Andrew login ID: _____

Full Name: _____

CS 15-213, Spring 2004

Final Exam

May 3, 2004

Instructions:

- Make sure that your exam has 20 pages and is not missing any sheets, then write your full name and Andrew login ID on the front.
- 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 120 points.
- The problems are of varying difficulty. The point value of each problem is indicated. 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. You may not use a calculator, laptop or any other electronic or wireless device. Good luck!

1 (16):	
2 (15):	
3 (10):	
4 (9):	
5 (12):	
6 (18):	
7 (9):	
8 (24):	
9 (10):	
TOTAL (123):	

Problem 1. (16 points):

This problem tests your understanding of how integers and floating point numbers are stored in memory. For Parts **A**-**D**, indicate what the output of the program would be, assuming that there are no errors during the compilation and execution of the program. Assume standard Linux alignment policies.

Consider the following C data structures.

```
struct point{ int x, y; };

struct MyStruct
{
  int i;
  union
  {
    struct point p;
    double d;
  } u;
  int j;
  double e;
};
```

Part A:

```
int funA(struct MyStruct *m)
{
  m->u.p.x = 5;
  m->u.p.y = 10;
}

int main()
{
  struct MyStruct m;

  memset( &m, 0, sizeof(struct MyStruct) );
  funA(&m);
  printf("%d, %d \n", m.u.p.x, m.u.p.y);
}
```

Output: _____

Continued . . .

(Question 1 cont'd)

Part B:

```
int funB(struct MyStruct m)
{
  m.u.p.x = 5;
  m.u.p.y = 10;
}

int main()
{
  struct MyStruct m;

  memset( &m, 0, sizeof(struct MyStruct) );
  funB(m);
  printf("%d, %d \n", m.u.p.x, m.u.p.y);
}
```

Output: _____

Part C:

```
int funC(struct MyStruct *m)
{
  int *ptr = (int*)m;
  ptr = ptr + 2;
  *ptr = 5;
  ptr++;
  *ptr = 10;
}

int main()
{
  struct MyStruct m;

  memset( &m, 0, sizeof(struct MyStruct) );
  funC(&m);
  printf("%d, %d \n", m.u.p.x, m.u.p.y);
}
```

Output: _____

Continued . . .

(Question 1 cont'd)

Part D:

```
int funD(struct MyStruct *m)
{
  *(int*)  &m->u.d = 5;
  *(int*) (&m->u.d + 1) = 10;
}

int main()
{
  struct MyStruct m;

  memset( &m, 0, sizeof(struct MyStruct) );
  funD(&m);
  printf("%d, %d \n", m.u.p.x, m.u.p.y);
}
```

Output: _____

Part E: Recall that a `float` is a 32-bit floating point number. It is divided into a 23 bit `frac` field, an 8 bit `exp` field, and a single-bit sign field.

```
void main()
{
  union hack    /* struct that lets us XOR floats */
  {
    float f;
    unsigned u;
  } max, temp;  /* 2 hack unions */

  float x = 1.0;
  float y = 0.5;


  do   /* find the largest fp-number < 2.0 */
    {
      max.f = x;
      x += y;
      y *= 0.5;
    } while (x != max.f && x < 2.0);


  /* Assume now that max.f stores largest fp-number < 2.0 */

  temp.f = 1.0;
  temp.u = temp.u ^ max.u;
}
```

At the end of this program,
how many bits in `temp.u` are set to 1? _____.

Problem 2. (15 points):

This problem tests your understanding of Unix system I/O.

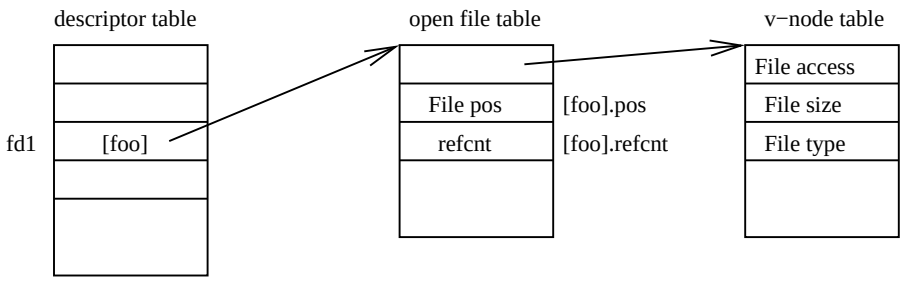
Part 1 (10 pts):

