Full Name:_____

COMP2310/COMP6310, 2022

Final Exam

November 14, 2022

Instructions:

- **Make sure that you push your final exam solution via Gitlab.**
- There is a sub-folder for each question and a text file for each part of the question in the exam repository. Please input your solutions (only) in the provided text files for each question. If you need to attach a diagram, please add it in the relavent folder for that question. You can use any common image format or a Pdf for your diagrams.
- The exam has a maximum score of 85 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.
- This exam is OPEN BOOK. You may use any books or notes you like. Good luck!

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

Problem 1. (20 points):

Multiple choice and Ture/False questions on a variety of topics. Pick one correct answer. To receive credit, clearly indicate/circle your choice for the correct answer.

1. Which of the following is true about TLB misses on an x86-64 system?
   (a) TLB miss occurs when the data CPU requests is not found in the L1-D cache.
   (b) TLB miss is handled by the kernel.
   (c) TLB miss sometimes results in page fault.
   (d) TLB miss is not a performance concern.

2. Which of the following is true about races?
   (a) A race occurs when correctness of the program depends on one thread reaching point a before another thread reaches point b.
   (b) Exclusive access to all shared resources eliminates race conditions.
   (c) Race conditions are the same as deadlocks.
   (d) All race conditions occur inside loops, since that is the only way we can interleave processes.

3. Each thread has its own:
   (a) Text data
   (b) Global values
   (c) Stack
   (d) Heap

4. Which of the following is the correct ordering (left-to-right) of a file's compilation cycle (a filename with no extension is an executable):
   (a) foo.c $\rightarrow$ foo.o $\rightarrow$ foo.s $\rightarrow$ foo
   (b) foo $\rightarrow$ foo.s $\rightarrow$ foo.o $\rightarrow$ foo.c
   (c) foo.c $\rightarrow$ foo.s $\rightarrow$ foo $\rightarrow$ foo.o
   (d) foo.c $\rightarrow$ foo.s $\rightarrow$ foo.o $\rightarrow$ foo

5. How many sets does a 4-way set-associatve cache with 256 byte capacity and 32 byte block size has?
   (a) 1
   (b) 2
   (c) 4
   (d) 8

6. Which of the following sentences about reader-writer locks is not true?

   (a) Many readers can hold the same rwlock at the same time.
   (b) Two writers cannot hold the same rwlock at the same time.
   (c) Many readers and exactly one writer can hold the same rwlock at the same time.
   (d) An rwlock can be used as a mutex.

7. When can short counts occur?

   (a) When an EOF is encountered during a read.
   (b) When a short int is used as a counter.
   (c) When reading or writing to disk files.
   (d) When the kernel runs out of kernel memory.

8. Which of the following is true about the page cache?

   (a) Page cache is managed by user-level software.
   (b) Page cache is exclusively used for memory-mapped files.
   (c) Page cache cannot be bypassed by user software.
   (d) Page cache amortizes the cost of accessing disks.

9. We use dynamic memory because:

   (a) The heap is significantly faster than the stack.
   (b) The stack is prone to corruption from buffer overflows.
   (c) Storing data on the stack requires knowing the size of that data at compile time.
   (d) Heap allows allocating data that has a lifetime not related to the execution of the current routine.

10. Consider the following segment of network code. Which of the following is true about `read()`?

```
fd = socket (AF_INET, SOCK_STREAM, 0) //return success
...
connect (fd, &serveraddr, sizeof(serveraddr)); //return success
write (fd, data, N); //return success
read (fd, buf, N);
...
```

   (a) the read() call is guaranteed to return quickly.
   (b) the read() call is guaranteed to never return.
   (c) the read() call is not guaranteed to return quickly.
   (d) the read() call is guaranteed to result in a program crash.

11. Which of the following is true about signals?

    (a) Signals can be used to count events because the kernel efficiently enqueues them.
    (b) A signal is sent to a process when a new connection arrives on a listening socket.
    (c) A signal handler is a separate logical flow that runs concurrently with the main program.
    (d) Kernel blocks pending signals of all types except the one currently being handled.

12. Which of the following is not true about physical pages?

    (a) A physical page has the same size as the virtual page.
    (b) Physical addresses have a one-to-one mapping to virtual addresses.
    (c) Two virtual pages in the same process can be mapped to the same physical page.
    (d) Two virtual pages in different processes can be mapped to the same physical page.

13. Multi-level page tables are used because:

    (a) They are space-efficient.
    (b) They provide faster address translation than single-level page tables.
    (c) They reduce the pressure on TLB.
    (d) They can be shared among multiple processes.

14. What is the maximum amount of physical memory a server with 44 bit physical addresses can have?

    (a) 64 GB
    (b) 256 GB
    (c) 4 TB
    (d) 16 TB

15. What is the approximate capacity of a hard disk with 512 byte sectors, 100 sectors per track, 10,000 tracks per surface, and one platter?

