CS/ECE 354 - Finals Practice Exam
Spring 2016

Topics to study for Finals

Topic	Weightage (points)
Cache Memories	20
Dynamic Memory Allocation	20
Virtual Memory	20
Exceptional Control Flow	20
Linking	20
Total	**100**

1. Dynamic Memory Allocator [10 points]

Allocator properties:

1. **Double word** (8 bytes) aligned.
2. **Explicit 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. **Free blocks** have **prev** and **next** pointers of size **4 bytes** each.
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. **zero-sized payloads** are not allowed.
7. **Allocated block size** = sizeof (header) + sizeof (payload) + sizeof (padding) + sizeof (footer)
8. **Free block size** = sizeof (header) + sizeof(prev) + sizeof(next) + sizeof (payload) + sizeof (padding) + sizeof (footer)

Please answer the following questions regarding this allocator:

A. Minimum block size = _____

B. Maximum block size = _____

C. 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 free block of size **32 bytes** is chosen by the allocator to satisfy the below malloc request. The header of this block is at the memory location **0x8090A0B4**

```
char *p = malloc(8);
```

Payload = _____ bytes

Padding = _____ bytes

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

0x_____

D. The contents of the header (and the footer) of a block in the allocator is **0x000000A9**.

   a. Is the block allocated or free? _____

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

   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

2. Cache Hits or Misses? [10 points]

Consider the following **matrix transpose** function.

```
void matrix_transpose (int dst[2][2], int src[2][2]) {
     int i, j;
     for (i = 0; i < 2; i++) {
          for (j = 0; j < 2; j++) {
               dst[j][i] = 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** 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 read and write misses, respectively.

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	
Row 1		

`dst array`

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

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

`src array`

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

`dst array`

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

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

For the following exceptions, specify the type of the exception.
The three types of possible events are:

1. Interrupts
2. System Calls
3. Faults

No.	Event	Exception Type
1.	Accessing memory at virtual address 0x00000000	
2.	Exiting a C program by calling exit(0)	
3.	Pressing CTRL + Z at the keyboard	
4.	Reading some data from a text file using read( )	
5.	Moving the mouse to click on a desktop icon	

4. Linking [20 points]

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

NOTE: Line numbers are provided only for the purposes of answering this question and they are NOT a part of the source files.

`fact.h`

```
=====
   1. #ifndef FACT_H
   2. #define FACT_H
   3. extern unsigned long long int fact(unsigned long int);
   4. #endif
```

fact.c

```
=====
   1.    #include "fact.h"
   2.
   3.    extern int g_num_ops;
   4.
   5.    unsigned long long int fact(unsigned long int n)
   6.    {
   7.         unsigned long long int result = 1;
   8.         g_num_ops++;
   9.         while (n > 1) {
   10.            result *= n;
   11.            --n;
   12.        }
   13.        return result;
   14.    }
```

main.c

