Cache and Final Exam) tells you that you must solve a series of cache related puzzles in order to be released.

Your series of puzzles relate to the same cache, which is initially empty (cold) at the beginning of each part, with all valid bits set to zero. The puzzles consist of transcripts of a guessing game played by previous prisoners. MemCaFE plays a guessing game with with its prisoners where they name an address, and MemCaFE responds with whether it is a HIT or a MISS, like a game of Battleship. If it is a MISS, MemCaFE loads the correct value into the cache from main memory for future queries. It resets the cache so that it starts cold at the beginning of each puzzle.

a) (10 points) Here is a transcript of the first puzzle with MemCaFE:

You	MemCaFE
0x000	MISS
0x001	HIT
0x002	HIT
0x003	HIT
0x004	HIT
0x005	HIT
0x006	HIT
0x007	HIT
0x008	MISS

MemCaFE asks: What is the block size (B) of this cache (in bytes)?

The first access is a miss in a cold cache. The next 7 sequential accesses are all hits implying that they were all brought into the cache with the first miss. Therefore, the size of a block must be 8 bytes. B = 8 bytes

b) (10 points) Here is an abbreviated transcript (you should be able to see the pattern) of your second puzzle with MemCaFE:

You	MemCaFE
0x000	MISS
0x008	MISS
0x000	HIT
0x010	MISS
0x000	HIT
0x018	MISS
0x000	HIT
0x020	MISS
0x000	HIT
(…time passes…)	
0x100	MISS
0x000	MISS

MemCaFE asks: What is the total size (C) of this cache (in bytes)?

*The access pattern alternates with accesses to 0x000 and addresses in 8-byte increments (the size of a block). Only after doing 32 increments does 0x000 get evicted from the cache and thereby causes a miss on the next access. Therefore the total size of the cache must be 32*8 or 256 bytes. C = 256 bytes*

c) (10 points) Here is the transcript of your third puzzle with MemCaFE:

You	MemCaFE
0x000	MISS
0x100	MISS
0x000	HIT
0x200	MISS
0x000	HIT
0x300	MISS
0x000	HIT
0x400	MISS
0x000	MISS

MemCaFE asks: What is the associativity (E) of this cache (number of lines per set)?

In this case, we increment our accesses by the size of the cache, namely, 0x100 or 256 bytes. After 4 increments 0x000 leads to a miss again, having been evicted from the cache. Therefore, there are 4 lines in each set as it took 4 additional accesses to get 0x000 evicted. E = 4 lines per set

d) (5 points) Finally, what is the number of bits needed to encode the set index in this cache?

C = 256 bytes
B = 8 bytes
E = 4 lines

*There are 32 bytes per set (B * E = 8 * 4 = 32), and 8 sets in the cache (S = C/BE; log S = log C – log B – log E = 8 – 3 – 2 = 3). Therefore, the set index requires 3 bits.*

3. Data Structure Representations (40 points)

a) (10 points) One day, while reading through old programs, you discover the following string in the data section of a program:

```
0x0000: 6d 6f 64 69 66 79 20 74  68 65 20 70 68 61 73 65
0x0010: 20 76 61 72 69 61 6e 63  65 00
```

When you convert the values here to ASCII characters, you see:

```
modify the phase variance.
```

where the ending period is a place holder for 0x00. (We use a "period" because 0x00 is unprintable.)

Write a function "string_length" (pseudo-C is fine) to compute the length of this string.

The function scans the string looking for the null character (0x00) while incrementing length for each character scanned before 0x00.

```
int string_length (char * string) {
      int length = 0;
      char * ptr;
      for (ptr = string; *ptr != 0x00; ptr++)
            length++;
      return (length);
}
```

b) (5 points) While reading through another old program, you discover a similar-looking string:

```
0x0000: 19 6d 6f 64 69 66 79 20  74 68 65 20 70 68 61 73
0x0010: 65 20 76 61 72 69 61 6e  63 65
```

When you convert these values to ASCII characters, you see:

```
.modify the phase variance
```

where the leading period is a place holder for 0x19. (We use a "period" because 0x19 is unprintable.)

Clearly these two strings contain the same data. What is the difference between the way this string is represented and the way the previous string is represented? Which is closest to the way that Java represents strings?

