Andrew login ID (Use block letters):______________________________________

Full Name (Use block letters):______________________________________

15-213 (18-243), Fall 2010
Final Exam

Friday, December 10. 2010

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID and full name on the front. Please write using clear block letters!
- This exam is closed book, closed notes, although you may use two 8 1/2 x 11 sheets of paper with your own notes. You may not use any electronic devices.
- 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 (20):	
2 (10):	
3 (06):	
4 (08):	
5 (09):	
6 (10):	
7 (07):	
8 (06):	
9 (10):	
10 (06):	
11 (06):	
TOTAL (98):	

Problem 1. (20 points):

Short answer and multiple choice questions on a variety of stimulating and refreshing topics.

1. Label the following networking system calls 1,2,3,4 or 5, in the order they should be called. [2 pts] (label with an X if the call is not used; a blank will receive no credit)

	Client	Server
listen	_x__	__3_
connect	_2__	_x__
accept	__x_	__4_
socket	__1_	__1_
bind	__x_	__2_

The remaining questions are multiple choice. Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10
X									

11	12	13	14	15	16	17	18	19	20
									X

2. Which of the following is NOT a universal property of reader-writer locks?
   - (a) Readers can only look at a shared item; writers can also modify it.
   - (b) If a writer has access to the item, then no other thread also has access.
   - (c) Any number of readers can read the item at the same time.
   - [✓ (d)] A writer waiting for an RW lock will get preference over subsequent read requests.

3. Starvation (in relation to threads) refers to:
   - [✓ (a)] A thread waiting for a lock indefinitely.
   - (b) A semaphore that gets locked but the thread never unlocks it after use.
   - (c) A thread is spawned but never joins the main thread when finished.
   - (d) A process fails to spawn a new thread because it's hit the maximum number of threads allowed.

4. How does x86 assembly store the return value when a function is finished?
   - (a) The `ret` instruction stores it in a special retval register.
   - (b) By convention, it is always in `%eax`.
   - (c) It is stored on the stack just above the `(%ebp)` of the callee.
   - (d) It is stored on the stack just above all the arguments to the function.

5. In IEEE floating point, what would be an effect of allocating more bits to the exponent part by taking them from the fraction part?

   (a) You could represent fewer numbers, but they could be much larger.
   (b) You could represent the same numbers, but with more decimal places.
   (c) You could represent both larger and smaller numbers, but with less precision.
   (d) Some previously representable numbers would now round to infinity

6. Consider the following two blocks of code, found in *separate files*:

```
/* main.c */                    /* foo.c */
int i=0;                        int i=1;
int main()                      void foo()
{                               {
  foo();                          printf(‘‘%d’’, i);
  return 0;                     }
}
```

   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”.

7. Which of the following is an example of external fragmentation?

   (a) A malloc’ed block needs to be padded for alignment purposes.
   (b) A user writes data to a part of the heap that isn’t the payload of a malloc’ed block.
   (c) There are many disjoint free blocks in the heap.
   (d) A user malloc’s some heap space and never frees it.

8. Which of the following is NOT the default action for any signal?

   [✓ (a)] The process terminates all of its children.
   (b) The process terminates and dumps core.
   (c) The process terminates.
   (d) The process stops until restarted by a SIGCONT signal.

9. Which of the following is FALSE concerning x86-64 architecture?

   (a) A `double` is 64 bits long.
   (b) Registers are 64 bits long.
   (c) Pointers are 64 bits long.
   (d) Pointers point to locations in memory that are multiples of 64 bits apart.

10. Consider the following block of code:

```
int main()
{
  int a[213];
  int i;
  //int j = 15;
  for(i = 0; i < 213; i++)
  a[i] = i;
  return 0;
  a[0] = -1;
}
```

    Which of the following instances of ’bad style’ is present?

   (a) Dead code.
   (b) Magic numbers.
   (c) Poor indentation.
   (d) All of the above.

