Full Name:_______________________________________

COMP2310/COMP6310, 2022

Final Exam

November 14, 2022

Instructions:

- **Make sure you commit and push to Gitlab after you finish each question.**
- **Read each question carefully, even if the wording is similar to practice questions.**
- Each question has a sub-folder in the exam repository and a text file for each part of the question. Please input your solutions (only) in the provided text files for each question. If you need to attach a diagram, please add it to that question's relevant folder. You can use any standard image format or a PDF for your diagrams.
- The exam has a maximum score of 85 points (roughly 2 minutes per point).
- The problems vary in difficulty scale. The point value of each problem is indicated. Pile up the easy points quickly and then return to the harder problems.
- This is an open book exam. You may use any books or notes you like. Good luck!

Question	Score
1 (08):	
2 (08):	
3 (07):	
4 (16):	
5 (12):	
6 (10):	
7 (06):	
8 (18):	
TOTAL (85):	

Problem 1. (8 points):

Consider the following IA-32 (32-bit x86) assembly code for a function called foo. Assume the same assembly syntax (AT&T) and calling conventions (CDECL) presented in the lectures.

```
foo:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%edx
        cmpl 8(%ebp),%edx
        jne .L1
        xorl %eax,%eax
        jmp .L2
.L1:
        pushl 12(%ebp)
        movl 8(%ebp),%eax
        incl %eax
        pushl %eax
        call foo
        addl $8,%esp
        addl 8(%ebp),%eax
.L2:
        popl %ebp,%esp
        leave
        ret
```

1. How many arguments does foo take? Assume all arguments passed to the function are used in its body.
2. Rewrite the function in C.
3. Give a short (1-2 sentence) description of what this function computes.

Problem 2. (8 points):

The following problem concerns basic cache lookups.

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are 13 bits wide.
- The cache is 2-way set associative, with a 4 byte line size and 16 total lines.

In the following tables, **all numbers are given in hexadecimal**. The contents of the cache are as follows:

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	09	1	86	30	3F	10	00	0	99	04	03	48
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	2F	81	FD	09	0B	0	8F	E2	05	BD
3	06	0	3D	94	9B	F7	32	1	12	08	7B	AD
4	C7	0	06	78	07	C5	47	0	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	B7	26	2D	F0	0	0C	71	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

Part 1

1. What is the total capacity of the cache above including all the meta-data. Meta-data is all the bits that are not part of the actual data bytes. One example of meta-data is the tag. Write text with some detail so we know how you did it.

2. How many tag bits do we need for the above cache if we increase the set associativity to 16 without changing the line count? Write text with some detail so we know how you did it.

Part 2

For the given physical address, indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs. Note that associativity is 2.

If there is a cache miss, enter "-" for "Cache Byte returned".

Physical address: `0x08F1`

1. Physical address format (one bit per box)

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

2. Physical memory reference

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

Problem 3. (7 points):

Consider a 2-way set associative cache of size 32KB with block size of 16 bytes. Furthermore, the cache is write-back and write-allocate. You will calculate the miss rate for the following code using this cache. Remember that `sizeof(int) == 4`. Assume that the cache starts empty and that local variables and computations take place completely within the registers and do not spill onto the stack.

Part 1

Now consider the following code to copy one matrix to another. Assume that the `src` matrix starts at address 0 and that the `dest` matrix follows immediately follows it.

```
void copy_matrix(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (i=0; i<ROWS; i++) {
        for (j=0; j<COLS; j++) {
            dest[i][j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 64` and `COLS = 64`?
   Miss rate = ___________%
2. What is the cache miss rate if `ROWS = 64` and `COLS = 128`?
   Miss rate = ___________%

For each answer, you should either write an explanation or draw a diagram showing your reasoning.

Part 2

Now consider the following code for copying one matrix to another.

```
void copy_matrix(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (j=0; j<COLS; j++) {
        for (i=0; i<ROWS; i++) {
            dest[i][j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 64` and `COLS = 64`?
   Miss rate = ___________%

You should either write an explanation or draw a diagram showing your reasoning.

Problem 4. (16 points):

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to 4-byte words.
- Virtual addresses are 20 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way set associative with 16 total entries.