The string in part (a) uses a null character to mark the end of the string. The string in part (b) uses the first byte to represent the length of the string. Java strings are closest to the representation of part (b) but with an integer – 4 bytes – representing the length of the string and characters using 2 bytes each as they are represented in Unicode.

c) (5 points) Write a new function "string_length" (pseudo-C is fine) to compute the length of the string in part (b).

The program simply recovers the length of the string from the first character and casts it as an integer.

```
int string_length (char * string) {
      return ((int) *string);
}
```

d) (10 points) Below is the definition of a function safe_charAt:

```
char * safe_charAt(char * c, int i)
```

This function takes two arguments:

- c, the address of the start of a string
- i, an index into the string

and returns the address of the i^{th} character in the string. The first character in the string is the 0^{th} character. However, if i is out of bounds, the function returns 0.

Of the two formats presented in parts (a) and (b), which format would produce the fastest implementation of safe_charAt? Why?

The representation of part (a) would be slower because we have to scan the entire string to check that i is less than the length of the string. In fact, on average its run-time will be linear in the size of the string (or the value of the index, i). With the representation of part (b), we can check the bounds immediately as the length is the first byte in the string. It has a constant run-time.

Write the body (pseudo-C is fine) of the safe_charAt function given the string_length function as defined above.

```
char * safe_charAt(char * c, int i) {
   if ( (i < 0) || (string_length(c) >= i) ) return (0);
   else return( c + 1 + i);
}
```

e) (10 points) You wrote the following program:

```
struct stringy {
       int a[2];
       char c[10];
};

struct stringy s;
char * x;
s.a[0] = 0;
s.a[1] = 1;
s.a[2] = 0x05af38dc;
s.a[3] = 0xcafe666a;
x = safe_charAt( s.c, 3 );
```

What is the value of *x if:

- the string format from part (a) were used:

  The value of *x *is indeterminate because we may have overwritten the null character in the string* `c` *when writing to* `a[2]` *and* `a[3]` *overstepped the bounds of array* `a`*. The loop to find the length of the string* `c` *may never terminate.*

- the string format from part (b) were used:

  The value of *x *would be* `0xca` *because* `i = 3` *and that is less than the new length of the string (0x05) – the first byte of the value written to* `s.a[3]` *or by its more correct alias* `s.c[4]`.

4. Virtual Memory (35 points)

Our system has implements virtual memory with a translation look-aside buffer (TLB – 16 entries, 2-bytes lines, 4-way set associative), page table (PT – 1024 page entries, only the first 16 shown below), and memory cache (16 entries, 4-byte lines, direct-mapped). The initial contents are shown in the tables below. The virtual address is 16 bits (4 hex digits) while the physical address is 12 bits (3 hex digits); the page size is 64 bytes.

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	33	1	08	–	0	06	–	0	03	11	1
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page Table

VPN	PPN	Valid
000	28	1
001	–	0
002	09	1
003	02	1
004	–	0
005	16	1
006	–	0
007	–	0

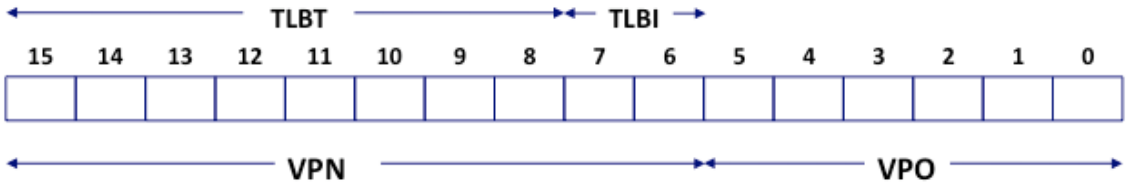
VPN	PPN	Valid
008	13	1
009	17	1
00A	33	1
00B	–	0
00C	–	0
00D	2D	1
00E	11	1
00F	0D	1