```
======
   1.     #include "fact.h"
   2.
   3.     #define N 10
   4.
   5.     int g_num_ops = 0;
   6.
   7.     int main()
   8.     {
   9.        unsigned long long int fact_res[10];
   10.       int i;
   11.       for (i = 0; i < N; ++i) {
   12.           fact_res[i] = fact(i);
   13.       }
   14.   }
```

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

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

Questions:

A. What are the files in which the variable g_num_ops is **declared**?

B. What are the files in which the variable g_num_ops is **defined**?

C. Which variables in **fact.c** need **relocation**?

D. Which variables in **main.c** need **relocation**?

E. What will happen if the variable **g_num_ops** in main.c is made **static**?

F. What type of object file is **a.out** ?
a.out is generated using the command: `gcc fact.c main.c -m32`

   i. **Relocatable Object File**

   ii. **Executable Object File**

G. Which variables in **main.c** are stored in the **data segment**?

H. Which variables in **fact.c** are stored in the **data segment**?

I. Which variables in **main.c** are stored in the runtime **stack segment**?

J. Which variables in **fact.c** are stored in the runtime **stack segment**?

5. Virtual Memory [20 points]

Consider the **page table** (part of it), the state of 4-entry fully associative TLB and the assumptions about memory organization as shown in tables below.

PAGE TABLE

VPN	**PTE**	**VPN**	**PTE**
0XFF	0X0712	…	…
0XFE	0X0F33	0X0F	0X0F0D
0XFD	0X0314	0X0E	0X0701
0XFC	0X0728	0X0D	0X0F1B
0XFB	0X0737	0X0C	0X0F22
0XFA	0X0712	0X0B	0X071A
0XF9	0X0727	0X0A	0X0A06
0XF8	0X0039	0X09	0X0A3E
0XF7	0X071F	0X08	0X0F34
0XF6	0X0F29	0X07	0X0613
0XF5	0X0070A	0X06	0X0017
0XF4	0X0F0F	0X05	0X0738
0XF3	0X071E	0X04	0X0F2D
0XF2	0X0604	0X03	0X060E
0XF1	0X031D	0X02	0X0016
0XF0	0X072B	0X01	0X0F10
…	…	0X00	0X0608

TABLE 1

ASSUMPTIONS
64 KB physical address space
64 KB virtual address space
256 byte pages
Fully associative TLB

TLB		
VPN	**PTE**	**Valid**
0X08	0X0F34	1
0XFA	0X0F02	0
0X14	0X073A	1
0X0E	0X0701	1

A **page table entry (PTE)** is **16 bits** with following break-up:
PTE[7:0] - physical page number (PPN)
PTE[8] - valid bit (V)
PTE[9] - read permission (R)
PTE[10] - write permission (W)
PTE[11] - Supervisor mode (S)
PTE[15:12] - always zero

A. How many bits are required for **virtual page number (VPN)** for:
   a. the assumptions in table 1
   b. the assumptions in table 1, but instead of 64KB physical address space, we have 128KB physical address space
   c. the assumptions in table 1, but instead of 64KB virtual address space, we have 32KB virtual address space

B. Given the assumptions in table 1, how many pages does the entire page table occupy?

C. If possible, convert the following **virtual addresses (VA)** into their corresponding **physical addresses (PA)**.

Mark the access as:
Page Fault - if the page is on disk, but not in main memory
Illegal Access- if the access does not have desired permission

Assume the process runs in **user mode** and there are **no unallocated pages** for this process.

Mark only one box for each access.

Access	Physical Address	Page Fault	Illegal Access
Read byte VA=0x04FB			
Write byte VA=0xFA80			
Write byte VA=0x1406			
Read byte VA=0x0220			
Read byte VA=0xF1F2			
Read byte VA=0x01F0			

D. Assume the TLB state shown in the table above when processor issues a read request to virtual address 0xF00D. Modify the TLB state to what it looks like after this access is successful. Make the change to TLB diagram above, not below.

6. Signals [10 points]

The following faulty piece of code tries to count the number of times the Ctrl+C key is pressed by the user:

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

int counter = 0;

void SIGINT_handler(int signo)
{
    if(signo == SIGINT)
    {
        counter++;
        sleep(10);
    }
}

int main (void)
{
    struct sigaction sigint_action;
    memset(&sigint_action, 0, sizeof(sigint_action));
    sigint_action.sa_handler = SIGINT_handler;
    sigaction(SIGINT, &sigint_action, NULL);

    printf("Entering while loop\n");

    while(1)
    {
        //Do nothing
    }

    return 0;
```

```
}
```

A user starts the program and waits until the statement "`Entering while loop`" is printed out and then starts pressing Ctrl+C 5 times within the next 5 seconds.

(a) What is the value of `counter` 50 seconds after the statement "`Entering while loop`" is printed?

(b) Explain why that value is lower or higher than expected?

7. Cache Miss Rate Analysis [10 points]

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 4×4 array of pixels. The machine you are working on has a 32 bytes direct-mapped cache with 16-byte lines. The C structures you are using are as follows:

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

struct pixel buffer[4][4];
int i, j;
```

Assume the following:

- sizeof(char) == 1 and sizeof(int) == 4.
- buffer begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array buffer.
- Variables i and j are stored in registers.

A. What percentage of writes in the following code will miss in the cache?

```
for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++){
            buffer[i][j].r = 0;
            buffer[i][j].g = 0;
            buffer[i][j].b = 0;
            buffer[i][j].a = 0;
      }
}
```

B. What percentage of writes in the following code will miss in the cache?

```
for (j = 0; j < 4; j++) {
      for (i = 0; i < 4; i++){
            buffer[i][j].r = 0;
            buffer[i][j].g = 0;
            buffer[i][j].b = 0;
            buffer[i][j].a = 0;
      }
}
```

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. Allocator uses **best-fit policy** for allocating new blocks.
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 in FIGURE 2 **after** a call to `char *p = malloc(4);` is executed. i.e. 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.

What is the value of **pointer variable p** if the call to malloc succeeds?

FIGURE 1
(BEFORE call to malloc)

Memory Address	Value in Memory
0x804901c	0xFE670031
0x8049018	0x00000008
0x8049014	0x12CD00AB
0x8049010	0x35670011
0x804900c	0xA1B2C3D4
0x8049008	0x00000010
0x8049004	0x12345678
Start of heap => 0x8049000	0x00000009

FIGURE 2
(AFTER call to malloc)

Memory Address	Value in Memory
0x804901c	
0x8049018	
0x8049014	
0x8049010	
0x804900c	
0x8049008	
0x8049004	
Start of heap => 0x8049000	

Good luck! :)