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

TLB			
Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	20	4	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	10	6	0
01	8	1	11	7	0
02	9	1	12	4	0
03	A	1	13	3	0
04	6	0	14	D	0
05	3	0	15	B	0
06	1	0	16	9	0
07	8	1	17	6	0
08	2	0	18	C	1
09	3	0	19	4	1
0A	1	1	1A	F	0
0B	6	1	1B	2	1
0C	A	1	1C	0	0
0D	D	0	1D	E	1
0E	E	0	1E	5	1
0F	D	1	1F	3	1

Part 1

1. How many total entries are there in the page table?
2. What is the maximum physical page count for this system?
3. Consider the page table for this question. We see that at least two virtual pages are mapped to the same physical page # 8. Is this scenario feasible? If so, provide one example. If not, explain why.
4. Which virtual address bits are used for the TLB index and tag? Explain why the specific bits are used.
5. Why does the virtual page size match the physical page size in all modern systems?

Part 2

For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a page fault, enter "-" for "PPN" and leave part 3 (the physical address format) blank.

Virtual address: `0x12A58`

1. Virtual address format (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

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

3. Physical address format (one bit per box)

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

Virtual address: `0x8147C`

1. Virtual address format (one bit per box)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Address translation

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

3. Physical address format (one bit per box)

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

Problem 5. (12 points):

The following problems refer to a file called `alphabets.txt`, with contents the ASCII string `abcdefghijkl`. You may assume calls to read() are atomic with respect to each other. The following file, read_and_print_one.h, is compiled with the code file on the next page.

```
#ifndef READ_AND_PRINT_ONE
#define READ_AND_PRINT_ONE
#include <stdio.h>
#include <unistd.h>
static inline void read_and_print_one(int fd)
{
    char c;
    read(fd, &c, 1);
    printf("%c", c); fflush(stdout);
}
#ENDIF
```

Part 1

Draw the descriptor tables, open file tables, and v-node tables for the following code (see next page) after the execution of line A and line B as marked by comments in the code. Note that we are mainly interested in the file descriptor(s), file position(s), and the relationship between the different tables.

Part 2

List all possible outputs of the following code (see next page). You can assume that if the OS schedules the child first after fork(), then the child executes all statements before the OS schedules the parent again.

```
#include "read_and_print_one.h"
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int file1;
    int file2;
    int file3;
    int pid;
    file1 = open("alphabets.txt", O_RDONLY);
    file3 = open("alphabets.txt", O_RDONLY);
    file2 = dup2(file3, file2); // LINE A
    read_and_print_one(file1);
    read_and_print_one(file3);
    pid = fork();
    if (!pid) {
        read_and_print_one(file2);
        close(file2);
        file2 = open("numbers.txt", O_RDONLY); // LINE B
        read_and_print_one(file2);
    }
    else {
        read_and_print_one(file3);
        wait(NULL);
        read_and_print_one(file2);
        read_and_print_one(file1);
    }
    read_and_print_one(file3);
    return 0;
}
```

Problem 6. (10 points):

This problem concerns sequential consistency in multicore processors. Consider three processes executing instructions on a shared-memory multicore processor. They share the variables a, b, and c. Assume the three variables are initialized to 0 at the start.

```
X1:
    a = 2;
    print(b,c);

X2:
    b = 2;
    print(a,c);
X3:
    c = 2;
    print(a,b);
```

Assuming sequential consistency, out of the outputs listed below, identify the valid outputs of the above program.

Give a trace for each valid output, and for each invalid output explain why.

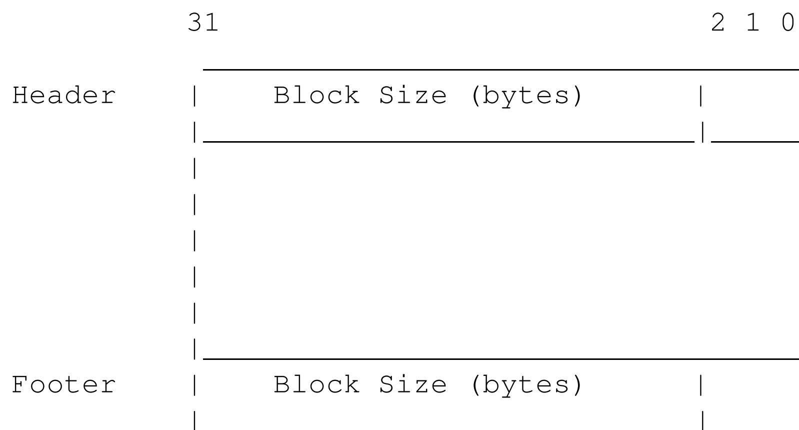
A. 002022 B. 202022 C. 000222

D. 020222 E. 222200

Problem 7. (6 points):

The following problem concerns dynamic storage allocation.