Cache

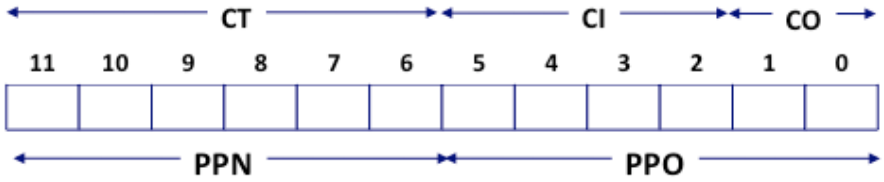
Index	Tag	Valid	B0	B1	B2	B3
0	28	1	99	1F	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Index	Tag	Valid	B0	B1	B2	B3
8	11	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

a) (5 points) Outline the bits corresponding to each of the components of the virtual address, namely, the virtual page number (VPN), the virtual page offset (VPO), the TLB set index (TLBI), and the TLB tag value (TLBT).



b) (5 points) Outline the bits corresponding to each of the components of the physical address, namely, the physical page number (PPN), the physical page offset (PPO), the cache set index (CI), the cache tag value (CT), and the cache byte offset (CO).



c) (5 points) What are the advantages of a set-associated TLB cache (with E=4) as opposed to a direct-mapped TLB cache?

The TLB is set-associative to ensure that there is a lower miss rate for page table entries that have already been accessed. If this small cache were direct-mapped, page table entries would be more likely to be evicted by an address with the same set index. The miss rate is lowered by exploiting set-associativity and the ability to cache a page table entry in one of four entries.

d) (20 points) Determine the returned values for the following sequence of accesses and specify whether a TLB miss, page fault, and/or cache miss occurred. In some cases, it may not be possible to determine what value is accessed or whether there is a cache miss or not. For these cases, simply write ND (for Not Determinable).

Virtual Address	Physical Address	Value	TLB Miss?	Page Fault?	Cache Miss?
0x03A2	*0x462*	*0x51*	*N*	*N*	*N*
0x02C1	*ND*	*ND*	*Y*	*Y*	*ND*
0x02B4	*0xCF4*	*ND*	*N*	*N*	*Y*
0x0000	*0xA00*	*0x99*	*Y*	*N*	*N*

5. Memory Reallocation (50 points)

In addition to malloc and free, the C standard library includes realloc, with the following signature and specification:

```
void * realloc(void * ptr, size_t size)
```

- The realloc() function first tries to change the size of the allocation pointed to by ptr to size, and return ptr, thereby shrinking the block in place.
- If the requested size is larger than the block then there is not enough room to enlarge the memory allocation pointed to by ptr, realloc() allocates a new block and copies as much of the old data pointed to by ptr as will fit in the new allocation, frees the old allocation, and returns a pointer to the newly allocated memory.
- If ptr is NULL, realloc() is identical to a call to malloc() for size bytes.
- If size is zero and ptr is not NULL, a new, minimum-sized object is allocated and the original object is freed.

a) (30 points) Write C-like pseudo-code for a simple realloc, using the explicit free list from lab 7. Be sure to adhere to the specification above. Your implementation does not need to be complicated or optimal (for example, do **NOT** worry about splitting a larger block into a smaller one – it is ok to waste the space). Your pseudo-code should be close to C (with comments as needed), but don't worry about syntax details if you can't remember them exactly.

To copy memory from one place to another, you may use the memcpy function:

```
void memcpy(void * new_ptr, void * old_ptr, size_t copy_size);
```

