Full Name:_____

Andrew ID:_____

15-213/513

Practice Final Exam

Instructions:

- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The problems are of varying difficulty. Focus on questions you can answer first and come back to harder ones.
- The only resource you may refer to is your hand-written notes on two A4 / legal-sized papers. You may not use any electronic devices during the exam.
- If you have any questions, raise your hand and a staff member will come help.

Do not write below this line

Problem 1. (2 points):

1. Explain the differences between big endianness and little endianness.

2) What does the following code segment output. Why?

```
char c = 0xff;
unsigned int s = c;
if (s > 0x100) printf("High!\n");
else printf("Low!\n");
```

Problem 2. (1 points):

Explain what the function interesting_fn performs:

```
(gdb) disassemble interesting_fn
Dump of assembler code for function interesting_fn:
   0x0000000000401122 <+0>: mov    $0x1,%eax
   0x0000000000401127 <+5>: mov    $0x0,%edx
   0x000000000040112c <+10>: cmp    %edi,%eax
   0x000000000040112e <+12>: jg     0x401137 <interesting_fn+21>
   0x0000000000401130 <+14>: add    %eax,%edx
   0x0000000000401132 <+16>: add    $0x2,%eax
   0x0000000000401135 <+19>: jmp    0x40112c <interesting_fn+10>
   0x0000000000401137 <+21>: mov    %edx,%eax
   0x0000000000401139 <+23>: retq
End of assembler dump.
```

Problem 3. (1 points):

Given the following assembly code for a factorial function:

```
(gdb) disas factorial
Dump of assembler code for function factorial:
   0x0000000000401122 <+0>: cmp    $0x1,%edi
   0x0000000000401125 <+3>: jle    0x401137 <factorial+21>
   0x0000000000401127 <+5>: push   %rbx
   0x0000000000401128 <+6>: mov    %edi,%ebx
   0x000000000040112a <+8>: lea    -0x1(%rdi),%edi
   0x000000000040112d <+11>: callq  0x401122 <factorial>
   0x0000000000401132 <+16>: imul   %ebx,%eax
   0x0000000000401135 <+19>: pop    %rbx
   0x0000000000401136 <+20>: retq
   0x0000000000401137 <+21>: mov    $0x1,%eax
   0x000000000040113c <+26>: retq
End of assembler dump.
```

You run factorial(5). When you start running factorial(5), the value of %rsp is at 0x7fffffffdf68.

1. Draw the stack diagram when factorial(4) is called, factorial(3), and factorial(2).

2. Draw the stack diagram when factorial(2) returns.

Problem 4. (1 points):

Given the following struct

```
typedef struct yeet {
    int a;
    double b;
    char *str;
    char d;
    short e;
} yeet;
```

Is there a way to reduce the size of struct? If so, give the new size of the struct and how to build it. If not, then why?

NOTE: Please also understand the offsets from the starting address for each value within the struct.

Problem 5. (1 points):

Here is an array with 6 elements:

```
short arr[] = {0x1234, 0x8326, 0x9742, 0x4200, 0x1521, 0x3531};
```

Given the array's starting address is 0x7fffffffdf86, what do these statements print? Justify your answers.

```
printf("2: %lx\n", *((long*)&arr[2]));
printf("3: %x\n", (unsigned char)*((char*)&arr[3]));
printf("4: %x\n", (unsigned char) *(((char*)arr)+ 3));
printf("5: %x\n", ((int*)arr)[1]);
```

Problem 6. (1 points):

As a security engineer for a software company, it is your job to perform attacks against your company's software and try to break it. One of your developers, Harry Q. Bovik, has written a password validator that he thinks is unbreakable! Below is the front-end to his system.

```
int main() {
   char buffer[20];
   printf("Enter your password here: ");
   scanf("%s", buffer);
   if (validate(buffer)) {
          getOnTheBoat();
          exit(0);
   }
   printf("Sorry, you do not have access.");
   return 0;
}
```

Note that scanf reads data from standard input and places it in the buffer passed into the second argument. Briefly explain how you could attack this program with a buffer overflow.

Problem 7. (3 points):

A fellow 213 student works on cutting edge research finding prime numbers. He wants to speed up his code by making it multi-threaded. He is running into some issues while implementing a thread safe version of the **next_prime** function and asks for your help.

```
struct big_number *next_prime(struct big_number current_prime) {
    static struct big_number next;
    next = current_prime;
    addOne(next);
    while (isNotPrime(next)) {
      addOne(next);
    }
    return &next;
}

struct big_number *ts_next_prime(struct big_number current_prime) {
    return next_prime(current_prime);
}
```

1. Why is the function **ts_next_prime** thread-unsafe?

2. Assume the mutex guarding the call to **next_prime** is initialized correctly in the following code.

```
struct big_number *ts_next_prime(struct big_number current_prime) {
    struct big_number *value_ptr;
    sem_wait(&mutex);
    value_ptr = next_prime(current_prime);
    sem_post(&mutex);
    return value_ptr;
}
```

The following modification to the function is still not thread safe. Explain why and describe an example execution with two threads showing the problem.

