CSE351 Autumn 2013 – Final Exam (11 Dec 2013)

Please read through the entire examination first! We designed this exam so that it can be completed in approximately 90 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 and at every part of every problem. 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 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: ***Sample Solution***

ID#: _____

Problem	Max Score	Score
1 (Appetizer)	30	***30***
2 (Soup/Salad)	30	***30***
3 (Entrée 1)	50	***50***
4 (Entrée 2)	70	***70***
5 (Dessert Tray)	20	***20***
TOTAL	**200**	***200***

1. Appetizer (True/False Answers) – 30pts total (2pts each)

A. A 4-byte integer can be moved into a 32-bit register using a movw instruction.
☐ True ☑ False

B. On a 64-bit architecture, casting a C integer to a double does not lose precision.
☑ True ☐ False

C. Shifting an int by 3 bits to the left (<< 3) is the same as multiplying it by 8.
☑ True ☐ False

D. In C, endianess makes a difference in how character strings (char*) are stored.
☐ True ☑ False

E. In C, storing multi-dimensional arrays in row major order makes it possible for pointer arithmetic to determine the address of an array element.
☑ True ☐ False

F. A struct can't have internal fragmentation if the elements of the struct are ordered from largest to smallest.
☑ True ☐ False

G. An instruction cache takes advantage of only spatial locality.
☐ True ☑ False

H. Caches are part of the instruction set architecture (ISA) of a computer.
☐ True ☑ False

I. Caches make computers slower by getting between the CPU and memory.
☐ True ☑ False

J. On a 64-bit architecture, if a cache block is 32 bytes, and there are 256 sets in the cache, the tag will be 53 bits.
☐ True ☑ False

K. A process's instructions are typically in a read-only segment of memory.
☑ True ☐ False

L. A shared library can be accessed from multiple virtual address spaces, but with only one copy in physical memory.
☑ True ☐ False

M. Virtual memory allows programs to act as if there is more physical memory than there actually exists on the computer.
☑ True ☐ False

N. Two running instances of the same process share the same memory address space.
☐ True ☑ False

O. Java generally has better performance than C.
☐ True ☑ False

2. Soup or Salad (Stacks and Pointers) – 30 pts total (10/A, 10/B, 10/C)

You are running a program on a 64-bit architecture, that uses stack frames that include saved old `%rbp` registers but passes arguments in registers. Assume integers are 4 bytes.

The program includes the definition for a data_structure type:

```
typedef struct data_struct {
  int *i;
  int a;
} data_struct;
```

as well as the definition of a print_structure function:

```
void print_struct(data_struct *d) {
  printf("%p\n", d);
  printf("0x%x\n", *(d->i + d->a));
}
```

This is a small snippet of code corresponding to `foo`, which has just been called:

```
int foo() {
  data_struct x;
  int n = 5;
  x.i = &n;
  x.a = <???>;
  print_struct(&x);
  . . .
}
```

The stack at this point of the execution of the program is shown below in 4-byte blocks (note that the stack is shown as is tradition, with higher addresses above lower ones):

```
0x7fffffffffffa040:  0x00000021
0x7fffffffffffa03c:  0x00000004
0x7fffffffffffa038:  0x00000000
0x7fffffffffffa034:  0x00402053
0x7fffffffffffa030:  0x7fffffff
0x7fffffffffffa02c:  0xffffa048
0x7fffffffffffa028:  0x00000009
0x7fffffffffffa024:  0x7fffffff
0x7fffffffffffa020:  0xffffa01c
0x7fffffffffffa01c:  0x00000005
0x7fffffffffffa018:  0x000024b7
0x7fffffffffffa014:  0x7fffffff
0x7fffffffffffa010:  0xffffa008
0x7fffffffffffa00c:  0x00000001
```

A. What is stored in the stack at the 8-bytes starting at location `0x7fffffffffa034`?

The return address to the instruction after the call to `foo` in the function that called `foo`.

B. What value was assigned to `x.a` in the function `foo`?

0x9

C. What will the call to `print_struct` output?
(Note: the "`%p`" format specifier prints the value of a pointer in hex format.)