Consider an allocator that uses an implicit free list. The layout of each allocated and free memory block is as follows:



Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The usage of the remaining 3 lower order bits is as follows:

- `bit 0` indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- `bit 1` indicates the use of the current block: 1 for allocated, 0 for free.
- `bit 2` is unused and is always set to be 0.

Given the contents of the heap shown on the left of the next page, a call to `free(0x400b020)` is executed. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.

Is the argument passed to the the call valid? If the argument is not valid, then explain. If the argument is valid, then show the new contents of the heap in the right table. Write your answers in the question file as hex values corresponding to data in memory locations (1)-(6) after the free call has completed. If any memory was modified, explain why.

Address		Address	
0x400b028	0x00000012	0x400b028	**(1)**
0x400b024	0x400b611c	0x400b024	0x400b611c
0x400b020	0x400b512c	0x400b020	0x400b512c
0x400b01c	0x00000012	0x400b01c	**(2)**
0x400b018	0x00000021	0x400b018	**(3)**
0x400b014	0x400b511c	0x400b014	0x400b511c
0x400b010	0x40000021	0x400b010	0x40000021
0x400b00c	0x00000013	0x400b00c	**(4)**
0x400b008	0x00000021	0x400b008	**(5)**
0x400b004	0x400b601c	0x400b004	0x400b601c
0x400b000	0x400b511c	0x400b000	0x400b511c
0x400affc	0x00000021	0x400affc	**(6)**

Problem 8. (18 points):

The following problem concerns storage I/O and your understanding of disk organization.

Part 1

A disk file `db.txt` contains a key-value database. Keys and values are both strings. Keys refer to persons, and values are persons' attributes (e.g., city, date of birth, highest education). Keys have a fixed size of 32 bytes (characters), including the null terminator. An integer follows a key (`sizeof(int) == 4`), and it represents the size of the value corresponding to the key. The value follows the size. Key-value pairs are stored as a sequential log. See the next page for a visual representation of the database file.

Your task is to write a C program using memory-mapped file I/O (MMIO) that splits the database into two files for storing keys and values, separately. Specifically, your program should do the following:

- Maps the key-value database file (`/home/dummypath/db.txt`) into the virtual address space.
- Create two new files called keys (`/home/dummypath/keys.txt`) and values (`/home/dummypath/values.txt`).
- Using MMIO, store in `keys.txt` all keys sequentially, reading one-by-one from the database file. Each key is followed by an offset. The offset points to the value corresponding to each key. The value is stored in `values.txt`.
- Using MMIO, store in `values.txt` all values (sequentially) corresponding to keys in the database.

See the next page for a visual representation of `keys.txt` and `values.txt`.

Please note the following.

- Your C code should be roughly correct and minor programming errors will be forgiven. However, please ensure to use the correct arguments to system-level functions.
- You should not use syscall I/O.
- The inner structure and schema of value strings (e.g., where each attribute begins and ends) is irrelavent to your code.

Write the code in the provided C file in the question sub-folder. We have already added the header files, and defined some constants for your convenience.

Part 2

How many 2MB records can be stored on a disk with 3 platters, 20,000 cylinders, 500 sectors per track, and 512 bytes per sector? Assume the filesystem-related meta-data overhead is accounted for as part of the 2MB size of each record. Provide some explanation for your answer so we know how you did it.

```
db.txt
+------+-------+--------+------+-------+--------+------+-------+--------+
| key1 | size1 | value1 | key2 | size2 | value2 | key3 | size3 | value3 |...
+------+-------+--------+------+-------+--------+------+-------+--------+

keys.txt
+------+---------+------+---------+------+---------+
| key1 | offset1 | key2 | offset2 | key3 | offset3 |...
+------+---------+------+---------+------+---------+

values.txt
+-------------+-------------+-------------+
| value1      | value2      | value3      |...
+-------------+-------------+-------------+
offset1       offset2       offset3
```