CSE351 Autumn 2014 – Final Exam (10 Dec 2014)

Please read through the entire examination first! We designed this exam so that it can be completed in 110 minutes and, hopefully, this estimate will prove to be reasonable.

There are 6 problems for a total of 150 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 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. 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 (no summary sheets, no calculators, no mobile phones, no laptops). 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.

Good Luck!

Name: **__Sample Solution_____**

UWNet ID: **_____**

Problem	Max Score	Score
1 (Potpourri)	30	
2 (Caches)	35	
3 (Virtual Memory)	24	
4 (Memory Allocation)	32	
5 (Java)	10	
6 (Variety Pack)	19	
TOTAL	**150**	

1. Potpourri! True/False (30 total, 2 pts each)

	True	False
A. Freeing memory with an implicit free list takes time linear in the number of free blocks in the worst case.		**F**
B. At runtime, a program can tell with certainty whether data are stored in the cache, physical memory, or disk.		**F**
C. Accessing an address in memory can be both a TLB hit and a cache miss.	**T**	

D. In C, the following declarations are equivalent (i.e. the variable a can be used in the same way in the remainder of the program in either case):
`int a[];`
`int* a;`
	T	

E. In C, it is okay for malloc to move allocated blocks around in the heap to get better utilization of memory.		**F**
F. In Java on x86-64, you can cast a long int to a pointer.		**F**
G. On x86-64, casting a C float to double has no precision loss.	**T**	
H. TLB is used to improve the speed of transferring data from memory to cache.		**F**
I. With garbage collection, the compiler identifies objects that you are no longer using and garbage collects them.		**F**
J. Java programs cannot take advantage of locality.		**F**
K. If I give you a Java virtual machine that runs on an x86 processor, you can also run the exact same virtual machine on a MIPS processor.		**F**
L. The compiler determines whether local variables are allocated on the stack or stored in registers.	**T**	
M. The operating system determines which process gets to run next.	**T**	
N. In C, the programmer decides whether arrays are allocated on the heap or the stack.	**T**	
O. The compiler assigns process IDs to individual processes.		**F**

2. Caches – 35 pts total (14/A, 6/B, 15C)

A. You are given a direct-mapped cache of total size 256 bytes, with cache block size of 16 bytes. The system's page size is 4096 bytes. The following C array has been declared and initialized to contain some values:

```
int x[2][64];
```

i. How many sets will the cache have?

256/16 = 16 sets

ii. How many bits will be required for the cache block offset?

4 bits

iii. If the physical addresses are 22 bits, how many bits are in the cache tag?

22 – 4 – 4 = 14 bits

iv. Assuming that all data except for the array **x** are stored in registers, and that the array **x** starts at address 0x0. Give the miss rate (as a fraction or a %) and total number of misses for the following code, assuming that the cache starts out empty:

```
int sum = 1;
int i;
for (i = 0; i < 64; i++) {
   sum += x[0][i] + x[1][i];
}
```

Miss Rate: ___**100%**_____ Total Number of Misses: ___**128**_____

v. What if we maintain the same total cache size and cache block size, but increase the associativity to 2-way set associative. Now what will be the miss rate and total number of misses of the above code, assuming that the cache starts out empty?

Miss Rate: ____**25%**_____ Total Number of Misses: ____**32**_____

2. (cont.)

B. Given the following access results in the form (address, result) on an empty cache of total size 16 bytes, what can you infer about this cache's properties? Assume LRU replacement policy. **Circle all that apply**.

(0, Miss), (8, Miss), (0, Hit), (16, Miss), (8, Miss)

a. The block size is greater than 8 bytes

b. The block size is less or equal to 8 bytes

c. This cache has only two sets

d. This cache has more than 8 sets

e. This cache is 2-way set associative

f. The cache is 4-way set associative

g. Using an 8 bit address, the tag would be 4 bits

h. Using an 8 bit address, the tag would be greater than 4 bits

i. None of the above

Block sizes will range from 1 to 8 bytes, thus the number of sets will range from 1 to 8. All combos of block sizes and number of sets will use only 3 bits, leaving 5 bits for the tag.

2. (cont.)

C. Given the following 2-way set-associative cache and its contents in a system with a 10-bit address:

Index	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7
0	07	1	99	1F	34	56	99	1F	34	56	11	1	DE	AD	BE	EF	DE	AD	BE	EF
1	03	1	27	A4	C5	23	00	00	00	01	1C	1	1F	2E	11	09	1F	2E	11	09
2	01	1	54	21	65	78	54	21	65	78	0F	0	CA	FE	12	34	CA	FE	12	34
3	0F	1	01	02	03	04	05	06	07	08	1C	0	12	34	56	78	13	24	57	68

What are the results of the following read operations (specify whether it is a hit or miss and the value if is determinable from the information given, otherwise just write ND for non-determinable)? Assume the cache uses a LRU replacement policy and that reads are executed in the order given below (addresses are given in hex).