0x7fff…ffa020
0x21

3. Entrée 1 (Heap Data Structures) – 50pts total (5/A, 5/B, 5/C, 5/D, 30/E)

We have a system with the following properties:

- a virtual address of 16 bits (4 hex digits),
- a physical address of 12 bits (3 hex digits),
- pages that are 256 bytes,
- a corresponding page table with 256 entries,
- a TLB with 16 entries that is 4-way set associative, and
- a cache of 128 bytes with 8-byte blocks arranged as 2-way set associative.

The current contents of the TLB, Page Table, and Cache are shown below:

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	07	0	1	06	-	0	3F	3	1
1	05	3	1	0A	-	0	00	B	1	01	F	1
2	07	-	0	0B	-	0	0F	2	1	2B	-	0
3	01	C	1	0C	1	1	02	0	1	1A	1	1

Page Table (only first 16 of the 256 PTEs are shown)

VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid
00	3	1	04	-	0	08	3	1	0C	F	1
01	6	1	05	-	0	09	-	0	0D	-	0
02	3	1	06	D	1	0A	1	1	0E	6	1
03	3	1	07	-	0	0B	-	0	0F	A	1

Cache

Index	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7
0	2F	1	99	1F	34	56	99	1F	34	56	11	1	DE	AD	BE	EF	DE	AD	BE	EF
1	2C	0	-	-	-	-	-	-	-	-	22	0	-	-	-	-	-	-	-	-
2	01	1	54	21	65	78	54	21	65	78	3F	0	-	-	-	-	-	-	-	-
3	0F	1	01	02	03	04	05	06	07	08	12	1	CA	FE	12	34	CA	FE	12	34
4	36	1	3E	DE	AD	0F	3E	DE	AD	0F	34	0	-	-	-	-	-	-	-	-
5	3D	0	-	-	-	-	-	-	-	-	23	1	1F	2E	11	09	1F	2E	11	09
6	23	1	12	5E	67	90	12	5E	67	90	12	0	-	-	-	-	-	-	-	-
7	13	0	-	-	-	-	-	-	-	-	0F	1	12	34	56	78	13	24	57	68

A. Specify which bits correspond to the components of the 16-bit virtual address, namely, the virtual page number (VPN) and the virtual page offset (VPO) by placing “VPN” or “VPO” in each cell.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VPN	*VPN*	*VPN*	*VPN*	*VPN*	*VPN*	*VPN*	*VPN*	*VPO*	*VPO*	*VPO*	*VPO*	*VPO*	*VPO*	*VPO*	*VPO*

B. Now do the same for the TLB by identifying the bits that are used for the TLB set index and the TLB tag, use the labels “TI” and “TT”, respectively.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TT	*TT*	*TT*	*TT*	*TT*	*TT*	*TI*	*TI*								

C. Working with the 12-bit physical address, specify which bits correspond to the physical page number (PPN) and the physical page offset (PPO) by using “PPN” and “PPO” labels in each cell.

11	10	9	8	7	6	5	4	3	2	1	0
PPN	*PPN*	*PPN*	*PPN*	*PPO*	*PPO*	*PPO*	*PPO*	*PPO*	*PPO*	*PPO*	*PPO*

D. Identify the elements of the physical address used by the cache: tag, set index, and block offset. Use the labels “CT”, “CI”, and “CO”, respectively.

11	10	9	8	7	6	5	4	3	2	1	0
CT	*CT*	*CT*	*CT*	*CT*	*CT*	*CI*	*CI*	*CI*	*CO*	*CO*	*CO*

E. Determine the returned values for the following sequence of accesses and specify whether a TLB miss, page fault, and/or cache miss occurred (by writing "Y" or "N" for yes or no, respectively). 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?
0x056A	*0xF6A*	*ND*	*N*	*N*	*Y*
0x01C2	*0xBC2*	*0x34*	*N*	*N*	*N*
0x00FC	*0x3FC*	*0x13*	*Y*	*N*	*N*
0x0400	*ND*	*ND*	*Y*	*Y*	*ND*

4. Entrée 2 (Virtual Memory) – 70pts total (25/A, 5/B, 5/C, 35/D)

We want to implement a version of `malloc` that uses a best-fit policy for finding a free block. To do this, we need to modify our `searchFreeList` function from the last lab assignment so that it returns the best-fit block rather than the first-fit.

