Andrew ID (print clearly!):______________________________________

Full Name:______________________________________

15-213/18-213, Fall 2012

Final Exam

Monday, December 10, 2012

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew ID and full name on the front.
- This exam is closed book, closed notes (except for 2 double-sided note sheets). You may not use any electronic devices.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 98 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

1 (18):	
2 (06):	
3 (06):	
4 (08):	
5 (10):	
6 (12):	
7 (06):	
8 (10):	
9 (12):	
10 (10):	
TOTAL (98):	

Problem 1. (18 points):

Multiple choice questions on a variety of stimulating and refreshing topics.

To receive credit, you must write your answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10

11	12	13	14	15	16	17	18	-	-
								-	-

1. Each thread has its own __________.

   (a) Heap
   (b) Stack
   (c) Global values
   (d) Text data

2. Simply decreasing the size of block headers used internally by malloc:

   (a) Decreases internal fragmentation
   (b) Increases internal fragmentation
   (c) Decreases external fragmentation
   (d) Increases external fragmentation

3. 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

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` → `foo.o` → `foo.s` → `foo`
   (b) `foo` → `foo.s` → `foo.o` → `foo.c`
   (c) `foo.c` → `foo.s` → `foo` → `foo.o`
   (d) `foo.c` → `foo.s` → `foo.o` → `foo`

5. Suppose an `int` A is stored at virtual address `0xff987cf0`, while another `int` B is stored at virtual address `0xff987d98`. If the size of a page is `0x1000` bytes, then A's physical address is numerically less than B's physical address.

   (a) Always true
   (b) Always false
   (c) Sometimes true, sometimes false
   (d) Not enough information

6. Assuming no errors, which one of the following functions returns exactly once?

   (a) `fork()`
   (b) `execve()`
   (c) `exit()`
   (d) `longjmp()`
   (e) `waitpid()`

7. On a 64-bit system, which of the following C expressions is equivalent to the C expression `(x[2] + 4)[3]`? Assume x is declared as `int **x`.

   (a) `*((*(x + 16)) + 28)`
   (b) `*((*(x + 2)) + 7)`
   (c) `**(x + 28)`
   (d) `*(((*x) + 2) + 7)`
   (e) `(**(x + 2) + 7)`

8. 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

(more)

9. A program blocks SIGCHLD and SIGUSR1. It is then sent a SIGCHLD, a SIGUSR1, and another SIGCHLD, in that order. What signals does the program receive after it unblocks both of those signals (you may assume the program does not receive any more signals after)?

   (a) None, since the signals were blocked they are all discarded.
   (b) Just a single SIGCHLD, since all subsequent signals are discarded.
   (c) Just a single SIGCHLD and a single SIGUSR1, since the extra SIGCHLD is discarded.
   (d) All 3 signals, since no signals are discarded.

10. Which of the following events does not generate a signal?

   (a) Division by zero
   (b) A new connection arrives on a listening socket
   (c) A write is attempted on a disconnected socket
   (d) NULL is dereferenced
   (e) A process whose parent has already terminated exits

11. In an x86-64 system, how many integers can be stored in a cache line if your cache is 4KB, is 4-way set-associative, and contains 4 sets?

   (a) 8
   (b) 16
   (c) 32
   (d) 64
   (e) 128

12. Which types of locality are leveraged by virtual memory?

   (a) Spatial locality
   (b) Temporal locality
   (c) Prime locality
   (d) Both (a) and (b)
   (e) Both (b) and (c)

13. Which of the following is not a section of an ELF file?

   (a) `.text`
   (b) `.static`
   (c) `.rodata`
   (d) `.data`
   (e) `.bss`

(more)

14. Choose the true statement.

    (a) All thread-safe functions are reentrant.
    (b) Some reentrant functions are not thread safe.
    (c) It is never a good idea to use persistent state across multiple function calls.
    (d) It is impossible to have a race condition between two threads as long as they have no shared state.
    (e) All functions which call non-thread-safe functions are themselves not thread safe.

15. 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) None of the above.

16. In the following code, a parent opens a file twice, then the child reads a character:

```
char c;
int fd1 = open("foo.txt", O_RDONLY);
int fd2 = open("foo.txt", O_RDONLY);
if (!fork()) {
    read(fd1, &c, 1);
}
```

    Clearly, in the child, `fd1` now points to the second character of `foo.txt`. Which of the following is now true in the parent?