Examine the following C code, and answer the questions below. You may assume that there are neither compilation nor execution errors.

```
 1: int main(int argc, char * argv[])
 2: {
 3:       int fd1, fd2, fd3;
 4:       char c;
 5:
 6:       fd1 = open("foo", O_RDONLY);
 7:       if (fork() == 0) {
 8:           fd2 = open("foo", O_RDONLY);
 9:           read(fd2, &c, 1);
10:           ...
11:       } else {
12:           wait(NULL);
13:           fd3 = open("bar", O_RDONLY);
14:           dup2(fd3, fd1);
15:           ...
16:       }
17:   ...
18: }
```


Continued . . .

(Question 2 cont'd)



The figure above represents some of the kernel data structures used to manage files. We use $[x]$ to denote file x's descriptor table entry, which points to file x's open file table. For example, in the above figure, $[foo]$ denotes descriptor table entry created in line 6 by the program on the previous page, which points to file foo's open file table. Correspondingly, $[foo].pos$ and $[foo].refcnt$ denote the "File pos" and "refcnt" entry in its open file table.

Using the notation described in the previous paragraph and the program on the previous page, please complete the following tables. When filling out a row, indicate the state of the system immediately after the line indicated has executed. Ignore table entries marked with a "——-".

Parent process:

line #	fd1	fd2	fd3	$[foo].pos$	$[foo].refcnt$	$[bar].pos$	$[bar].refcnt$
line 6	$[foo]$	——-	——-	0	1	——-	——-
line 13		——-					
line 14		——-					

Child process:

line #	fd1	fd2	fd3	$[foo].pos$	$[foo].refcnt$	$[bar].pos$	$[bar].refcnt$
line 8		——-	——-			——-	——-
line 9		——-	——-			——-	——-

Continued . . .

(Question 2 cont'd)

Part 2 (5 pts):

This problem is concerned with a simple client-server application. The server is called `dump`. It waits for a client connection request on port **3304**, and then outputs to `stdout` every character that it receives from the client. The client is called `msger`. Its source code is listed below. The header files and the variable declarations are omitted in this source, but you can assume that the code compiles and runs without any errors.

```
int main(int argc, char * argv[])
{
        /* the variable declarations are omitted here  */

        sock = socket(AF_INET, SOCK_STREAM, 0);

        client.sin_addr.s_addr = htonl(INADDR_ANY);
        client.sin_family = AF_INET;
        client.sin_port = 0;

        server.sin_addr.s_addr = inet_addr("192.168.0.1");
        server.sin_family = AF_INET;
        server.sin_port = htons(3304);

        connect(sock, (struct sockaddr *)&server, sizeof(server));

        if (fork() == 0) {
            dup2(STDOUT_FILENO, sock);
        }

        strcpy(msg_buf, "hello world\n");
        rio_writen(sock, msg_buf, strlen(msg_buf));
        close(sock);
}
```

A. This client uses `rio_writen()`, what is the main reason for using `rio_writen()` instead of the standard unix system call `write()`? Limit your answer to no more than **one** sentence.

B. Suppose `dump` is running on a computer with IP address "192.168.0.1". You launch `msger` on a computer with IP address "192.168.0.2", please fill in the following table with the output on `stdout` from both computers. Write "Nothing" if there is no output.

192.168.0.1	192.168.0.2

Problem 3. (10 points):

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

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

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

Suppose the above code generates the following assembly code:

```
copy:
  push   %ebp
  mov    %esp,%ebp
  mov    0xc(%ebp),%edx
  push   %ebx
  lea    (%edx,%edx,2),%eax
  mov    0x8(%ebp),%ebx
  lea    0x0(,%ebx,8),%ecx
  lea    (%edx,%eax,4),%eax
  sub    %ebx,%ecx
  add    %ebx,%eax
  mov    array2(,%eax,4),%eax
  add    %edx,%ecx
  mov    %eax,array1(,%ecx,4)
  mov    (%esp,1),%ebx
  pop    %ebx
  ret
```

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

`M` = _____

`N` = _____

Problem 4. (9 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 }
```


Continued . . .

(Question 4 cont'd)

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 5. (12 points):

The code samples below are designed to test your understanding of multithreading, semaphores, and the effects of multiple threads modifying the same data. For each section, you will be asked to specify how many different outputs the given code can have. Since there may be many possible outputs, you will not be asked to provide the outputs themselves. You may assume, for the purposes of this problem, that printf executes atomically.