A. Implement the body of `searchFreeList`:

```
static void * searchFreeList(size_t reqSize) ;

/* This function takes one argument, reqSize, that corresponds
to the minimum size of the block required, and returns a
pointer to the header of the block. */
```

- `reqSize` includes the size of the header
- If there are no free blocks that can fit the given `reqSize` then just return NULL
- Assume that all blocks are less than `INT_MAX` in size
- If there are multiple blocks that are equally best fit then just return either

```
static void * searchFreeList(size_t reqSize) {

BlockInfo * curFreeBlock;
size_t bestFitSize = INT_MAX;
BlockInfo * bestFitBlock = NULL;

curFreeBlock = FREE_LIST_HEAD;

// INSERT YOUR CODE HERE.  SHOULD BE 5-12 LINES.




while (curFreeBlock != NULL){
   size_t curFreeBlockSize = SIZE(curFreeBlock->sizeAndTags);
   if (curFreeBlockSize >= reqSize &&
       curFreeBlockSize < bestFitSize) {
         bestFitSize = curFreeBlockSize;
         bestFitBlock = curFreeBlock;
   }
   curFreeBlock = curFreeBlock->next;
}




return bestFitBlock;
}
```

B. In **ONE** sentence give an advantage of a best-fit policy.

Less fragmentation.

C. In **ONE** sentence give a disadvantage of a best-fit policy.

Slower, O(n) for `malloc`*, etc.*

D. Suppose we are implementing a mark and sweep garbage collector. Implement the mark function.

```
void mark(void * ptr);

/* This function takes a pointer, checks if it points to the
payload of an allocated block, and if so, will mark that block
and recursively mark all blocks that are reachable. */
```

Details:

- We are working on a 64-bit machine with an 8-byte word size and alignment.
- We have the function:

  ```
  BlockInfo * is_pointer(void * ptr);

  /* If ptr points somewhere inside the payload of an
  allocated block on the heap then this returns a pointer
  to the beginning of that block. Otherwise, it will return
  NULL. */
  ```
- We have an additional tag called `TAG_MARKED` that indicates whether a block has already been marked (the 3rd lowest-order bit) and is defined as follows: `#define TAG_MARKED 4`
- All valid pointers start at word-aligned boundaries.

```
static void mark(void * ptr){
  BlockInfo * block = is_pointer(ptr);
  //The pointer isn't valid
  if (block == NULL){
    return;
  }
  //If the block has already been marked then just return
  if (block->sizeAndTags & TAG_MARKED){
    return;
  }

// INSERT YOUR CODE HERE.  SHOULD BE 5-10 LINES.

  //Mark the block
  block->sizeAndTags |= TAG_MARKED;

  //Recursively mark every possible pointer in the block
  void * possiblePtr = UNSCASED_POINTER_ADD(block,
                         WORD_SIZE);
  void * endOfBlock  = UNSCALED_POINTER_ADD(block,
                         SIZE(block->sizeAndTags));
  while (possiblePtr < endOfBlock){
    mark(possiblePtr);
    possiblePtr++;
  }

}
```

5. Dessert Tray (Caches, Assembly) – 20pts total (A/10, B/10)

A. If a cache has a block size of 256 bytes, what is the miss rate we expect in a row-major sequential traversal of an array of 32-byte structs (assume we make four accesses to each struct)?

*Since our blocks are 256 bytes, there will be 8 structs in each block loaded from memory. Thus, the first access to a struct will generate a miss, the next three accesses to the same struct will be hits as will the 4 accesses to each of the next 7 structs. Thus, there will be one miss every 4*8 accesses or a miss rate of 1/32 or 3.125%.*

B. Given a struct with three elements such as:

```
struct person {
     int i;
     char c[48];
     float t;
}
```

and that a pointer to such a struct is currently stored in %rax and an index to be used for the array c is stored in %rbx, write a single x86-64 assembly instruction to read that indexed character from memory and place it in %rcx's low-order bits while zeroing the rest.

```
movzbq    4(%rax, %rbx), %rcx
```

REFERENCES

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = .5$
$2^2 = 4$	$2^{-2} = .25$
$2^3 = 8$	$2^{-3} = .125$
$2^4 = 16$	$2^{-4} = .0625$
$2^5 = 32$	$2^{-5} = .03125$
$2^6 = 64$	$2^{-6} = .015625$
$2^7 = 128$	$2^{-7} = .0078125$
$2^8 = 256$	$2^{-8} = .00390625$
$2^9 = 512$	$2^{-9} = .001953125$
$2^{10} = 1024$	$2^{-10} = .0009765625$

