CS/ECE 354 - Final Exam
Spring 2016

Name: _____

UW Student ID Number: _____

CS 354 Section Number: _____

Instructions:

1. Do not open this exam until you are instructed to by the examiners.
2. Fill in your **NAME** and **UW Student ID** on the first page of this exam.
3. You have **2 hours** to answer all the questions.
4. You are allowed to use **one handwritten 8.5 x 11 inch cheat sheet** during the exam.
5. Please check that this exam contains **18 printed pages** with a total of **8 questions**.
6. You are **NOT allowed** to use any electronic devices (including calculators) during the exam.

Grading

1. _____	5. _____	
2. _____	6. _____	Total: _____
3. _____	7. _____	
4. _____	8. _____	

1. Dynamic Memory Allocator [10 points]

Allocator properties:

1. **Single word** (4 bytes) aligned.
2. **Implicit free list** is used for free block organization.
3. All blocks have a **header** of size 4 bytes and a **footer** of size 4 bytes.
4. **bit 0** (least significant bit) in the header indicates the use of the current block:
    a. 1 for allocated
    b. 0 for free
5. **zero-sized payloads** are not allowed.
6. **block size** = sizeof (header) + sizeof (payload) + sizeof (padding) + sizeof (footer)

Please answer the following questions regarding this allocator:

A. Minimum block size = _____12_____

B. For the following memory allocation, what is the size of the payload and padding that will be used in the allocated block?

You should **assume** the following:

a. A free block of size **16 bytes** is chosen by the allocator to satisfy the following malloc request.
b. The header of this chosen free block is at the memory location **0x8040AB08.**

```
char *p = malloc(3);
```

Payload = ___3_____ bytes

Padding = ___5_____ bytes

Memory address stored in the `pointer p (in hexadecimal)`:

0x8040AB0C

C. The contents of the header (and the footer) of a block in the allocator is **0x00000035**.

a. Is the block allocated or free? _____allocated_____

b. What is the size of the block (in decimal)? _____52_____

c. Is the contents of the header of this block valid with respect to this allocator? **Remember:** For a block to be valid with respect to an allocator, its size should satisfy the alignment requirement of the allocator.

YES (OR) NO (YES!)

2. Cache Hits or Misses? [10 points]

This problem tests your understanding of **conflict misses**. Consider the following **matrix copy** function.

```
void matrix_copy (int dst[2][2], int src[2][2]) {
     int i, j;
     for (i = 0; i < 2; i++) {
          for (j = 0; j < 2; j++) {
               dst[i][j] = src[i][j];
          }
     }
}
```

Assume this code runs on a machine with the following properties:

- `sizeof(int) == 4.`
- The `src array` starts at address 0 and the `dst array` starts at address 16 (decimal).
- There is a single L1 data cache that is **direct-mapped, write allocate, and write back** with a **block size of 8 bytes.**
- The cache has a **total size of 16 data bytes** and the cache is **initially empty.**
- Accesses to the src and dst arrays are the only sources of reads and writes, respectively.
- Variables i and j are stored in **registers**.

A. For each row and col, indicate whether the access to `src[row][col]` and `dst[row][col]` is a **hit (h)** or a **miss (m)**. For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

`src array`

	Column 0	**Column 1**
Row 0	m	m
Row 1	m	m

`dst array`

	Column 0	**Column 1**
Row 0	m	m
Row 1	m	m

B. Repeat part A for a cache with a **total size of 32 data bytes**.

`src array`

	Column 0	**Column 1**
Row 0	m	h
Row 1	m	h

`dst array`

	Column 0	**Column 1**
Row 0	m	h
Row 1	m	h

3. Interrupts, Faults, and System calls [10 points]

Assume the CPU is executing a C program and the instruction that is currently being executed is called the current instruction. The following events occur when your C program is executing the current instruction. For these events, specify if they are synchronous or asynchronous and if they return to the current instruction or the next instruction.

For the **Timing** column write: **Synchronous** OR **Asynchronous**
For the **Returns To** column, write **Current Instruction** OR **Next Instruction**

No.	Event	Timing	Returns To
1.	Clicking on a desktop icon using the mouse	Async	Next
2.	Opening a text file for reading	Sync	Next
3.	Accessing a virtual page that is uncached (i.e. present on disk but not in physical memory)	Sync	Current
4.	Increasing the size of the heap memory using `sbrk()`	Sync	Next
5.	The hard disk informs the CPU that a write has been completed	Async	Next

4. Linking [20 points]

Consider the three files **`sum.h, sum.c`** and **`main.c`** as shown below and answer the questions that follow.