    (a) `fd1` and `fd2` both point to the first character.
    (b) `fd1` and `fd2` both point to the second character.
    (c) `fd1` points to the first character while `fd2` points to the second character.
    (d) `fd2` points to the first character while `fd1` points to the second character.

17. 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.

(more)

18. Consider the following two blocks of code, which are contained in *separate files*:

```
/* main.c */
int i = 0;
int main() {
    foo();
    return 0;
}
```

```
/* foo.c */
int i = 1;
void foo() {
    printf("%d", i);
}
```

What will happen when you attempt to compile, link, and run this code?

(a) It will fail to compile.

(b) It will fail to link.

(c) It will raise a segmentation fault.

(d) It will print “0”.

(e) It will print “1”.

(f) It will sometimes print “0” and sometimes print “1”.

Problem 2. (6 points):

Floating point encoding. In this problem, you will work with floating point numbers based on the IEEE floating point format. We consider two different 6-bit formats:

Format A:

- There is one sign bit s.
- There are $k = 3$ exponent bits. The bias is $2^{k-1} - 1 = 3$.
- There are $n = 2$ fraction bits.

Format B:

- There is one sign bit s.
- There are $k = 2$ exponent bits. The bias is $2^{k-1} - 1 = 1$.
- There are $n = 3$ fraction bits.

For formats A and B, please write down the binary representation for the following (use round-to-even). Recall that for denormalized numbers, $E = 1 - \text{bias}$. For normalized numbers, $E = e - \text{bias}$.

Value	Format A Bits	Format B Bits
One	`0 011 00`	`0 01 000`
Three		
$7/8$		
$15/8$		

Problem 3. (6 points):

Arrays. Consider the C code below, where H and J are constants declared with `#define`.

```
int array1[H][J];
int array2[J][H];

void copy_array(int x, int y) {
    array2[x][y] = array1[y][x];
}
```

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

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

What are the values of `H` and `J`?

H =

J =

Problem 4. (8 points):

Loops. Consider the following x86-64 assembly function:

```
loop:
    # on entry: a in %rdi, n in %esi
    movl    $0, %r8d
    movl    $0, %ecx
    testl   %esi, %esi
    jle .L3
.L6:
    movl    (%rdi,%rcx,4), %edx
    leal    3(%rdx), %eax
    testl   %edx, %edx
    cmovns  %edx, %eax
    sarl    $2, %eax
    addl    %eax, %r8d
    addq    $1, %rcx
    cmpl    %ecx, %esi
    jg  .L6
.L3:
    movl    %r8d, %eax
    ret
```

Fill in the blanks of the corresponding C code.

- You may only use the C variable names `n`, `a`, `i` and `sum`, not register names.
- Use array notation in showing accesses or updates to elements of `a`.

```
int loop(int a[], int n)
{
    int i, sum;

    sum = _____;

    for (i = ____________; ____________; ____________) {

        sum += ______________;

    }

    return ____________;
}
```

Problem 5. (10 points):

Stack discipline. Consider the following C code and its corresponding 32-bit x86 machine code. Please complete the stack diagram on the following page.

```
int fact(int n) {
    if (n == 1)
        return n;
    else
        return n * fact(n-1);
}
```

```
080483a4 <fact>:
 80483a4:   55                      push   %ebp
 80483a5:   89 e5                   mov    %esp,%ebp
 80483a7:   53                      push   %ebx
 80483a8:   83 ec 04                sub    $0x4,%esp
 80483ab:   8b 5d 08                mov    0x8(%ebp),%ebx
 80483ae:   83 fb 01                cmp    $0x1,%ebx
 80483b1:   74 0e                   je     80483c1 <fact+0x1d>
 80483b3:   8d 43 ff                lea    0xffffffff(%ebx),%eax
 80483b6:   89 04 24                mov    %eax,(%esp)
 80483b9:   e8 e6 ff ff ff          call   80483a4 <fact>
 80483be:   0f af d8                imul   %eax,%ebx
 80483c1:   89 d8                   mov    %ebx,%eax
 80483c3:   83 c4 04                add    $0x4,%esp
 80483c6:   5b                      pop    %ebx
 80483c7:   5d                      pop    %ebp
 80483c8:   c3                      ret
```

A. Draw a detailed picture of the stack, starting with the caller invoking `fact(4)`, and ending immediately **before** the call instruction that invokes `fact(2)`.