Assembly Code Instructions:

`push`	push a value onto the stack and decrement the stack pointer
`pop`	pop a value from the stack and increment the stack pointer
`call`	jump to a procedure after first pushing a return address onto the stack
`ret`	pop return address from stack and jump there
`mov`	move a value between registers and memory
`lea`	compute effective address and store in a register
`add`	add src (1st operand) to dst (2nd) with result stored in dst (2nd)
`sub`	subtract src (1st operand) from dst (2nd) with result stored in dst (2nd)
`and`	bit-wise AND of src and dst with result stored in dst
`or`	bit-wise OR of src and dst with result stored in dst
`sar`	shift data in the dst to the right (arithmetic shift) by the number of bits specified in 1st operand
`jmp`	jump to address
`jne`	conditional jump to address if zero flag is not set
`cmp`	subtract src (1st operand) from dst (2nd) and set flags
`test`	bit-wise AND src and dst and set flags

Suffixes for `mov` instructions:

`s` or `z` for sign-extended or zero-ed

Suffixes for all instructions:

`b`, `w`, `l`, or `q` for byte, word, long, and quad, respectively

Reference from Lab 5:

The functions, macros, and structs from lab5. These are all identical to those in the lab. Note that some of them will not be needed in answering the following questions.

Structs:

```
struct BlockInfo {
  // Size of the block (in the high bits) and tags for whether the
  // block and its predecessor in memory are in use.  See the SIZE()
  // and TAG macros, below, for more details.
  size_t 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;
};
```

Macros:

```
/* Macros for pointer arithmetic to keep other code cleaner.  Casting
   to a char* has the effect that pointer arithmetic happens at the
   byte granularity. */
#define UNSCALED_POINTER_ADD …
#define UNSCALED_POINTER_SUB …

/* TAG_USED is the 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).  If the previous block is not used, we can learn the
   size of the previous block from its boundary tag */
#define TAG_PRECEDING_USED 2;

/* SIZE(blockInfo->sizeAndTags) extracts the size of a 'sizeAndTags'
   field. Also, calling SIZE(size) selects just the higher bits of
   'size' to ensure that 'size' is properly aligned.  We align 'size'
   so we can use the low bits of the sizeAndTags field to tag a block
   as free/used, etc, like this:

      sizeAndTags:
      +-----------------------------------------+
      | 63 | 62 | 61 | 60 |  . . . .  | 2 | 1 | 0 |
      +-----------------------------------------+
        ^                                     ^
      high bit                            low bit

   Since ALIGNMENT == 8, we reserve the low 3 bits of sizeAndTags for
   tag bits, and we use bits 3-63 to store the size.
   Bit 0 (2^0 == 1): TAG_USED
   Bit 1 (2^1 == 2): TAG_PRECEDING_USED
*/
#define SIZE …
```

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

/* Size of a word on this architecture. */
# define WORD_SIZE 8

/* Minimum block size (to account for size header, next ptr, prev ptr,
   and boundary tag) */
#define MIN_BLOCK_SIZE …

/* Pointer to the first BlockInfo in the free list, the list's head.
   A pointer to the head of the free list in this implementation is
   always stored in the first word in the heap.  mem_heap_lo() returns
   a pointer to the first word in the heap, so we cast the result of
   mem_heap_lo() to a BlockInfo** (a pointer to a pointer to
   BlockInfo) and dereference this to get a pointer to the first
   BlockInfo in the free list. */
#define FREE_LIST_HEAD …
```

Functions:

```
/* Insert freeBlock at the head of the list.  (LIFO) */
void insertFreeBlock(BlockInfo* freeBlock);

/* Remove a free block from the free list. */
void removeFreeBlock(BlockInfo* freeBlock);

/* Coalesce 'oldBlock' with any preceeding or following free blocks. */
void coalesceFreeBlock(BlockInfo* oldBlock);

/* Allocate a block of size size and return a pointer to it. */
void * mm_malloc (size_t size);

/* Free the block referenced by ptr. */
void mm_free (void *ptr);
```