    (a) 512 MB
    (b) 1 GB
    (c) 2 GB
    (d) 10 GB

16. I/O multiplexing cannot fully utilize multicore processors.

    (a) True
    (b) False

17. Processes are more space-efficient for the kernel to manage than threads.

    (a) True
    (b) False

18. HTTP is a text-based kernel-level protocol.

    (a) True
    (b) False

19. TCP and UDP are both connection-oriented protocols.

    (a) True
    (b) False

20. Garbage collection can have an impact on memory locality.

    (a) True
    (b) False

Problem 2. (10 points):

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

```
int array1[M][N];
int array2[N][M];
void copy_array(int i, int j) {
array2[i][j] = array1[j][i];
}
```

Suppose the above C code generates the following x86-64 assembly code:

```
# On entry:
# %edi = i
# %esi = j
#
copy_array:
movslq %esi,%rsi
movslq %edi,%rdi
movq %rdi, %rax
salq $3, %rax
addq %rdi, %rax
addq %rsi, %rax
leaq (%rsi,%rsi,2), %rsi
leaq (%rdi,%rsi,4), %rsi
movl array1(,%rsi,4), %edx
movl %edx, array2(,%rax,4)
ret
```

What are the values of M and N?

1. M =
2. N =

Annotate the above code with comments next to each assembly statement (e.g., `rax=i`) to receive partial credit.

Problem 3. (10 points):

Consider the C program below. (For simplicity assume all calls to `fork` and `wait` succeed and that writes to standard output are unbuffered). Please use the white space on this page to draw the process graph.

```
void foo() {
  if (fork() == 0) {
    if (fork() == 0)
      printf("A");
    else
      wait(NULL);
    printf("B");
  } else {
    wait(NULL);
    if (fork() != 0) {
      printf("C");
      wait(NULL);
    } else {
      printf("D");
    }
  }
  printf("E");
}
```

A. Draw the process graph for this program.

B. List all the possible different outputs of this program.

Problem 4. (10 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	06	06	06	37
2	EB	0	2F	81	FD	09	0B	0	8F	E2	05	BD
3	06	0	3D	94	9B	F7	32	1	32	08	7B	AD
4	C7	0	06	06	06	06	47	0	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	32	26	2D	F0	1	32	32	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

For the given cache byte returned on a cache hit, fill in the missing bits of the physical addresses used to access the cache below. Write your answers (0 or 1) in each of the boxes below. Two one-bit hints are provided for each cache byte returned.

1. **Cache byte returned**: `06`

   Physical Address (one bit per box)

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

2. **Cache byte returned**: `32`

   Physical Address (one bit per box)

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

Problem 5. (10 points):

Consider a direct mapped cache of size 256 bytes with block size of 32 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.

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 = 4` and `COLS = 4`?
   Miss rate = _____%
2. What is the cache miss rate if `ROWS = 8` and `COLS = 8`?
   Miss rate = _____%
3. What is the cache miss rate if `ROWS = 8` and `COLS = 12`?
   Miss rate = _____%

Problem 6. (10 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:

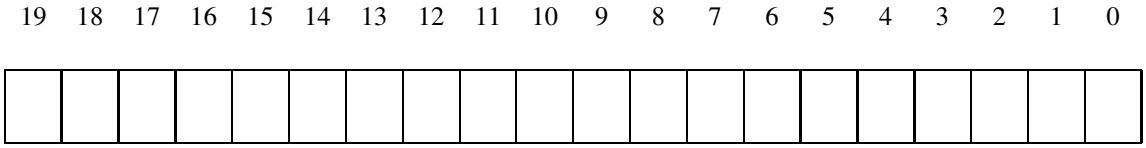
TLB			
Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	20	4	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

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	0	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

A. Part 1

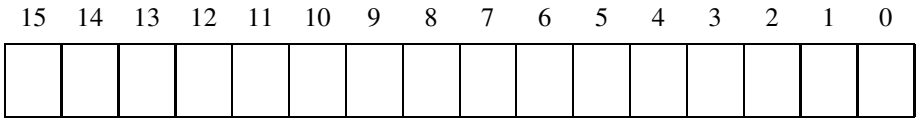
(a) The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO The virtual page offset
VPN The virtual page number
TLBI The TLB index
TLBT The TLB tag



(b) 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:

PPO The physical page offset
PPN The physical page number



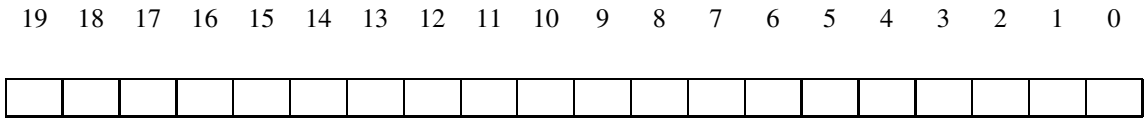
B. Part 2

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 C blank.