Address to be read	Tag (give bits)	Set (give bits)	Block Offset (give bits)	Hit or Miss (H or M)	Value read (or ND)
0x389	***11100***	***01***	***001***	***H***	***2E***
0x30C	***11000***	***01***	***100***	***M***	***ND***
0x3BB	***11101***	***11***	***011***	***M***	***ND***
0x308	***11000***	***01***	***000***	***H*** *(brought in by0x30C access)*	***ND***
0x0E3	***00111***	***00***	***011***	***H***	***56***

3. Virtual Memory – 24 pts total (3/A, 3/B, 3/C, 15/D)

We have a system with the following properties:

- a virtual address of 13 bits (4 hex digits),
- a physical address of 11 bits (3 hex digits),
- pages that are 64 bytes,
- a corresponding page table, and
- a TLB with 16 entries that is 4-way set associative.

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

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	07	00	1	06	-	0	1F	03	1
1	0C	03	1	0A	-	0	00	0B	1	01	0F	1
2	07	-	0	0C	02	1	0F	01	1	0B	-	0
3	01	1C	1	0C	01	1	04	01	0	1A	01	1

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

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

3. (cont.)

A. Specify which bits correspond to the components of the 13-bit virtual address, namely, the virtual page number (VPN) and the virtual page offset (VPO) by placing "VPN" or "VPO" in each cell.

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

B. Now do the same for the TLB by identifying the bits of the 13-bit virtual address that are used for the TLB set index and the TLB tag, use the labels "TI" and "TT", respectively. Leave any other bits blank.

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

C. Working with the 11-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.

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

3. (cont.)

D. Determine the physical address, TLB miss or hit, and whether there is a page fault for the following virtual address accesses (write "Y" or "N" for yes or no, respectively, in the TLB Miss? And Page Fault? columns). If you can't determine the PPN and/or physical address and/or TLB miss and/or Page Fault, simply write ND (for non-determinable) in the appropriate entry in the table.

Virtual Address	VPN (give bits)	TT (give bits)	TI (give bits)	PPN (give bits)	Physical Address (give bits)	TLB Miss?	Page Fault?
0x0C40	*0110001*	*01100*	*01*	*00011*	*00011000000*	*N*	*N*
0x02CF	*0001011*	*00010*	*11*	*01000*	*01000001111*	*Y*	*N*
0x130A	*1001100*	*10011*	*00*	*ND*	*ND*	*Y*	*ND*

4. Memory Allocation – 32pts total (3/A, 20/B, 9/C-E)

Why did you leave your laptop open when you went back for that second piece of pumpkin pie?! Your well-meaning younger cousin tried to "help you" finish lab 5 but you suspect he/she has actually inserted bugs into your free list. To track down the problem, you need to implement the `bugCheck()` method, which scans your free list, and checks each block to see if the tag bits in its header properly indicate that this block is free. If it finds a block that is marked improperly, it should set the 3rd low order bit to 1 (you knew there was something useful we could do with that bit!).

- It should leave all other bits as it finds them.
- It only needs to check and set tag bits in the header, not in the boundary tag.
- It does not need to check if the bit for "predecessor in use" is set properly.

`bugCheck()` should also return the total number of bytes on the free list that it has tagged in this manner as potential bugs. There is lab 5 code on the last pages of the exam.

A) Fill in the value you use to set the 3rd low order bit. You must use this in your code.

```
#define POSSIBLE_BUG   ___4__________________
```

B) Implement the body of `bugCheck()`

```
/* This function scans the free list and for each block that is not
properly marked as free in the header:
   -  Set its POSSIBLE_BUG bit to 1
   -  Add its size to the running total of buggy bytes
It returns the total number of bytes tagged as possible bugs. */

static size_t bugCheck(void) {

   BlockInfo * curFreeBlock;
   size_t bugBytes = 0;
   curFreeBlock = FREE_LIST_HEAD;
   // INSERT YOUR CODE HERE. SHOULD BE 5-12 LINES.
```

```
   while (curFreeBlock != NULL) {
       if (curFreeBlock->sizeAndTags & TAG_USED) {
         bugBytes += SIZE(curFreeBlock->sizeAndTags);
         curFreeBlock->sizeAndTags |= POSSIBLE_BUG;
       }
       curFreeBlock = curFreeBlock->next;
  }
```

```
   return bugBytes;
}
```

4. (cont.)

C) True/False In a C program, freeing the same address multiple times has no effect.

D) In a C program, forgetting to free something is called ___**a memory leak**_____.