```
sem_t sem;

int main()
{
    int j;
    pthread_t tids[3];
    sem_init(&sem, 0, 1);
    for (j=0; j<3; j++) {
        pthread_create(&tids[j], NULL, doit, NULL);
    }
    for (j=0; j<3; j++) {
        pthread_join(&tids[j], NULL);
    }
    return 0;
}
```

This is the main function for all three problems below. The only differences in the problems are the details of the thread function, `doit` listed on the next page.

Continued . . .

(Question 5 cont'd)

part A:

```
int i = 0;

void *doit(void *arg)
{
    P(&sem);
    i = i + 1;
    printf("%d\n", i);
    V(&sem);
}
```

How many different outputs are possible on executing the code above? _____

part B:

```
int i = 0;

void *doit(void *arg)
{
    P(&sem);
    i = i + 1;
    V(&sem);
    printf("%d\n", i);
}
```

How many different outputs are possible on executing the code above? _____

part C:

```
int i = 0;

void *doit(void *arg)
{
    i = i + 1;
    printf("%d\n", i);
}
```

When the above code runs, at least one of the four outputs listed below are *not* possible.

```
(a)  3       (b) 3         (c) 1          (d)  1
     2           3             3               1
     1           3             1               1
```

List the letters identifying the illegal outputs here: _____

Problem 6. (18 points):

When attempting to debug a program, it is often helpful to look at the stack to determine what functions have been called and with what arguments. This is called a backtrace and can be performed in gdb with the `bt` command. Your task for this section of the exam is to harness your knowledge of the x86 calling conventions, combined with your understanding of different data representations to perform a manual backtrace. To facilitate this, we have provided you with the contents of the stack in hexadecimal format. We have also provided you with a table mapping certain addresses to strings. Additionally, we have given you the address ranges for the machine code of the various functions. Finally, when the `bt` is executed you are inside the `bar()` function and the value of `ebp` is 0xbffff988.

Your backtrace should terminate at the `main()` function (and should also include any arguments to main!).

You may assume that all arrays of strings are terminated with a NULL pointer.

When providing your answers, please specify the name of the argument and use the following conventions:

- characters should be written in single quotes (e.g. character='c').
- integers should be written in decimal format (e.g. integer=23).
- strings should be in double quotes (e.g. string="This is a string").
- string arrays should be written in square brackets and be comma delimited (e.g. array=["This", "is", "an", "array", (NULL)])
- pointers should be written as hexadecimal numbers (e.g. pointer=0xbffffa84)

For example: main(argc=2, argv=["/bin/foo", "Word", (NULL)])

Continued ...

The stack is as follows:

```
  Address          Data
0xbffff988:     0xbffff9b8
0xbffff98c:     0x08048401
0xbffff990:     0xbffffb28
0xbffff994:     0xdeadbeef
0xbffff998:     0xbffff9a8
0xbffff99c:     0x4002c4ed
0xbffff9a0:     0x00000000
0xbffff9a4:     0x0804833a
0xbffff9a8:     0x000000a0
0xbffff9ac:     0x080483e9
0xbffff9b0:     0xbffff9d0
0xbffff9b4:     0x00000000
0xbffff9b8:     0xbffff9d8
0xbffff9bc:     0x08048378
0xbffff9c0:     0x0000000f
0xbffff9c4:     0x0000002d
0xbffff9c8:     0x000000d5
0xbffff9cc:     0x080483e9
0xbffff9d0:     0xbffff9d0
0xbffff9d4:     0x00000000
0xbffff9d8:     0xbffff9f8
0xbffff9dc:     0x4002c4ed
0xbffff9e0:     0x00000002
0xbffff9e4:     0xbffffa2c
0xbffff9e8:     0x00000004
0xbffff9ec:     0x080483e9
0xbffff9f0:     0xbffff9d0
0xbffff9f4:     0x00000000
0xbffff9f8:     0xbffffa48
0xbffff9fc:     0x00000002
0xbffffa00:     0x75656869
0xbffffa04:     0x4002c3ac
0xbffffa08:     0xbffffb08
0xbffffa0c:     0xbffffb28
0xbffffa10:     0x00000000
0xbffffa14:     0xbffffb38
0xbffffa18:     0x00000000
0xbffffa1c:     0x080483e9
0xbffffa20:     0xbffff9d0
0xbffffa24:     0x00000002
0xbffffa28:     0x00000000
0xbffffa2c:     0xbffffb08
0xbffffa30:     0xbffffb88
0xbffffa34:     0x00000000
0xbffffa38:     0x000d5213
0xbffffa3c:     0x080483e9
0xbffffa40:     0xbffff9d0
0xbffffa44:     0x0000003b
```

The following table lists a few selected characters and their associated ASCII values.

Hex	Char	Hex	Char	Hex	Char	Hex	Char
2b	’+’	2c	’,’	2d	’-’	2e	’.’
2f	’/’	30	’0’	65	’A’	97	’a’

Continued . . .

(Question 6 cont'd)

The following table lists the addresses at which certain strings are found in the executable.

Address	String
0xbffffb08	"/bin/foo"
0xbffffb28	"Walrus"
0xbffffb48	"Game"
0xbffffb68	"Dawg"
0xbffffb88	"Lock"
0xbffffba8	"Worst"
0xbffffbc8	"Wiz"
0xbffffbe8	"Mack"

The following table lists the addresses at which certain functions begin in the executable. You should assume that each function immediately follows the function listed above it in the executable.

Address	Prototype
0x0804835c	`int main(int argc, char *argv[]);`
0x08048388	`int foo(void *addr);`
0x08048392	`void fancy(int i, char *str);`
0x08048398	`int hammer(int a, char c, int i);`
0x08048404	`char bar(char *str, double *dp);`
0x0804840e	`double chouse(int i);`

Please fill in the following fields as per the guidelines given above. Note that you might not need all of the space provided.

1. `bar` (_____

_____)

2. _____ (_____

_____)

3. _____ (_____

_____)

4. _____ (_____

_____)

Problem 7. (9 points):

This problem deals with dynamic memory allocators.

Part A: Suppose that you have been asked to implement a dynamic memory allocator for a real-time system with strict bounds on the amount of time for an operation (a malloc, a free, etc). In order to combat false fragmentation, you have decided to use some sort of coalescing. Which type of coalescing (immediate or deferred) should you use, and why? Use at most two sentences for your answer.

Part B: Assume a minimum block size of 32 bytes for this part of the problem. Do implicit free lists or explicit free lists use more memory. That is, will one of these strategies necessarily use more space than the other, assuming that everything else in the allocators are identical, and why? Use at most two sentences for your answer.

Part C: Assume you are designing a special purpose dynamic memory allocator. One of the interesting properties of this system is that only 8 different sizes, all of which you know in advance and all of which are small relative to the page size of the system, will be requested by the user (though there are no guarantees how many times each of these will be requested). What are the most important ways in which your design for this allocator would differ from your design of a general purpose allocator? Use at most 2 sentences for your answer.

Problem 8. (24 points):

You are working for a company that is developing a new processor and the compiler team is way behind schedule. In order to sell your product you need to show high performance on certain kernels important to your potential users. You have been asked to optimize the following kernel. You can't use assembly because the ISA of the processor may change.

You can assume that `N` and `M` are large and that `M` is always even. The processor will have a cache, it won't be large compared to `N*M`. `a`, `b`, and `e` are all 2-D `NxM` arrays. `e` may have values other than 0 in it before the call to `kernel`.

```
void
kernel(int N, int M, int* a, int *b, int *e)
{
   int i,j,k;

   for (i=0; i<M; i++)
      for (j=0; j<N; j++)
         if (isodd(a[j*M+i])) {
            for (k=0; k<M; k++)
               e[j*M+i] += a[j*M+i]*b[j*M+k];
         }
}

int
isodd(int val)
{
  return ((val % 2) == 1);
}
```

Use this space for your work. Nothing on this page will be used in the grading of this question. This is a two part question. Put your final answer on the next two pages.

Continued . . .

(Question 8 cont'd)

For your reference here is the original code again:

```
void kernel(int N, int M, int* a, int *b, int *e)
{
   int i,j,k;

   for (i=0; i<M; i++)
      for (j=0; j<N; j++)
         if (isodd(a[j*M+i])) {
            for (k=0; k<M; k++)
               e[j*M+i] += a[j*M+i]*b[j*M+k];
         }
}

int isodd(int val)
{
  return ((val % 2) == 1);
}
```

Part 1: Provide an optimized version of `kernel` here. Please be neat as illegible answers will recieve no credit.

Continued . . .

(Question 8 cont'd)

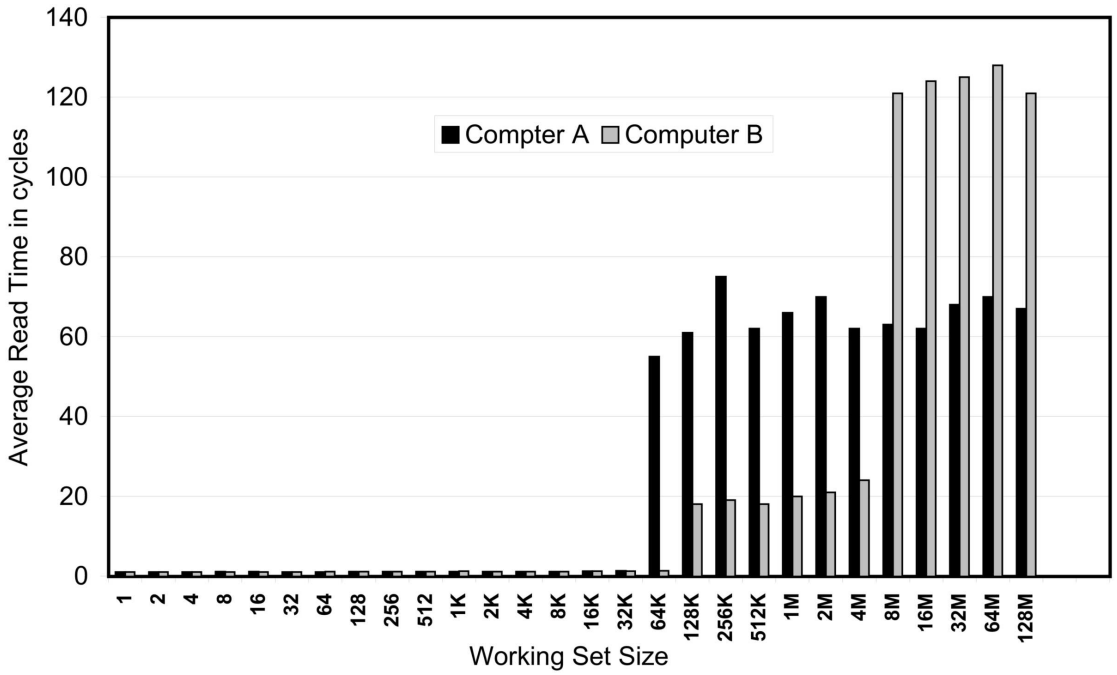
For your reference here is the original code again:

```
void kernel(int N, int M, int* a, int *b, int *e)
{
   int i,j,k;

   for (i=0; i<M; i++)
      for (j=0; j<N; j++)
         if (isodd(a[j*M+i])) {
            for (k=0; k<M; k++)
               e[j*M+i] += a[j*M+i]*b[j*M+k];
         }
}

int isodd(int val)
{
  return ((val % 2) == 1);
}
```

Part 2: List the optimizations you performed and, in **no** more than one short sentence (a sentence fragment is ok too), the rationale behind each one.

Problem 9. (10 points):

Imagine that you are given two computers (A and B) that operate at the same processor clock rate, but that use different cache structures. The manufactures did not provide you with any information on hat kind of cache organizations are used in these machines. Instead you ran a test program that measures how fast one memory read operation is for sequentially reading one array of a given size. You get the following result:



On the vertical axis is the number of cycles for one memory read instruction. The horizontal axis is the size of the test array in bytes.

1. How many levels of caching is used by computer **A** and how large are these caches?

2. How many levels of caching is used by computer **B** and how large are these caches?

3. A database program is run on these computers. The most time consuming part of this particular database uses a hash table of 250,000 entries of 8 bytes each. Which computer is faster?

4. Does the answer to part 3 changes if the hash table size is increased? If yes, at what size?

5. Based on this data, is it possible to tell how these caches are organized?