Full Name:————————————————

Student ID #:————————————————

UW CSE 351, Winter 2013

Final Exam

March 20, 2013
2:30pm - 4:20pm

Instructions:

- Write your full name and UW student ID number on the front of the exam. When the exam begins, make sure that your copy is not missing any of the 13 pages before proceeding.

- Read over the entire exam before starting to work, and be sure to carefully read the instructions for every problem.

- Write your answers in the space provided below or next to each problem. If you make a mess, clearly indicate your final answer. **Be sure to answer all parts of all questions!**

- Some pages can be removed from the exam to avoid flipping back and forth while working on the problems. A note at the top of the page will indicate these pages. If you remove these pages, make sure that the rest of your exam is still securely fastened.

- Don't spend too much time on one problem if there are other problems that you haven't answered yet. There are 100 total points and 110 minutes to take the exam.

- **No books, notes, or electronic devices may be used during the exam.** You may not communicate with other students during the exam, but please ask the instructor / TAs if you need clarification for some problem.

Problem 1	(16 points):
Problem 2	(20 points):
Problem 3	(28 points):
Problem 4	(24 points):
Problem 5	(12 points):
TOTAL	(100 points):

Problem 1 (16 points):

Answer the following questions with a few words or sentences:

A. What are the two key abstractions that processes provide for programmers?

B. After an exception occurs and the operating system's exception handler finishes running, one of **three** things may happen. What are these three possible actions?

C. Describe one way to achieve good *spatial locality* in the programs that you write.

D. In a computer system that uses virtual memory, how many page tables are in the system?

Problem 2 (20 points):

A bitmap image is composed of pixels. Each pixel in the image is represented using four values: three for the primary colors (red, green and blue - RGB) and one for the transparency information defined as an alpha channel.

In this problem, you will evaluate the cache performance of code that traverses a bitmap of pixels. You will use a **direct-mapped cache of size 512 bytes with 8-byte blocks**.

You are given the following definitions:

```
typedef struct {
    char r;
    char g;
    char b;
    char a;
} pixel;

pixel bitmap[16][16];
int i, j;
int sum_r = 0, sum_g = 0, sum_b = 0, sum_a = 0;
```

Also assume that:

- `sizeof(char) = 1`.

- `bitmap` begins at memory address `0`.

- The cache is initially empty.

- The `bitmap` array is stored in row-major order.

- The variables with type `int` are stored in registers and any access to these variables does not cause a cache miss or impact the cache in any way.

A. How many sets are in the cache? _____

B. How many bits are in the block offset? _____

C. How many bits are in the set index? _____

Problem 2, continued:

D. What percent of the cache reads in the following code will result in a cache miss?

```
for (i = 0; i < 16; i++){
    for (j = 0; j < 16; j++){
        sum_r += bitmap[i][j].r;
        sum_g += bitmap[i][j].g;
        sum_b += bitmap[i][j].b;
        sum_a += bitmap[i][j].a;
    }
}
```

Miss rate for reads from `bitmap`: _____%

E. If the cache size were doubled, what would the miss rate for the previous part now be? _____%

F. There are three types of cache misses: cold/compulsory, conflict, and capacity misses. Which of these types of cache misses occur when the above code is run on the 512 byte cache?

Problem 3 (28 points):

In this problem you will translate virtual addresses into physical addresses to access data in the memory hierarchy. If you wish, you may carefully detach this page for use on the following problems; you do not need to turn in this page.

The system has the following characteristics:

- Memory is byte addressable and memory accesses are to **1-byte words** (not 4-byte words).

- Virtual addresses are 16 bits wide.

- Physical addresses are 13 bits wide.

- The page size is 512 bytes.

- The TLB is 8-way set associative with 16 total entries.

- There is a single data cache that is 2-way set associative, with a 4 byte block size and 8 sets.