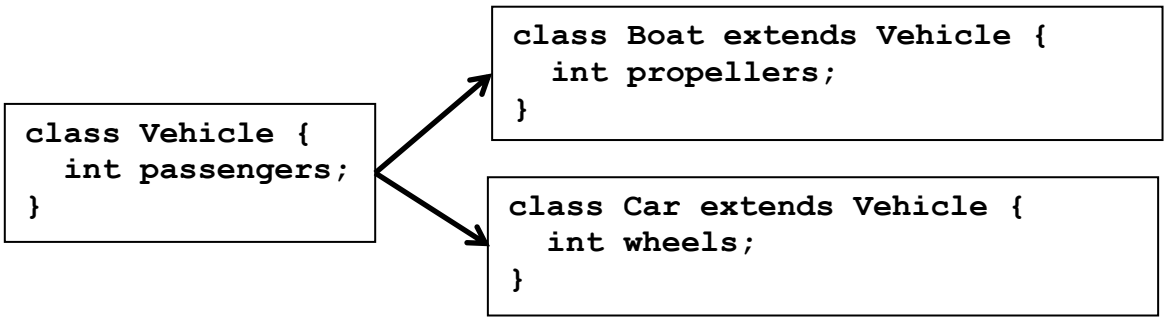
E) For each of the following, indicate whether it is likely to have a positive impact on Utilization (U) or Throughput(T). If it will likely have a positive impact on both, check both U and T. If it is not likely to have a positive impact on either, check Neither.

Memory Allocation Technique	Utilization (U)	Throughput (T)	Neither
First Fit		X	
Segregated List Allocator	X	X	
Coalescing	X		
Best-Fit	X		
Splitting	X		

5. Java – 10pts total

A. A vtable in Java is very similar to __**a jump table**_____ (something we discussed earlier in the course).

B. Given the following Java class hierarchy:



And the following additional code:

```
class FinalExam {

     public static void main(String[] args) {
          Vehicle  v1 = new Vehicle();  // line 1
          Car      c  = new Car();      // line 2
          Vehicle  v2 = new Car();      // line 3
          Boat     b  = (Boat) v1;      // line 4
     }
}
```

Circle <u>all</u> of the items below that will be <u>true</u>:

i. Line 3 will cause a compiler error.

ii. Line 4 will cause a compiler error.

iii. Line 3 will cause a run-time error.

iv. <u>Line 4 will cause a run-time error.</u>

v. Variable `c` will be on the heap.

vi. <u>What `v1` refers to will be on the heap.</u>

vii. `Car` objects will only have space allocated for one `int` field (`wheels` and `passengers` will refer to the same location in memory).

viii. **<u>Objects of the same type can share a single vtable.</u>**

6. Variety Pack – 19pts total

A. Which of the following are considered to be a part of an ISA? (Circle all that apply)
   i. The size of physical memory
   ii. The number of registers
   iii. The word size
   iv. The size of the cache in bytes
   v. The number of cycles per instruction

B. Given a multidimensional array of floats A[6][8], if A starts at address 0, at what byte address is the element A[4][7]? (all addresses given in decimal)
   i. 240
   ii. 312
   iii. 156
   iv. 200
   v. None of the above

C. Which of the following is not true about Java? (Circle all that apply)
   i. Java characters inside of a String are two bytes in size
   ii. An array's size is stored at the front of the array
   iii. Each object in Java stores a pointer to its class' vtable
   iv. When compiled, Java code is turned into Java bytecode
   v. None of the above ***(None of the above are false statements.)***

D. Given the following struct in x86-64:

```
struct student {
   char name[25];
   int id;
   char year[10];
   double gpa;
};
```

What is the total size of this struct in bytes?

name:25 (+ 3 = 28), id:4 (=32), year:10 (+ 6 = 48), gpa:8 = 56 bytes total

As a programmer, could you have declared this struct differently so that it uses less memory? If no, explain why not. If yes, show how you would declare it and give the new total size in bytes.

```
struct student {
   double gpa;
   int id;
   char name[25];
   char year[10];
};
```

Note that the order of name and year could be reversed. Total = 48 bytes, 47 + 1 byte at the end.

6. (cont.)

E. Assume the following C instructions have been executed:

```
int a = 4;
int* b = &a;
```

In the debugger, you are at a breakpoint and observe that **`b`** is stored in **`%rdx`**. After the following instruction is executed:

```
lea (%rdx,%rdx,4), %rdx
```

Which (if any) of these will definitely be true: (Circle all that apply)

i. a will contain the value 5

ii. a will contain the value 20

iii. b will contain the value 5

iv. b will contain the value 20

v. %rdx will contain the value 5

vi. %rdx will contain the value 20

vii. None of the above

F. In Java, a 2-d array of (4-byte) ints: (Circle all that apply)

i. Can be stored on the heap

ii. Can be stored on the stack

iii. Are stored in one contiguous block

iv. Take up more total space than the same array declared in C

v. None of the above

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
`shl`	shift data in the dst to the left (logical shift) by the number of bits specified in the 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, respectively

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 exam 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 …
```