11. Consider the following structure declarations on a 64-bit Linux machine.

```
struct RECORD {
    long value2;
    double value;
    char tag[3];
};

struct NODE {
    int ref_count;
    struct RECORD record;
    union {
        double big_number;
        char string[12];
    } mix;
};
```

Also, a global variable named `my_node` is declared as follows:

```
struct NODE my_node;
```

If the address of `my_node` is `0x6008e0`, what is the value of `&my_node.record.tag[1]` ?

(a) `0x6008f8`
(b) `0x6008fa`
(c) `0x6008f9`
(d) `0x6008f5`
(e) `0x6008f1`

12. With reference to the previous question, what is the size of `my_node` in bytes ?

(a) 48
(b) 44
(c) 40
(d) 42
(e) 50

13. Which of the following x86 instructions can be used to add two registers and store the result without overwriting either of the original registers?

(a) `mov`
(b) `lea`
(c) `add`
(d) None of the above

14. Which of these uses of caching is not crucial to program performance?

    (a) Caching portions of physical memory
    (b) Caching virtual address translations
    (c) Caching virtual addresses
    (d) Caching virtual memory pages
    (e) None of the above (that is, they are all crucial)

15. Assuming all the system calls succeed, which of the following pieces of code will print the word ”Hello” to `stdout`?

    (a)
    ```
    int fd = open("hoola.txt", O_RDWR);
    dup2(fd, STDOUT_FILENO);
    printf("Hello");
    fflush(stdout);
    ```
    (b)
    ```
    int fd = open("hoola.txt", O_RDWR);
    dup2(fd, STDOUT_FILENO);
    write(STDOUT_FILENO, "Hello", 5);
    ```
    (c)
    ```
    int fd = open("hoola.txt", O_RDWR);
    dup2(fd, STDOUT_FILENO);
    printf("Hello");
    ```
    (d)
    ```
    int fd = open("hoola.txt", O_RDWR);
    dup2(STDOUT_FILENO, fd);
    write(fd, "Hello", 5);
    ```
    (e)
    ```
    int fd = open("hoola.txt", O_RDWR);
    dup2(fd, STDOUT_FILENO);
    write(fd, "Hello", 5);
    ```

16. Consider the following piece of code. Note that the file name is the same for both calls to `open`, and assume the file `one.txt` exists.

    ```
    int fd = open("one.txt", O_RDWR);
    int fd2 = open("one.txt", O_RDONLY);
    ```

    Which of the following statement is true?

    (a) `fd` and `fd2` will share the same file offset
    (b) `fd2` will be invalid because you cannot have two open file descriptors referring to the same file at the same time.
    (c) Both `fd` and `fd2` will have an initial file offset that is set to the end of the file
    (d) Whatever is written to the file through `fd`, can be read using `fd2`
    (e) In total, there will be two copies of the file `one.txt` in memory, one associated with `fd` and the other with `fd2`. Any changes made in a copy will **not** be reflected in the other copy.

17. In malloclab, we provided code for an implicit list allocator (the naive implementation). Many students improved this code by creating an explicit linked list of free blocks. Which of the following reason(s) explain(s) why an explicit linked list implementation has better performance?

    I. Immediate coalescing when freeing a block is significantly faster for an explicit list

    II. The implicit list had to include every block in the heap, whereas the explicit list could just include the free blocks, making it faster to find a suitable free block.

    III. Inserting a free block into an explicit linked list is significantly faster since the free block can just be inserted at the front of the list, which takes constant time.

    (a) I only.
    (b) II only.
    (c) III only.
    (d) II and III only.
    (e) All I, II and III.

18. Suppose a local variable `int my_int` is declared in a function named `func`. Which of the following is considered safe in C?

    (a) `func` returns `&my_int` and the caller dereferences the returned pointer.
    (b) `func` returns `&my_int` and the caller prints the returned pointer to the screen
    (c) `func` sets the value of a global variable to `&my_int` and returns. The global variable is unchanged up to the point another function dereferences the global variable.
    (d) None of the above

19. If a parent forks a child process, to which resources might they need to synchronize their access to prevent any unexpected behavior?