This function copies copy_size bytes from memory starting at old_ptr to memory starting at new_ptr. Here is the relevant code from our explicit free list implementation for reference. You may call any of the functions below.

```
/* Alignment of blocks returned by mm_malloc. */
#define ALIGNMENT 8

/* Size of a word on this architecture. */
#define WORD_SIZE sizeof(void*)

/* Minimum implementation-internal block size (to account for size
   header, next ptr, prev ptr, and boundary tag).  This is NOT the
   minimum size of an object that may be returned to the caller of
   mm_malloc. */
#define MIN_BLOCK_SIZE 4*WORD_SIZE
```

```
/* Macros for pointer arithmetic to keep other code cleaner. Casting
   to a char* has the effect that pointer arithmetic happens at the
   byte granularity (i.e. POINTER_ADD(0x1, 1) would be 0x2). (By
   default, incrementing a pointer in C has the effect of incrementing
   it by the size of the type to which it points (e.g. BlockInfo).) */
#define POINTER_ADD(p,x) ((char*)(p) + (x))
#define POINTER_SUB(p,x) ((char*)(p) - (x))

/* A BlockInfo contains information about a block, including the size
   and usage tags, as well as pointers to the next and previous blocks
   in the free list.

   Note that the next and prev pointers are only needed when the block
   is free. To achieve better utilization, mm_malloc would use the
   space for next and prev as part of the space it returns.

   +-------------+
   | sizeAndTags | <- BlockInfo pointers in free list point here
   +-------------+
   |    next     | <- Pointers returned by mm_malloc point here
   +-------------+
   |    prev     |
   +-------------+
   |    space    |
   |     ...     |
*/

struct BlockInfo {
/* Size of the block (log(ALIGNMENT) high bits) and tags for whether
   the block and its predecessor in memory are in use. */
int sizeAndTags;
   // Pointer to the next block in the free list.
struct BlockInfo* next;
   // Pointer to the previous block in the free list.
struct BlockInfo* prev;
};
typedef struct BlockInfo BlockInfo;

/* SIZE(x) selects just the higher bits of x to ensure that it is
   properly aligned. Additionally, the low bits of the sizeAndTags
   member are used to tag a block as free/used, etc. */
#define SIZE(x) ((x) & ~(ALIGNMENT - 1))

/* TAG_USED bit mask used in sizeAndTags to mark a block as used. */
#define TAG_USED 1

/* TAG_PRECEDING_USED is the bit mask used in sizeAndTags to indicate
   that the block preceding it in memory is used. (used in turn for
   coalescing) */
#define TAG_PRECEDING_USED 2

void insertFreeBlock(BlockInfo* freeBlock);
void removeFreeBlock(BlockInfo* freeBlock);
void coalesceFreeBlock(BlockInfo* oldBlock);
void * mm_malloc (size_t size);
void mm_free (void *ptr);
```

Implement realloc here (the cases for ptr=NULL and size=0 are already done):

```
void * mm_realloc(void * ptr, size_t size) {
  // Standard code for rounding to the right 8-byte-aligned size...
  // Add one word for the initial size header.
  int reqSize = size + WORD_SIZE;
  if (reqSize <= MIN_BLOCK_SIZE) {
      // Minimum size... one for next, one for prev, one for
      // boundary tag, one for the size header.
      reqSize = MIN_BLOCK_SIZE;
  } else {
      // Round up for correct alignment
      reqSize = ALIGNMENT * ((size + ALIGNMENT - 1) / ALIGNMENT);
  }
  BlockInfo * oldBlock;
  int currentTotalSize, currentPayloadSize;
  void * newPtr;
  if (ptr == NULL) {
     return mm_malloc(size);
  }
  if (size == 0) {
     mm_free(ptr);
     return mm_malloc(WORD_SIZE);
  }

// PLACE YOUR CODE HERE ( about 8-16 lines of pseudo-C )

  // Find the header of the block being reallocated
  oldBlock = (BlockInfo *) POINTER_SUB(ptr, WORD_SIZE);

  // Get its size and figure out the payload size
  currentTotalSize = SIZE(oldBlock->size);
  currentPayloadSize = currentTotalSize - WORD_SIZE;

  // If the requested size is the same or smaller than the old size
  // use as is.
  if (reqSize <= currentTotalSize){
     return ptr;
  }

  // If more space required, allocate new block, copy, and free old.
  newPtr = mm_malloc(size);
  memcpy(newPtr, ptr, currentPayloadSize);
  mm_free(ptr);
  return newPtr;
}
```

b) (10 points) Describe when it is safe to call realloc without updating any other pointers in the program. HINT: consider pointers that may exist in the program.

It is safe when (1) no pointers point to addresses in the block being reallocated or (2) when the reallocation requests a smaller size than the block's current size. In the latter case, the new reallocated block is guaranteed to fit in its current location, so realloc will not need to move it thereby not requiring any pointers to be updated.

c) (10 points) When a reallocation requests less space than the size of the current block, our implementation does not move the block but simply changes its size. What problem might this cause as the program continues to run? How might we fix this problem? How does your solution affect performance?

Since we shrink a block in place, we leave a fragment behind. Although this could be added to the free list, it causes fragmentation. Eventually, we may have enough free memory but not enough of it contiguous to grant an allocation request. A solution would be to do a new allocation whenever the size of the block changes so that we can find a best fit that minimizes fragmentation. The cost is lower performance as now all blocks that change size need to be moved.