Virtual address: `0x07A50`

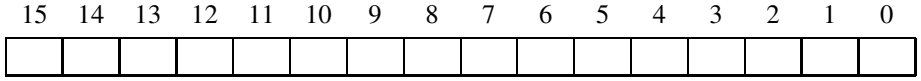
(a) Virtual address format (one bit per box)



(b) Address translation

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

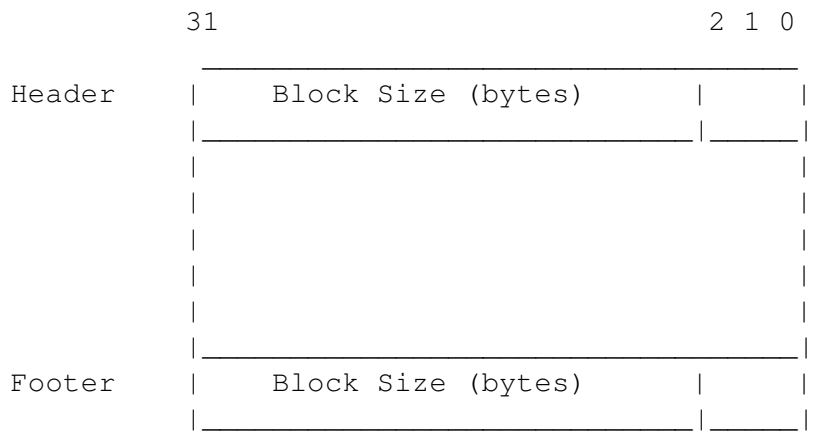
(c) Physical address format (one bit per box)



Problem 7. (10 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 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.

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(0x500b010)` 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	
0x500b028	0x00000013
0x500b024	0x500b1008
0x500b020	0x500b512c
0x500b01c	0x00000013
0x500b018	0x00000011
0x500b014	0x500b511c
0x500b010	0x500b601c
0x500b00c	0x00000011
0x500b008	0x00000012
0x500b004	0x500b601c
0x500b000	0x500b4024
0x500affc	0x00000012

Address	
0x500b028	
0x500b024	0x500b1008
0x500b020	0x500b512c
0x500b01c	
0x500b018	
0x500b014	0x500b511c
0x500b010	
0x500b00c	0x00000011
0x500b008	0x00000012
0x500b004	0x500b601c
0x500b000	0x500b4024
0x500affc	

Problem 8. (10 points):

The following problem concerns memory-mapped I/O (mmio).

A disk file contains a key-value database. Keys are strings that identify persons and values are persons' attributes (e.g., city, date-of-birth, highest education). You can assume keys have a fixed size of 32 bytes (characters), including the null terminator. A key is followed by an integer (`sizeof(int) == 4`) that represents the size of the value corresponding to the key. The size is followed by the value. See below for the visual representation of the database file.

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

Your task is to write a C function called `query_database(char* db, char* key)`. The function takes two arguments (1) a string representing the pathname of the key-value database file and (2) a string representing the (query) key. The function returns an integer that represents the size of the next key (sequentially) stored in the database file.

You will need the following system-level and string comparison function descriptions.

- `int open (const char *name, int flags);`
- ```
  void *mmap(void addr, size_t length, int prot,
                    int flags, int fd, off_t offset);
  ```
- `int strcmp (const char* str1, const char* str2);`

We have already written the first few statements for the function implementation on the next page. Use **memory mapped file I/O** to map the database file into the virtual address space. Then write the remaining code for finding the (query) key and returning the proper integer value.

Write your code on the next page.
  ```

```
int query_database(char *db, char* key) {

  struct stat stat;
  int fd, size;

  fd = open(db, O_RDONLY);
  if (fd < 0 || fstat(fd, &stat) < 0) {
    fprintf(stderr, "open() or fstat() failed.\n");
    return (1);
  }
  size = stat.st_size;

  // write your code below this comment









}
```

Problem 9. (10 points):

Consider the following code. The intended behaviour of this program is to create four threads that each print out the value in `vargp`. Each thread should print out a distinct value, but the order in which threads print out their value does not matter. For simplicity, you may assume all calls to `pthread` functions and `printf` succeed.

```
#define N 4
pthread_mutex_t lock;
int array[N] = {0, 0, 0, 0};

void *foo(void *vargp) {
  int id = *((int *) vargp);              // f1
  pthread_mutex_lock(&lock);              // f2
  printf("id: %d\n", id);                 // f3
  pthread_mutex_unlock(&lock);            // f4
  return NULL;                            // f5
}

int main(void) {
  pthread_t tids[N];
  int i;
  pthread_mutex_init(&lock, NULL);
  for (i = 0; i < N; i++)                       // m1
    pthread_create(&tids[i], NULL, foo, &i);    // m2
  for (i = 0; i < N; i++)                       // m3
    pthread_join(tids[i], NULL);                // m4
  pthread_mutex_destroy(&lock);
  return 0;
}
```

A. Briefly explain why this program will not behave correctly.

B. Consider the following attempt at fixing the program by modifying the `main` function. Explain why this correctly fixes the issue.

```
int main(void) {
  pthread_t tids[N];
  int i;
  pthread_mutex_init(&lock, NULL);
  for (i = 0; i < N; i++) {
    pthread_create(&tids[i], NULL, foo, &i);
    pthread_join(tids[i], NULL);
  }
  pthread_mutex_destroy(&lock);
  return 0;
}
```

C. What is a drawback of the fix suggested in part B?

D. Suggest a better way to fix this program by only modifying the `pthread_create` loop that does not have the same problem as you identified in part C.

END OF EXAMINATION