Full Name:_______________________________________

COMP2310/COMP6310, 2022

Practice Exam

Semester 2, 2023

Instructions:

- The exam has a maximum score of 50 points.
- The problems vary in difficulty scale. The point value of each problem is indicated. Advice: Pile up the easy points quickly and then come back to the harder problems.

  Good luck!

1 (05):
2 (05):
3 (14):
4 (16):
5 (08):
6 (02):
TOTAL (50):

Problem 1. (5 points):

Consider the source code below, where `M` and `N` are constants declared with `#define`.

```
int array1[M][N];
int array2[N][M];

int copy(int i, int j)
{
    array1[i][j] = array2[j][i];
}
```

Suppose the above code generates the following assembly code:

```
copy:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
  movl 8(%ebp),%ecx
  movl 12(%ebp),%ebx
  leal (%ecx,%ecx,8),%edx
  sall $2,%edx
  movl %ebx,%eax
  sall $4,%eax
  subl %ebx,%eax
  sall $2,%eax
  movl array2(%eax,%ecx,4),%eax
  movl %eax,array1(%edx,%ebx,4)
  popl %ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

What are the values of `M` and `N`?

`M` = **15**

`N` = **9**

Problem 2. (5 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	1	06	78	07	C5	05	1	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

The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO	The block offset within the cache line
CI	The cache index
CT	The cache tag

12	11	10	9	8	7	6	5	4	3	2	1	0
CT	**CT**	**CT**	**CT**	**CT**	**CT**	**CT**	**CT**	**CI**	**CI**	**CI**	**CO**	**CO**

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.

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

Physical address: `0E34`

A. Physical address format (one bit per box)

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

B. Physical memory reference

Parameter	Value
Byte offset	0x**0**
Cache Index	0x**5**
Cache Tag	0x**71**
Cache Hit? (Y/N)	**Y**
Cache Byte returned	0x**0B**

Problem 3. (14 points):

Consider a direct mapped cache of size 64K 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.

A. 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 = 128` and `COLS = 128`?
   Miss rate = **100%**
2. What is the cache miss rate if `ROWS = 128` and `COLS = 192`?
   Miss rate = **25%**
3. What is the cache miss rate if `ROWS = 128` and `COLS = 256`?
   Miss rate = **100%**

B. Now consider the following two implementations of a horizontal flip and copy of the matrix. Again assume that the `src` matrix starts at address 0 and that the `dest` matrix follows immediately follows it.

```
void copy_n_flip_matrix1(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][COLS - 1 - j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?
   Miss rate = **25%**

2. What is the cache miss rate if `ROWS = 128` and `COLS = 192`?
   Miss rate = **25%**

```
void copy_n_flip_matrix2(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][COLS - 1 - j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?
   Miss rate = **25%**

2. What is the cache miss rate if `ROWS = 192` and `COLS = 128`?
   Miss rate = **75%**

Problem 4. (16 points):

This problem tests your understanding of exceptional control flow in C programs. Assume we are running code on a Unix machine. The following problems all concern the value of the variable `counter`.

Part I (6 points)

```
int counter = 0;

int  main()
{
    int i;

    for (i = 0; i < 2; i ++){
        fork();
        counter ++;
        printf("counter = %d\n", counter);
    }

    printf("counter = %d\n", counter);
    return 0;
}
```

A. How many times would the value of `counter` be printed: **10**

B. What is the value of `counter` printed in the first line? **1**

C. What is the value of `counter` printed in the last line? **2**

Part II (6 points)

```
pid_t pid;
int counter = 0;

void handler1(int sig)
{
    counter ++;
    printf("counter = %d\n", counter);
    fflush(stdout);   /* Flushes the printed string to stdout */
    kill(pid, SIGUSR1);
}

void handler2(int sig)
{
    counter += 3;
    printf("counter = %d\n", counter);
    exit(0);
}

main() {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while(1) {};
    }
    else {
        pid_t p; int status;
        if ((p = wait(&status)) > 0) {
            counter += 2;
            printf("counter = %d\n", counter);
        }
    }
}
```

What is the output of this program?

counter = 1

counter = 3

counter = 3

Part III (4 points)

```
int counter = 0;

void handler(int sig)
{
    counter ++;
}


int  main()
{
    int i;

    signal(SIGCHLD, handler);

    for (i = 0; i < 5; i ++){
        if (fork() == 0){
            exit(0);
        }
    }

    /* wait for all children to die */
    while (wait(NULL) != -1);

    printf("counter = %d\n", counter);
    return 0;
}
```

A. Does the program output the same value of `counter` every time we run it? **No**

B. If the answer to A is `Yes`, indicate the value of the `counter` variable. Otherwise, list all possible values of the `counter` variable.

Answer: `counter` = **1,2,3,4,5**

Dynamic storage allocation

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:

```
           31                            2  1  0
            __________________________________
Header     |    Block Size (bytes)       |     |
           |_____________________________|_____|
           |                                   |
           |                                   |
           |                                   |
           |                                   |
           |                                   |
           |___________________________________|
Footer     |    Block Size (bytes)       |     |
           |_____________________________|_____|
```

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

Problem 5. (8 points):

Given the contents of the heap shown on the left, show the new contents of the heap (in the right table) after a call to `free(0x400b010)` is executed. Your answers should be given as hex values. 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.

Address	
0x400b028	0x00000012
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	0x00000012
0x400b018	0x00000013
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000013
0x400b008	0x00000013
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	0x00000013

Address	
0x400b028	**0x00000022**
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	**0x00000012**
0x400b018	**0x00000013**
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	**0x00000022**
0x400b008	**0x00000013**
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	**0x00000013**

Problem 6. (2 points):

Consider the following function `func` that it is run concurrently on two threads. There is a global array `state` that each thread will update to indicate it is ready to continue. Both threads will wait until both entries in `state` are 1 before continuing. You may assume that `tid` contains the id of the thread, and that all locks and threads have been initialised correctly.

```
int state[2] = {0, 0};
pthread_mutex_t locks[2];

void func(int tid) {
  int other_tid = (tid - 1) & 1;
  int ready = 0;
  pthread_mutex_lock(&locks[tid]);
  state[id] = 1;
  while (ready == 0) {
    pthread_mutex_lock(&locks[other_tid]);
    ready = state[other_id];
    pthread_mutex_unlock(&locks[other_tid]);
  }
  pthread_mutex_unlock(&locks[tid]);
  continue();
}
```

1. Briefly explain why this code will always result in a deadlock.

   It deadlocks because both threads will try to lock `locks[other_tid]` before unlocking `locks[tid]`.

2. How could you re-order the statements in `func` to avoid this deadlock?

   Fix it by moving the second unlock before the while loop.