    (a) malloc'ed memory
    (b) stack memory
    (c) global variables
    (d) file descriptors
    (e) None of the above

Problem 2. (10 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 formats:

Format A: 8-bit floating point numbers:

- There is one sign bit s. $s = 1$ indicates negative numbers.
- There are $k = 4$ exponent bits. The bias is $2^{k-1} - 1 = 7$.
- There are $n = 3$ fraction bits.

Format B: 9-bit floating point numbers:

- There is one sign bit s. $s = 1$ indicates negative numbers.
- There are $k = 4$ exponent bits. The bias is $2^{k-1} - 1 = 7$.
- There are $n = 4$ fraction bits.

1. How would you represent positive infinity using **format A**?

   Binary representation for positive infinity: ________________

2. How would you represent $\sqrt{-100}$ using **format B**?

   Give an example binary representation: ________________

3. For formats A and B, please write down the binary representation and the corresponding values for the following (use round-to-even):

Description	Format A binary	Format A value	Format B binary	Format B value
Zero	0 0000 000	0	0 0000 0000	0
Largest normalized value				
Smallest positive number				
Negative one		-1		-1
2.625				

Problem 3. (6 points):

Accessing 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[y][x] = array1[x][y];
}
```

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
        leaq    (%rsi,%rsi,8), %rdx
        addq    %rdi, %rdx
        movq    %rdi, %rax
        salq    $4, %rax
        subq    %rdi, %rax
        addq    %rsi, %rax
        movl    array1(,%rax,4), %eax
        movl    %eax, array2(,%rdx,4)
        ret
```

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

```
H =

J =
```

Problem 4. (8 points):

Assembly/C translation. Consider the following C code and assembly code for an interesting function:

```
int rofl(int *a, int n)
{
    int i, k;

    for(i = 0; i < n; i++)
    {
        k = a[i];

        if(i == k)
        {
            ________;
        }

        if(________)
        {
            return k;
        }

        a[i] = ________;
        a[k] = k;
    }

    return ________;
}
```

```
40055c <rofl>:
40055c: test   %esi,%esi
40055e: jle    40058e <rofl+0x32>
400560: mov    %rdi,%r8
400563: mov    $0x0,%ecx
400568: mov    (%r8),%edx
40056b: cmp    %edx,%ecx
40056d: je     400583 <rofl+0x27>
40056f: movslq %edx,%rax
400572: lea    (%rdi,%rax,4),%r9
400576: mov    (%r9),%eax
400579: cmp    %edx,%eax
40057b: je     400593 <rofl+0x37>
40057d: mov    %eax,(%r8)
400580: mov    %edx,(%r9)
400583: add    $0x1,%ecx
400586: add    $0x4,%r8
40058a: cmp    %ecx,%esi
40058c: jg     400568 <rofl+0xc>
40058e: mov    $0xffffffff,%edx
400593: mov    %edx,%eax
400595: retq
```

A. Using your knowledge of C and assembly, fill in the blanks above with the appropriate expressions.

B. *Extra credit (1 point).* Briefly describe what the `rofl` function does. Hint: Think about what happens when every integer k in the array `a` satisfies $0 \leq k \leq n - 1$.

Problem 5. (9 points):

Representing and accessing structures. The following problems concern the compilation of C code involving `struct`'s.

A. In the following C code, the declarations of data types `type1_t` and `type2_t` are given by `typedef`'s, and the declaration of the constant `CNT` is given by a `#define`:

```
typedef struct {
    type1_t y[CNT];
    type2_t x;
} a_struct;

void p1(int i, a_struct *ap) {
    ap->y[i] = ap->x;
}
```

Compiling the code for IA32 gives the following assembly code:

```
# i at 8(%ebp), ap at 12(%ebp)
p1:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        movsbl  28(%eax),%ecx
        movl    8(%ebp), %edx
        movl    %ecx, (%eax,%edx,4)
        popl    %ebp
        ret
```

Give a combination of values for the two data types and `CNT` that could yield the above assembly code:

`type1_t`:

`type2_t`:

`CNT`:

B. In the following C code, the declaration of data type `type_t` is given by a `typedef`, and the declaration of the constant `CNT` is given by a `#define`:

```
typedef struct {
    int left;
    type_t m[CNT];
    int right;
} b_struct;

int p2(int i, b_struct *bp) {
    return bp->left * bp->right;
}
```

For some combinations of `type_t` and `CNT`, the following x86-64 code is generated:

```
# bp in %rsi
p2:
        movl    24(%rsi), %eax
        imull   (%rsi), %eax
        ret
```

For each of the combinations below, indicate whether it could (Y) or could not (N) cause the above code to be generated:

`type_t`	`CNT`	Generated? (Y/N)
`int`	6	
`short`	9	
`char`	17	
`char *`	5	
`double`	2	
`struct { int i; double d[2]; }`	1	

Problem 6. (0xa points):

The stack discipline. This problem deals with stack frames in Intel IA-32 machines. Consider the following C function and corresponding assembly code.

```
struct node_t;
typedef struct node_t{
    void * elem;
    struct node_t *left;
    struct node_t *right;
} node;

void oak(node * tree, void (*printFunc)(node *)){
    /*POINT A*/
    (*printFunc)(tree);
    if (tree->left) {
        /*POINT B*/
        oak(tree->left,printFunc);
    }
    if (tree->right) {
        oak(tree->right,printFunc);
    }
}
```

```
00000000 <oak>:
 0: 55                push   %ebp
 1: 89 e5             mov    %esp,%ebp
 3: 83 ec 18          sub    $0x18,%esp
 6: 89 5d f8          mov    %ebx,0xfffffff8(%ebp)
 9: 89 75 fc          mov    %esi,0xfffffffc(%ebp)
 c: 8b 5d 08          mov    0x8(%ebp),%ebx
 f: 8b 75 0c          mov    0xc(%ebp),%esi
12: 89 1c 24          mov    %ebx,(%esp)
           /*POINT A*/
15: ff d6             call   *%esi
17: 8b 43 04          mov    0x4(%ebx),%eax
1a: 85 c0             test   %eax,%eax
1c: 74 0c             je     2a <oak+0x2a>
1e: 89 74 24 04       mov    %esi,0x4(%esp)
22: 89 04 24          mov    %eax,(%esp)
           /*POINT B*/
25: e8 fc ff ff ff    call   26 <oak+0x26>
2a: 8b 43 08          mov    0x8(%ebx),%eax
2d: 85 c0             test   %eax,%eax
2f: 74 0c             je     3d <oak+0x3d>
31: 89 74 24 04       mov    %esi,0x4(%esp)
35: 89 04 24          mov    %eax,(%esp)
38: e8 fc ff ff ff    call   39 <oak+0x39>
3d: 8b 5d f8          mov    0xfffffff8(%ebp),%ebx
40: 8b 75 fc          mov    0xfffffffc(%ebp),%esi
43: 89 ec             mov    %ebp,%esp
45: 5d                pop    %ebp
46: c3                ret
```

(over)

Please draw a picture of the stack frame, starting with any arguments that might be placed on the stack for the `oak` function, showing the stack at each of points `A`, and `B`, as specified in the code above. Your diagram should only include actual values where they are known, if you do not know the value that will be placed on the stack, simply label what it is (i.e., ”old ebp”).

Stack A:

Stack B:

Problem 7. (7 points):

Cache memories. In this problem, we will consider the performance of the cache. You can make the following assumptions:

- There's only one level of cache.
- Block size is 4 bytes.
- The cache has 4 sets.
- Each cache set has two lines.
- Replacement policy is LRU.