`sum.h`

```
=====
   1.    #ifndef SUM_H
   2.    #define SUM_H
   3.    extern int sum(int, int);
   4.    #endif
```

sum.c

```
=====
   1.    #include "sum.h"
   2.    extern int num_ops;
   3.    static int global_sum = 0;
   4.
   5.    int sum(int x, int y)
   6.    {
   7.       global_sum += (x+y);
   8.       num_ops++;
   9.       return x+y;
   10.   }
```

main.c

```
======
   1.    #include "sum.h"
   2.    #define SUCCESS 0
   3.
   4.    int num_ops = 0;
   5.
   6.    int main()
   7.    {
   8.        int a = 10;
   9.        int b = 3;
   10.       int result = sum(a,b);
   11.       return SUCCESS;
   12.   }
```

The final executable **`a.out`** is produced by running the following command:

```
% gcc sum.c main.c -m32
```

Questions:

A. The variable `num_ops` is **ONLY declared but not defined** in the file _sum.c_

B. The variable `num_ops` is **defined** in the file _main.c_

C. Do the following variables / functions need **relocation** when the executable (a.out) is formed? (Just answer: **Yes** or **No**)

   a. Variable `global_sum` in `sum.c` - _Y_

   b. Variable `result` in `main.c` - __N___

   c. Variable `num_ops` in `main.c` - ___Y_____

   d. Function `sum()` in `sum.c` - _____Y_____

   e. Function `main()` in `main.c` - ____Y_____

D. Can the variable `global_sum` be accessed in the file `main.c` without changing the type of the variable `global_sum` in `sum.c`? If yes, explain how. If no, explain why not?

E. What type of object file is **sum.o** ?
sum.o is generated using the command: `gcc -c sum.c -m32`

   i. **Relocatable Object File**

   ii. **Executable Object File**

F. Which part of **program memory** (code, data, stack, heap) are the following variables / functions stored?

   a. Variable `global_sum` in `sum.c` - _____data_____

   b. Variable `result` in `main.c` - _____stack_____

   c. Variable `num_ops` in `main.c` - _____data_____

   d. Function `sum()` in `sum.c` - _____code_(or text)_____

   e. Function `main()` in `main.c` - _____code_(or text)_____

5. Virtual to Physical Address Translation [20 points]

The following are the assumptions of our memory system.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Virtual addresses are **14 bits** wide (n = 14).
- Physical addresses are **12 bits** wide (m = 12).
- The page size is **64 bytes** (P = 64).
- The TLB is 2-way set associative with a total of 8 total entries.
- The L1 d-cache is physically addressed and direct mapped, with a 4-byte cache block size and 4 total sets.
- ALL CONTENTS in the TLB, L1-CACHE and PAGE TABLE are in **HEXADECIMAL**.

The following is a snapshot of the TLB, L1-cache, and Page Table in our system.

Figure A: **TLB** - 4 Sets, 8 entries, 2-way set associative

Set index	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	09	0D	1
1	03	2D	1	02	-	0
2	02	0A	1	08	23	1
3	07	-	0	03	0D	1

Figure B: **L1 CACHE** - 4 sets, 4 byte blocks, direct mapped

Set index	Tag	Valid	block[0]	block[1]	block[2]	block[3]
0	19	1	99	11	23	11
1	5D	1	A0	B0	C0	D0
2	1B	1	00	02	04	08
3	36	0	-	-	-	-

Figure C: **PAGE TABLE** - Only the first 8 entries are shown

VPN	PPN	Valid
00	2F	1
01	-	0
02	3A	1
03	02	1
04	-	0
05	16	1
06	17	1
07	-	0

Fill in the parts A to E below when the **Virtual Address: 0x0197** is translated into its corresponding physical address and when the data byte in that physical address is accessed.

A. Virtual Address (in binary)

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

If there is a page fault, mark "--" for PPN in the table below.

B. Address Translation

Parameter	Value (in hexadecimal)
VPN	0x06
TLB index	2
TLB tag	01
TLB hit? (Y/N)	N
Page fault? (Y/N)	N

PPN	0x17

If there is a page fault, you can leave the parts C, D, and E blank.