- The stack diagram should begin with the argument for `fact` that the caller has placed on the stack. To help you get started, we have given you the first one.
- Use the actual values for function arguments, rather than variable names. For example, use 3 or 2 instead of `n`.
- For callee-saved registers that are pushed to the stack, simply note the register name (e.g, `%ebx`).
- Always label `%ebp` and give its value when it is pushed to the stack, e.g., `old %ebp: 0xffff1400`.

```
Value of %ebp when fact(4) is called: 0xffffd848
Return address in function that called fact(4): 0x080483e6

Stack       The diagram starts with the
addresss    argument for fact(4)
            +------------------------------------+
0xffffd830  |                 4                  |
            +------------------------------------+
0xffffd82c  |                                    |
            +------------------------------------+
0xffffd828  |                                    |
            +------------------------------------+
0xffffd824  |                                    |
            +------------------------------------+
0xffffd820  |                                    |
            +------------------------------------+
0xffffd81c  |                                    |
            +------------------------------------+
0xffffd818  |                                    |
            +------------------------------------+
0xffffd814  |                                    |
            +------------------------------------+
0xffffd810  |                                    |
            +------------------------------------+
```

B. What is the final value of `%ebp`, immediately **before** execution of the instruction that calls `fact(2)`?

`%ebp=0x`________________________

C. What is the final value of `%esp`, immediately **before** execution of the instruction that calls `fact(2)`?

`%esp=0x`________________________

Problem 6. (12 points):

Cache memories. Consider the following matrix transpose function

```
typedef int array[2][2];

void transpose(array dst, array src) {
  int i, j;

  for (j = 0; j < 2; j++) {
    for (i = 0; i < 2; i++) {
      dst[i][j] = src[j][i];
    }
  }
}
```

running on a hypothetical 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 and write-allocate, with a block size of 8 bytes.
- Accesses to the `src` and `dst` arrays are the only sources of read and write accesses to the cache, respectively.

A. Suppose the cache has a total size of 16 data bytes (i.e., the block size times the number of sets is 16 bytes) and that the cache is initially empty. Then for each `row` and `col`, indicate whether each 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		
	col 0	col 1
row 0	m	m
row 1	m	h

`dst` array		
	col 0	col 1
row 0	m	m
row 1	m	m

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

`src` array		
	col 0	col 1
row 0	m	h
row 1	m	h

`dst` array		
	col 0	col 1
row 0	m	h
row 1	m	h

Problem 7. (6 points):

Linking. Consider the executable object file `a.out`, which is compiled and linked using the command

```
unix> gcc -o a.out main.c foo.c
```

and where the files `main.c` and `foo.c` consist of the following code:

```
/* main.c */
#include <stdio.h>

int a = 1;
static int b = 2;
int c = 3;

int main()
{
    int c = 4;

    foo();
    printf("a=%d b=%d c=%d\n", a, b, c);
    return 0;
}
```

```
/* foo.c */
int a, b, c;

void foo()
{
    a = 5;
    b = 6;
    c = 7;
}
```

What is the output of `a.out`?

Answer: a=__5__, b=__2___, c=__4___

Problem 8. (10 points):

Exceptional control flow. Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main()
{
    int val = 2;

    printf("%d", 0);
    fflush(stdout);