3. Use the code below for the next three questions.

```
struct big_number *ts_next_prime(struct big_number current_prime) {
    struct big_number *value_ptr;
    struct big_number *ret_ptr = ____A____;
    sem_wait(&mutex);
    value_ptr = next_prime(current_prime);
    ____B____;
    sem_post(&mutex);
    return ret_ptr;
}
```

What is the correct fix for A?

Which of the following is the correct fix for B?

Please explain why these fixes are correct.

Problem 8. (2 points):

Consider the following code sample. You may assume that no call to ‘fork‘, ‘exec‘, ‘wait‘, or ‘printf‘ will ever fail, and that ‘stdout‘ is immediately flushed following every call to ‘printf‘.

```
int global_x = 0;
int main(int argc, char *argv[]) {
    global_x = 17;
    /* Assume fork never fails */
    if( !fork() ) {
      global_x++;
      printf("Child: %d\n", global_x);
    }
    else {
      wait(NULL);
      global_x--;
      printf("Parent: %d\n", global_x);
    }
    return 0;
}
```

1. What is printed by this program?
   - child: 18; parent: 16
   - parent: 16; child: 18
   - child: 16; parent: 18
   - child: 17; parent: 17
   - child: 16; parent: 16
2. Assume we removed the call to ‘wait‘. What is printed by this program?
   - child: 18; parent: 16
   - parent: 16; child: 18
   - child: 16; parent: 18
   - child: 17; parent: 17
   - child: 16; parent: 16

Problem 9. (3 points):

This problem tests your understanding of exceptional control flow. Consider the following program. You may assume that ‘printf‘ is unbuffered and executes atomically. The program ‘/bin/echo‘ prints its command line argument to‘stdout‘.

```
1 sigset_t s1;
2 static int count = 0;
3
4 char *argv[] = {"/bin/echo", "Hello", NULL};
5
6 pid_t pid;
7
8 void handler () {
9      printf ("Bye\n");
10 }
11
12 int main () {
13      int i = 0;
14
15      signal (SIGCHLD, handler);
16
17       sigemptyset (&s1);
18       sigaddset (&s1, SIGCHLD);
19
20       sigprocmask (SIG_BLOCK, &s1, NULL);
21
22       for (i = 0; i < 3; i++) {
23             if (fork() == 0) {
24             count++;
25             execve ("/bin/echo", argv, NULL);
26             }
27       wait(NULL);
28       }
29
30       sigprocmask (SIG_UNBLOCK, &s1, NULL);
31
32 }
```

1. What are the possible outputs of the program?

2. When the program reaches line 31, what are the possible values that count may have?

3. Consider the same code, without blocking SIGCHLD, i.e. with lines 20, and 30 removed. Would the output be similar? If you answered no, list one possible output.

Problem 10. (1 points):

The following problem refer to a file called ‘numbers.txt‘, with contents the ASCII string 0123456789. You may assume calls to ‘read()‘ are atomic with respect to each other. The following file, ‘read_and_print_one.h‘, is used in the ‘main‘ function below.

```
#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
```

Now consider the following code:

```
#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("numbers.txt", O_RDONLY);
    file3 = open("numbers.txt", O_RDONLY);
    file2 = dup2(file3, file2);
    read_and_print_one(file1);
    read_and_print_one(file2);
    pid = fork();
    if (!pid) {
      read_and_print_one(file3);
      close(file3);
      file3 = open("numbers.txt", O_RDONLY);
      read_and_print_one(file3);
    } else {
      wait(NULL);
      read_and_print_one(file3);
      read_and_print_one(file2);
      read_and_print_one(file1);
    }
    read_and_print_one(file3);
    return 0;
}
```

List all the possible outputs of the code above.

Problem 11. (3 points):

Assume a System that has

- A two way set associative TLB
- A TLB with 8 total entries
- 2^8 byte page size
- 2^{16} bytes of virtual memory

TLB			
Index	Tag	Frame Number	Valid
0	0x13	0x30	1
	0x34	0x58	0
1	0x1F	0x80	0
	0x2A	0x72	1
2	0x1F	0x95	1
	0x20	0xAA	0
3	0x3F	0x20	1
	0x3E	0xFF	0

1. Given the virtual address '0x7E85', use the table above to get the physical address. If there is not enough information to get the address, enter "Not enough information".