Consider the following function which sets a 4×4 square in the upper left corner of an array to zero. You should assume that only operations involving `array` change the cache, that `array[0][0]` is at address `0x1000000`, and that the cache is empty when `clear4x4` is called.

```
#define LENGTH 8

void clear4x4(char array[LENGTH][LENGTH]){
    int row, col;
    for(col = 0; col < 4; col++){
        for(row = 0; row < 4; row++){
            array[row][col] = 0;
        }
    }
}
```

A. (3 pts) How many cache misses will there by when `clear4x4` is called?

   Number of cache misses: ____________________

B. (3 pts) If `LENGTH` is changed to 16 how many cache misses will `clear4x4` have when called?

   Number of cache misses: ____________________

C. (1 pt) If `LENGTH` is changed to 17, will calling `clear4x4` have a larger, smaller, or equal number of cache misses than when `LENGTH` is 16? Circle the correct answer.

   - 16×16 will have MORE misses than 17×17.
   - 16×16 and 17×17 will have an EQUAL number of MISSES.
   - 17×17 will have MORE misses than 16×16.

Problem 8. (6 points):

Processes vs. threads. This problem tests your understanding of the some of the important differences between processes and threads. Consider the following C program:

```
#include "csapp.h"

/* Global variables */
int cnt;
sem_t mutex;


/* Helper function */
void *incr(void *vargp)
{
    P(&mutex);
    cnt++;
    V(&mutex);
    return NULL;
}
```

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

    sem_init(&mutex, 0, 1); /* mutex=1 */

    /* Processes */
    cnt = 0;
    for (i=0; i<2; i++) {
        incr(NULL);
        if (fork() == 0) {
            incr(NULL);
            exit(0);
        }
    }
    for (i=0; i<2; i++)
        wait(NULL);
    printf("Procs:   cnt = %d\n", cnt);

    /* Threads */
    cnt = 0;
    for (i=0; i<2; i++) {
        incr(NULL);
        pthread_create(&tid[i], NULL, incr, NULL);
    }
    for (i=0; i<2; i++)
        pthread_join(tid[i], NULL);
    printf("Threads: cnt = %d\n", cnt);

    exit(0);
}
```

A. What is the output of this program?

```
Procs:   cnt = _2_

Threads: cnt = _4_
```

Problem 9. (10 points):

Virtual memory. You are taking an operating systems class where you must write a kernel that supports virtual memory. Unfortunately, you have been stuck with a not-so-bright partner who is under the illusion that address translations are performed by the kernel. Without consulting you, he went ahead and wrote a translation function, called `log_to_phys`, which is shown on the following page.

Address translations are actually done by hardware of course, but you realize that by studying your partner's code, you can learn some valuable information about the system – information that will earn you 10 points on your 213 final exam!

As you study your partner's code, keep in mind the following things:

1. Pointers and `unsigned int`'s are both **2 bytes** long on this particular system (i.e., `sizeof(unsigned int)` $= 2$ and `sizeof(void *)` $= 2$).
2. You do not have to worry about any type of pointer arithmetic in this problem.
3. Although the code is silly in the sense that translations are not done in software, you can assume that the functionality is correct.

(over)

```
/* Note to self: Recall that on this machine, sizeof(unsigned int) = 2
   and sizeof(void *) = 2 */


/*
 * log_to_phys - logical is a variable which contains a virtual
 *               address. The physical translation is returned.
 */
void * log_to_phys (void * logical, void ** pd_base)
{

  /* Casting to unsigned int is done so you don’t
     have to worry about any pointer arithmetic */
  unsigned int logical_addr = (unsigned int) logical;
  unsigned int pd_base_u = (unsigned int) pd_base;

  unsigned int offset = logical_addr & (0x1F);
  unsigned int temp = logical_addr >> 5;
  unsigned int index1 = (temp & 0x780) >> 7;
  unsigned int index2 = temp & 0x7F;

  unsigned int * pde_addr = (unsigned int *) (pd_base_u + (index1 << 1));
  unsigned int entry1 = *pde_addr;

  /* Check valid bit */
  if(!(entry1 & 0x1)) {
    /* This is how you throw a page fault, right? */
    return NULL;
  }

  /* Discard the valid bit now */
  entry1 = entry1 & (˜0x1);

  unsigned int * pte_addr = (unsigned int *) (entry1 + (index2 << 1));
  unsigned int entry2 = *pte_addr;