    if (fork() == 0) {
        val++;
        printf("%d", val);
        fflush(stdout);
    }
    else {
        val--;
        printf("%d", val);
        fflush(stdout);
        wait(NULL);
    }
    val++;
    printf("%d", val);
    fflush(stdout);
    exit(0);
}
```

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program. You will be graded on each sub-problem as follows:

- If you circle no answer, you get 0 points.
- If you circle the right answer, you get 2 points.
- If you circle the wrong answer, you get -1 points (so don't just guess wildly).

A. `01432` Y N

B. `01342` Y N

C. `03142` Y N

D. `01234` Y N

E. `03412` Y N

Problem 9. (12 points):

Address translation. This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 20 bits wide.
- Physical addresses are 18 bits wide.
- The page size is 1024 bytes.
- The TLB is 2-way set associative with 16 total entries.

The contents of the TLB and the first 32 entries of the page table are shown as follows. **All numbers are given in hexadecimal**.

TLB			
Index	Tag	PPN	Valid
0	03	C3	1
	01	71	0
1	00	28	1
	01	35	1
2	02	68	1
	3A	F1	0
3	03	12	1
	02	30	1
4	7F	05	0
	01	A1	0
5	00	53	1
	03	4E	1
6	1B	34	0
	00	1F	1
7	03	38	1
	32	09	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
000	71	1	010	60	0
001	28	1	011	57	0
002	93	1	012	68	1
003	AB	0	013	30	1
004	D6	0	014	0D	0
005	53	1	015	2B	0
006	1F	1	016	9F	0
007	80	1	017	62	0
008	02	0	018	C3	1
009	35	1	019	04	0
00A	41	0	01A	F1	1
00B	86	1	01B	12	1
00C	A1	1	01C	30	0
00D	D5	1	01D	4E	1
00E	8E	0	01E	57	1
00F	D4	0	01F	38	1

Part 1

1. The diagram below shows the format of a virtual address. Please indicate the following fields by labeling the diagram:

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

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

2. The diagram below shows the format of a physical address. Please indicate the following fields by labeling the diagram:

PPO	The physical page offset
PPN	The physical page number

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

Part 2

For the given virtual addresses, please 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 the physical address blank.

Virtual address: `078E6`

1. Virtual address (one bit per box)

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

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x1E	TLB Hit? (Y/N)	
TLB Index	0x6	Page Fault? (Y/N)	
TLB Tag	0x3	PPN	0x57

3. Physical address(one bit per box)

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

Virtual address: `04AA4`

1. Virtual address (one bit per box)

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

2. Address translation

Parameter	Value	Parameter	Value
VPN	0x12	TLB Hit? (Y/N)	
TLB Index	0x2	Page Fault? (Y/N)	
TLB Tag	0x2	PPN	0x68

3. Physical address(one bit per box)

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

Problem 10. (10 points):

Concurrency, races, and synchronization. Consider a simple concurrent program with the following specification: The main thread creates two peer threads, passing each peer thread a unique integer *thread ID* (either 0 or 1), and then waits for each thread to terminate. Each peer thread prints its thread ID and then terminates.

Each of the following programs attempts to implement this specification. However, some are incorrect because they contain a race on the value of `myid` that makes it possible for one or more peer threads to print an incorrect thread ID. Except for the race, each program is otherwise correct.

You are to indicate whether or not each of the following programs contains such a race on the value of `myid`. You will be graded on each subproblem as follows:

- If you circle no answer, you get 0 points.
- If you circle the right answer, you get 2 points.
- If you circle the wrong answer, you get -1 points (so don't just guess wildly).

A. **Does the following program contain a race on the value of `myid`?** Yes No

```
void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    Free(vargp);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i, *ptr;

    for (i = 0; i < 2; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], 0, foo, ptr);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

B. **Does the following program contain a race on the value of `myid`?** Yes No

```
void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid[i], NULL, foo, &i);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
}
```

C. **Does the following program contain a race on the value of `myid`?** Yes No

```
void *foo(void *vargp) {
    int myid;
    myid = (int)vargp;
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid[i], 0, foo, i);
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

D. **Does the following program contain a race on the value of `myid`?** Yes No

```
sem_t s; /* semaphore s */

void *foo(void *vargp) {
    int myid;
    P(&s);
    myid = *((int *)vargp);
    V(&s);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    sem_init(&s, 0, 1); /* S=1 INITIALLY */

    for (i = 0; i < 2; i++) {
        Pthread_create(&tid[i], 0, foo, &i);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

E. **Does the following program contain a race on the value of `myid`?** Yes No

```
sem_t s; /* semaphore s */

void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    V(&s);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    sem_init(&s, 0, 0); /* S=0 INITIALLY */

    for (i = 0; i < 2; i++) {
        Pthread_create(&tid[i], 0, foo, &i);
        P(&s);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```