2. Given the physical address '0x3020', use the table above to get the virtual address. If there is not enough information to get the address, enter "Not enough information".

3. Given the virtual address '0xD301', use the table above to get the physical address. If there is not enough information to get the address, enter "Not enough information".

Problem 12. (2 points):

In this question, we will consider the utilization score of various malloc implementations on the following code:

```
#define N 64
void *pointers[N];
int i;

for (i = 0; i < N; i++) {
    pointers[i] = malloc(8);
}

for (i = 0; i < N; i++) {
    free(pointers[i]);
}

for (i = 0; i < N; i++) {
    pointers[i] = malloc(24);
}
```

1. Consider a malloc implementation that uses an implicit list with headers of size 8 bytes and no footers. In order to keep payloads aligned to 8 bytes, every block is always constrained to have size a multiple of 8. The header of each block stores the size of the block, and since the 3 lowest order bits are guaranteed to be 0, the lowest order bit is used to store whether the block is allocated or free. A first-fit allocation policy is used. If no unallocated block of a large enough size to service the request is found, 'sbrk' is called for the smallest multiple of 8 that can service the request. No coalescing or block splitting is done.

NOTE: You do NOT need to simplify any mathematical expressions. Your final answer may include multiplications, additions, and divisions.

a) After the given code sample is run, how many total bytes have been requested from 'sbrk'?

b) How many of those bytes are used for currently allocated blocks, including internal fragmentation and header information?

c) How many of those bytes are used to store free blocks, including header information?

d) Give the fraction of the total number of bytes requested by the user by the end of the trace (not including calls to malloc that have subsequently been freed) over total number of bytes allocated by 'sbrk'. You do not need to simplify the fraction.

2. Consider another malloc implementation that never calls ‘sbrk‘ for a size less than 32 bytes. In every other way the implementation is identical to the implementation in question A. Note that since no block splitting is done, this means the size of each block, including the header, will always be at least 32 bytes. Again, there is no need to simplify mathematical expressions.

a) After the given code sample is run, how many total bytes have been requested from ‘sbrk‘?

b) How many of those bytes are used for currently allocated blocks, including internal fragmentation and header information?

c) How many of those bytes are used to store free blocks, including header information?

d) Give the fraction of the total number of bytes requested by the user by the end of the trace (not including calls to malloc that have subsequently been freed) over total number of bytes allocated by ‘sbrk‘. You do not need to simplify the fraction.

Problem 13. (1 points):

Consider a computer with an 8-bit address space and a direct-mapped 64-byte data cache with 8-byte cache blocks.

How many bits will be needed to represent the block offset?

How many bits will be needed to represent the set index?

How many bits will be needed to represent the cache tag?

The table below shows a trace of load addresses accessed in the data cache. Assume the cache is initially empty. For each row in the table, please write down the values for each row in the two rightmost columns, indicating (i) the set number (in decimal notation) for that particular load, and (ii) whether that loads hits (H) or misses (M) in the cache (write either "H" or "M" accordingly).

Load No.	Hex Address	Binary Address
1	43	0100 0011
2	b2	1011 0010
3	40	0100 0000
4	f9	1111 1001
5	b2	1011 0010
6	93	1001 0011
7	d0	1101 0000
8	b0	1011 0000
9	67	0110 0111
10	07	0000 0111

Problem 14. (2 points):

In this problem you will estimate the miss rates for some C functions. Assumptions:

- 16-way set associative L1 cache (E = 16) with a block size of 32 bytes (B = 32).
- N is very large, so that a single row or column cannot fit in the cache.
- sizeof(int) == 4
- Variables i, k, and sum are stored in registers.
- The cache is cold before each function is called.

1. Given the code below, choose the closest miss rate for sum1.

```
int sum1(int A[N][N], int B[N][N]) {
    int i, k, sum = 0;
    for (i = 0; i < N; i++)
      for (k = 0; k < N; k++)
           sum += A[i][k] + B[k][i];
    return sum;
}
```

- 1/16
- 1/8
- 1/4
- 1/2
- 9/16
- 1

2. Given the code below, choose the closest miss rate for sum2.

```
int sum2(int A[N][N], int B[N][N]) {
    int i, k, sum = 0;
    for (i = 0; i < N; i++)
      for (k = 0; k < N; k++)
           sum += A[i][k] + B[i][k];
    return sum;
}
```

- 1/16
- 1/8
- 1/4
- 1/2
- 9/16
- 1

Problem 15. (1 points):

Consider a multi-threaded proxy that handles requests concurrently and a single-threaded proxy that handles requests serially.

Under which circumstances would the multi-threaded proxy perform better than the single-threaded proxy?

Under which circumstances would the single-threaded proxy perform no worse than the multi-threaded proxy?