In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB, the page table for the first 32 pages, and the data cache are as follows:

TLB			
Index	Tag	PPN	Valid
0	09	2	1
	12	2	1
	10	0	1
	08	0	1
	05	3	0
	13	1	0
	10	3	0
	18	3	0
1	04	4	0
	0C	2	0
	12	0	0
	08	5	0
	06	6	0
	03	3	0
	07	0	0
	02	7	1

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	6	1	10	0	1
01	5	0	11	5	0
02	3	1	12	2	1
03	4	1	13	4	0
04	2	0	14	6	0
05	7	1	15	2	0
06	1	0	16	4	0
07	3	0	17	6	0
08	5	1	18	1	0
09	4	0	19	2	0
0A	3	0	1A	5	1
0B	2	0	1B	7	1
0C	5	0	1C	6	0
0D	6	0	1D	2	0
0E	1	1	1E	3	0
0F	0	0	1F	1	0

2-way Set Associative Cache											
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	19	1	99	11	23	11	00	0	99	11	23	11
1	15	0	4F	22	EC	11	2F	1	55	59	0B	41
2	1B	1	00	02	04	08	0B	1	01	03	05	07
3	06	0	84	06	B2	9C	12	0	84	06	B2	9C
4	07	0	43	6D	8F	09	05	0	43	6D	8F	09
5	0D	1	36	32	00	78	1E	1	A1	B2	C4	DE
6	11	0	A2	37	68	31	00	1	BB	77	33	00
7	16	1	11	C2	11	33	1E	1	00	C0	0F	00

Problem 3, Part 1 (8 points):

Make sure that we can tell your *I*'s from your *T*'s in your answers to these questions!

A. The box below shows the bits of a virtual address. Indicate (by labeling the diagram) the bits that would be used to determine the following:

   VPO   The virtual page offset
   VPN   The virtual page number
   TI   The TLB index
   TT   The TLB tag

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
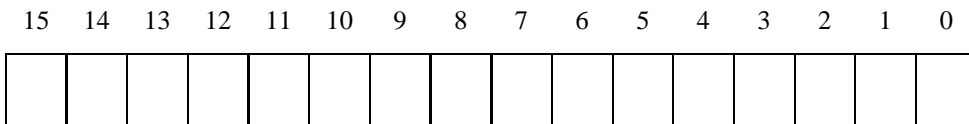
B. The box below shows the bits of a physical address. Indicate (by labeling the diagram) the bits that would be used to determine the following:

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

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

Problem 3, Part 2:

Perform the address translation and data cache access for the given virtual address by filling in all of the parts below with values **in hex**.

If there is a page fault, enter "-" for "PPN" and leave parts C and D blank. If there is a cache miss, enter "-" for "Cache Byte returned".

Virtual address: 0x3155

A. Virtual address in binary (one bit per box):

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

B. Address translation:

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

C. Physical address in binary (one bit per box):

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

D. Physical memory reference:

Parameter	Value
Cache Block Offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Problem 3, Part 3:

Perform the address translation and data cache access for the given virtual address by filling in all of the parts below with values **in hex**.

If there is a page fault, enter "-" for "PPN" and leave parts C and D blank. If there is a cache miss, enter "-" for "Cache Byte returned".

Virtual address: `0x1DDE`

A. Virtual address in binary (one bit per box):

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

B. Address translation:

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

C. Physical address in binary (one bit per box):

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

D. Physical memory reference:

Parameter	Value
Cache Block Offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Problem 4 (24 points):

The table below shows the contents of the heap on a 64-bit big-endian system. The heap is managed by a dynamic memory allocator that uses an explicit free list like that used in Lab 5. The allocator maintains 8-byte block alignment, and initially has FREE_LIST_HEAD = 0x02200008. As in Lab 5, allocated blocks begin with a sizeAndTags header and have no footer, while free blocks begin with a BlockInfo struct and have a sizeAndTags footer. Bit 0 of sizeAndTags is set if this block is allocated (TAG_USED), and bit 1 of sizeAndTags is set if the preceding block is allocated (TAG_PRECEDING_USED).

If you wish, you may carefully detach this page for use on the following problems; you do not need to turn in this page.