  /* Check valid bit */
  if(!(entry2 & 0x1)) {
    /* This is how you throw a page fault, right? */
    return NULL;
  }

  /* Discard the valid bit now */
  entry2 = entry2 & (˜0x1);

  /* This is the logical address! */
  return ((void *) (entry2 | offset));
}
```

The following questions refer to the code on the previous page:

1. How many bytes are the pages (virtual and physical pages are the same size)?

2. How many entries are in the page directory for this architecture?

3. How many bytes long is each entry of the page directory?

4. How many entries are in each page table for this architecture?

5. How many bytes long is each entry of a page table?

Problem 10. (6 points):

Signals. Consider the following two different snippets of C code. Assume all functions return without error, no signals are sent from other processes, and `printf` is atomic.

Code Snippet 1:

```
int main() {
    int pid = fork();
    if(pid > 0){
        kill(pid, SIGKILL);
        printf("a");
    }else{
        /* getppid() returns the pid
           of the parent process */
        kill(getppid(), SIGKILL);
        printf("b");
    }
}
```

Code Snippet 2:

```
int a = 1;

void handler(int sig){
    a = 0;
}

void emptyhandler(int sig){
}

int main() {
    signal(SIGINT, handler);
    signal(SIGCONT, emptyhandler);

    int pid = fork();
    if(pid == 0){
        while(a == 1)
            pause();
        printf("a");
    }else{
        kill(pid, SIGCONT);
        printf("b");
        kill(pid, SIGINT);
        printf("c");
    }
}
```

For each code snippet write a Y next to an outcome if it could occur, otherwise write N.

Snippet 1 Outcome	Possible? (Y/N)
Nothing is printed.	
“a” is printed.	
“b” is printed.	
“ab” is printed.	
“ba” is printed.	
A process does not terminate.	

Snippet 2 Outcome	Possible? (Y/N)
Nothing is printed.	
“ba” is printed.	
“abc” is printed.	
“bac” is printed.	
“bca” is printed.	
A process does not terminate.	

Problem 11. (6 points):

Synchronization. This problem is about synchronizing a producer/consumer system that shares a queue between two threads:

- The code for the producer/consumer system is shown on the following page.
- The *producer thread* adds data items to the back of the queue, and the *consumer thread* removes and processes data items from the front of the queue.
- The queue is initially empty, and has a capacity of 10 data items.
- The producer thread can only create 2 items at a time. Similarly, the consumer must consume 3 items at a time. In particular, the `produce2` function, which is called by the producer thread, produces and adds two data items to the queue each time it is called. Similarly, the `consume3` function, which is called by the consumer thread, removes 3 data items from the queue each time it is called. (These function declarations are not shown here.)

Your task is to modify the code on the following page:

- Add the necessary semaphore operations to guarantee the *produce-2-items* and *consume-3-items* requirements.
- Add the appropriate calls `sem_init` to initialize the two semaphores to the correct values.

Recall that semaphore functions have the following prototypes:

- `P(sem_t *sem);`
- `V(sem_t *sem);`
- `sem_init(sem_t *sem, NULL, unsigned int val);`

(over)

Here is the code that you will update:

```
#include <pthread.h>
#include <semaphore.h>

/* Semaphores */
sem_t cons_sem, prod_sem;

/* Producer thread */
void* producer(void* vargp)
{
  while(1){
    /* Insert semaphore operation(s) here */



    produce2(); /* Produce 2 items */
    /* Insert semaphore operation(s) here */



  }
  return NULL;
}

/* Consumer thread */
void * consumer(void* vargp)
{
  while(1){
    /* Insert semaphore operation(s) here */



    consume3(); /* Consume 3 items */
    /* Insert semaphore operation(s) here */



  }
  return NULL;
}
```

```
/* Main routine */
int main()
{
  pthread_t tid;

  /* Initialize semaphores */



  pthread_create(&tid, NULL,
    producer, NULL);
  pthread_create(&tid, NULL,
    consumer, NULL);

  return;
}
```