C. Physical Address (in binary)

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

D. Physical Address (in hexadecimal): 0x_____5D7_____

E. Physical Memory Reference

If a cache miss happens for the **physical address**, mark "--" for the cache byte returned.

Parameter	**Value (in hexadecimal)**
Byte offset	0x03
Cache index	0x01
Cache tag	0x5D
Cache hit? (Y/N)	Y
Cache byte returned	D0

6. Signals [10 points]

What is the output of the following program if the following event occurs while the program is executing?

EVENT: CTRL + C is pressed ONLY once **after** the program prints the integer 3 and **before** the iteration for x = 4 starts.

Note: For space reasons, we are not checking error return codes. You can assume the program

runs without any errors.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int x = 0;

void sigint_handler(int sig)
{
    x = 7;
}

int main()
{
    signal(SIGINT, sigint_handler);

    while (x <= 10) {
        printf("%d\n", x++);
        sleep(1);
    }

    return 0;
}
```

OUTPUT:

0 1 2 3 7 8 9 10

7. Cache Miss Rate Analysis [10 points]

You are evaluating the cache performance of the following code on a machine with a **64-byte direct-mapped data cache (C = 64)** with **block size of 16-bytes (B = 16)**.

You are given the following definitions:

```
struct point {
      int x;
      int y;
};

struct point grid[4][4];
int total_x = 0, total_y = 0;
int i, j;
```

You should also assume the following:

- sizeof(int) == 4.
- grid begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array grid.
- Variables i, j, total_x, and total_y are stored in registers.

A. Determine the cache performance for the following **code snippet 1**:

Code snippet 1:

```
for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++) {
            total_x += grid[i][j].x;
      }
}

for (i = 0; i < 4; i++) {
```

```
        for (j = 0; j < 4; j++) {
            total_y += grid[i][j].y;
        }
    }
```

1. What is the total number of reads that miss in the cache? **16**

2. What is the miss rate? 50 %

B. Determine the cache performance for the following **code snippet 2**:

Code snippet 2:

```
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            total_x += grid[i][j].x;
            total_y += grid[i][j].y;
        }
    }
```

1. What is the total number of reads that miss in the cache? **8**
2. What is the miss rate? **25%**

C. Which of these 2 code snippets is better with respect to cache performance?

a. Code snippet #1

b. **Code snippet #2**

Why? (just a single line explanation is sufficient)

Better spatial locality or miss rate is lesser.

8. Heap Memory [10 points]

Consider an allocator with the following properties.

Allocator properties:

1. **Single word** (4 bytes) aligned.
2. **Implicit free list** is used for free block organization.
3. **Immediate coalescing** is done when a block is freed.
4. All blocks have a header of size **4 bytes.**
5. **bit 0** (least significant bit) in the header indicates the use of the current block:
    a. 1 for allocated
    b. 0 for free
6. **block size** = sizeof (header) + sizeof (payload) + sizeof (padding)

Given the contents of the heap shown in FIGURE 1, show the new contents of the heap **after** a call to **free(0x804900C)** is executed. i.e. Write the contents of the highlighted memory locations 0x8049000, 0x8049008, and 0x8049018 in FIGURE 2 after the call to free is executed. Your answers should be given as **hexadecimal** values.

Note that the address grows from **bottom up**. i.e. The heap starts at the address 0x8049000 and grows upwards. The first block is stored starting at the address 0x8049000.

Remember that the allocator uses **immediate coalescing**, that is, adjacent free blocks are merged immediately each time a block is freed.

FIGURE 1
(BEFORE call to free)

Memory Address	Value in Memory
...	...
0x8049024	0x1234567F
0x8049020	0x7F3E4501
0x804901c	0xFE670031
0x8049018	0x00000010
0x8049014	0x12CD00AB
0x8049010	0x35670011
0x804900c	0xA1B2C3D4
0x8049008	0x00000011
0x8049004	0x12345678
Start of heap => 0x8049000	0x00000009

FIGURE 2
(AFTER call to free(0x804900C))

Memory Address	Value in Memory
...	...
0x8049024	0x1234567F
0x8049020	0x7F3E4501
0x804901c	0xFE670031
0x8049018	**0x00000010**
0x8049014	0x12CD00AB
0x8049010	0x35670011
0x804900c	0xA1B2C3D4
0x8049008	**0x00000020**
0x8049004	0x12345678
Start of heap => **0x8049000**	**0x00000009**

Good luck! :)

Rough Work

Rough Work

Rough Work