Address in memory	Contents
0x022000e0	0x00000000 00000001
0x022000d8	0x00000000 00000042
0x022000d0	0x0df0adba 0df0adba
0x022000c8	0x0df0adba 0df0adba
0x022000c0	0x0df0adba 0df0adba
0x022000b8	0x0df0adba 0df0adba
0x022000b0	0x00000000 02200008
0x022000a8	0x00000000 02200050
0x022000a0	0x00000000 00000042
0x02200098	0x0df0adba 0df0adba
0x02200090	0xaaaaaaaa aaaaaaaa
0x02200088	0xaaaaaaaa aaaaaaaa
0x02200080	0x00000000 00000021
0x02200078	0x00000000 00000032
0x02200070	0x00000000 00000020
0x02200068	0x00000000 00000020
0x02200060	0x00000000 022000a0
0x02200058	0x00000000 00000000
0x02200050	0x00000000 00000032
0x02200048	0x0df0adba 0df0adba
0x02200040	0x0df0adba 0df0adba
0x02200038	0x0df0adba 0df0adba
0x02200030	0x00000000 00000021
0x02200028	0x00000000 0000002a
0x02200020	0x00000000 00000000
0x02200018	0x00000000 00000000
0x02200010	0x00000000 022000a0
0x02200008	0x00000000 0000002a
0x02200000	0x00000000 02200008

```
struct BlockInfo {
    size_t sizeAndTags;
    struct BlockInfo* next;
    struct BlockInfo* prev;
};
```

Problem 4 (24 points):

Note: for each of these problems, assume that the state of the heap is that shown on the previous page - do not include the effects of any previous problems when answering later problems.

A. Suppose the application calls `malloc(5 * WORD_SIZE)`, requesting a new allocation of 5 **words**. What address will be returned to the application if the allocator uses a *first-fit* policy (meaning it begins the search for a free block from the beginning of the free list every time)?

B. Suppose the application calls `malloc(5 * WORD_SIZE)`, requesting a new allocation of 5 **words**. What address will be returned to the application if the allocator uses a *best-fit* policy?

C. The application just called `malloc(2 * WORD_SIZE)` to request a new allocation of 2 **words**, and received the address `0x02200088` as a return value. How many payload words does this allocated block in the heap actually have room for? What is this an example of?

Problem 4, continued:

D. Suppose the application calls `malloc(10 * WORD_SIZE)`, requesting a new allocation of 10 *words*. Even though there are more than 10 free words in the heap, the allocator cannot immediately handle this request. What is this an example of?

E. Suppose the application calls `malloc(10 * WORD_SIZE)`, requesting a new allocation of 10 *words*. The allocator cannot immediately handle this request - what must the allocator do first before it will be able to find a suitable free block?

F. What are the two performance goals that all memory allocators attempt to maximize, but which are often in conflict with each other?

Problem 5 (12 points):

Answer the following questions with a few words or sentences:

A. What is one difference between references in Java and pointers in C?

B. Consider the following Java class declaration. How many *dereference operations* are performed every time a Point object's `samePlace()` method is called?

```java
class Point {
    double x;
    double y;

    Point() {
        x = 0;
        y = 0;
    }

    boolean samePlace(Point p) {
        return (x == p.x) && (y == p.y);
    }
}
```

C. What is the main advantage of interpreting Java code in a virtual machine, as opposed to compiling it to machine code ahead of time as is done for C?

References

If you wish, you may carefully detach this page from the exam; you do not need to turn in this page.

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = 0.5$
$2^2 = 4$	$2^{-2} = 0.25$
$2^3 = 8$	$2^{-3} = 0.125$
$2^4 = 16$	$2^{-4} = 0.0625$
$2^5 = 32$	$2^{-5} = 0.03125$
$2^6 = 64$	$2^{-6} = 0.015625$
$2^7 = 128$	$2^{-7} = 0.0078125$
$2^8 = 256$	$2^{-8} = 0.00390625$
$2^9 = 512$	$2^{-9} = 0.001953125$
$2^{10} = 1024$	$2^{-10} = 0.0009765625$

$2^a * 2^b = 2^{a+b}$

$2^a / 2^b = 2^{a-b}$

Hexadecimal to binary conversion:

```
0x0 = 0000
0x1 = 0001
0x2 = 0010
0x3 = 0011
0x4 = 0100
0x5 = 0101
0x6 = 0110
0x7 = 0111
0x8 = 1000
0x9 = 1001
0xA = 1010
0xB = 1011
0xC = 1100
0xD = 1101
0xE = 1110
0xF = 1111
```

Hexadecimal to decimal conversion:

```
0x10 = 16
0x20 = 32
0x30 = 48
0x40 = 64
0x50 = 80
0x60 = 96
...
```