Andrew login ID:_____________________________________

Full Name:_____________________________________

Recitation Section:_____________________________________

CS 15-213/18-243, Fall 2009
Exam 2

Thursday, October 29th, 2009

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–J) on the front.
- **Do not write any part of your answers outside of the space given below each question. Write clearly and at a reasonable size. If we have trouble reading your handwriting you will receive no credit on that problem.**
- The exam has a maximum score of XXX 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. No calculators or other electronic devices are allowed.
- Good luck!

1 (21):	
2 (4):	
3 (15):	
4 (16):	
5 (12):	
6 (16):	
TOTAL (84):	

Problem 1. (21 points):

1. What is the most likely immediate result of executing the following code:

```
int foo[10]
int *p = (int *) malloc(4*sizeof(int));
p = p - 1;
*p = foo[0];
```

   (a) Initialize the first array element to 4
   (b) Segmentation fault
   (c) Reset the pointer p to point to the array named foo
   (d) Corruption of malloc header information

2. What is the maximum number of page faults per second that can be serviced in a system that has a disk with an average access time of 10ms?

   (a) 10
   (b) 100
   (c) 50
   (d) Depends on the percentage of memory accesses that are page faults

3. Why does Count Dracula not have to worry about his program's memory addresses overlapping those of other processes run on the same system?

   (a) Each process has its own page table
   (b) The linker carefully lays out address spaces to avoid overlap
   (c) The loader carefully lays out address spaces to avoid overlap
   (d) He does need to worry

4. Dr. Frankenstein has a disk that rotates at 7,200 RPM (8ms per full revolution), has an average seek time of 5ms, and has 1000 sectors per track. How long (approximately) does the average 1-sector access take?

   (a) Not enough information to determine the answer
   (b) 13ms
   (c) 9ms
   (d) 10.5ms

5. How many times does exec() return?
   (a) 0
   (b) 1
   (c) 2
   (d) 0 or 1, depending on whether or not an error occurs

6. Which of the following is **not** a default action for any signal type?
   (a) The process terminates.
   (b) The process reaps the zombies in the waitlist.
   (c) The process stops until restarted by a SIGCONT signal.
   (d) The process ignores the signal.
   (e) The process terminates and dumps core.

7. Imagine a process (called ”process A”) that calls fork() three times. If all three child processes terminate before process A is picked by the kernel to be run again, how many times could process A receive SIGCHLD?
   (a) 0
   (b) 1
   (c) 3
   (d) 1 or 3
   (e) Not enough information to determine

Problem 2. (4 points):

1. Consider the following program compiled for x86-64:

```
#include <malloc.h>

int main()
{
    int a = 0;
    int *b = malloc(sizeof(int));

    if ((&a) > b) {
        printf("Trick!\n");
    } else {
        printf("Treat!\n");
    }

    return 0;
}
```

   What does this program print out and why? (You can assume that the malloc() call does not fail)

Problem 3. (15 points):

You are provided with several files, each of which contains a simple text string without any whitespace or special characters. The list of files with their respective contents is given below:

one.txt	`abc`
two.txt	`nidoking`
three.txt	`conflageration`

You are also presented with the `main()` function of three small programs (header includes omitted), each of which uses simple and familiar functions that perform file i/o operations. For each program, determine what will be printed on `stdout` based on the code and the contents of the file. Assume that calls to `open()` succeed, and that each program is run from the directory containing the above files. (The program execution order does not matter; the programs are independent.)

Program 1:

```
void main() {
  char c0 = 'x', c1 = 'y', c2 = 'z';
  int r, r2 = open("one.txt", O_RDONLY);

  read(r2, &c0, 1);
  r = dup(r2);
  read(r2, &c1, 1);
  close(r2);
  read(r,  &c2, 1);

  printf("%c%c%c", c0, c1, c2);
}
```

output to `stdout` from Program 1:	

Program 2:

```
void main() {
  char c0 = ’x’, c1 = ’y’, c2 = ’z’;
  char scrap[4];
  int pid, r, r2 = open("two.txt", O_RDONLY);
  r = dup(r2);

  if (!(pid = fork())) {
    read(r, &c0, 1);
    close(r2);
    r2 = open("two.txt", O_RDONLY);
    read(r2, &scrap, 4);
  } else {
    waitpid(pid, NULL, 0);
    read(r,  &c1, 1);
    read(r2, &c2, 1);
  }

  printf("%c%c%c", c0, c1, c2);
}
```

output to `stdout` from Program 2:	

Program 3:

```
void main() {
  char c[3] = {’x’, ’y’, ’z’};
  int r, r2, r3;

  r  = open("three.txt", O_RDONLY);
  r2 = open("three.txt", O_RDWR);
  dup2(1, r3);
  dup2(r2, 1);

  read(r, &c[0], 1);
  printf("elephant");
  fflush(stdout);
  read(r,   &c[1], 1);
  read(r2,  &c[2], 1);
  write(r3, &c[0], 3);

  printf("%c%c%c", c[0], c[1], c[2]);
}
```

output to `stdout` from Program 3:	

Problem 4. (16 points):

Your evil TA Punter Hitelka has redesigned the fish machines to make buflab impossible! Normally, on x86 systems, a program's stack grows down, to lower memory addresses, making a called function have a **lower** stack address than the calling function. The new fish machines have stack frames that grow up, this means that a called function has a **higher** stack address than the calling function.

For example, under stack-down convention, having main() call foo() would create

```
0x0f0 +----------------+
      |  main's stack |
      |      frame    |
0x0e0 +----------------+
      |  foo's stack  |
      |      frame    |  | Stack Growing Down |
0x0d0 +----------------+  V                    V
```

Under the new stack-up convention, having main() call foo() would create

```
0x110 +----------------+
      |  foo's stack  | ^                   ^
      |     frame     | | Stack Growing Up |
0x100 +----------------+
      |  main's stack |
      |     frame     |
0x0f0 +----------------+
```

This means that a push instruction would increment %esp, and a pop instruction would decrement %esp. Bufflab now contains the following function, which Punter claims to be un-exploitable:

```
int exploitMe(){
    char password[100];

    /*prompt the user for the password*/
    printf("what is the password?\n");

    /*read it in*/
    gets(password);

    printf("You shall not pass!\n");
    return false;
}
```

1

First, let's go back to the old model of the stack growing down. Please draw a stack diagram from the perspective of the gets() function. Assume that main() calls exploitMe.

```
+--------------------------------------+
|          Ret Addr to Main            |
+--------------------------------------+
|                                      |  Stack Growing Down! |
|                                      |                      V
|                                      |
|                                      |
```

2

Describe a buffer overflow exploit you could use to make exploitMe return true if the stack grew down. You do not need to write the exploit, just describe how it would work.

3

Now, draw the stack diagram under the new stack-grows-up scheme. (Hint: this should be very easy given the answer to part 1). We will grade this based off your answer to part 1)

```
|                                      |
|                                      |                   ^
|                                      |  Stack Growing UP! |
+--------------------------------------+
|          Ret Addr to Main            |
+--------------------------------------+
```

4

Is it possible that Punter is wrong and this is exploitable? If so, please describe an exploit that would make exploitMe return true. Otherwise explain why it is impossible.

Problem 5. (12 points):

Consider the following C program, with line numbers:

```
1    int main() {
2      int counter = 0;
3      int pid;
4
5      if( !(pid = fork()) ) {
6        while((counter < 2) && (pid = fork()) ) {
7            counter++;
8            printf("%d", counter)
9        }
10       if (counter > 0) {
11           printf("%d", counter);
12       }
13     }
14     if(pid) {
15       waitpid(pid, NULL, 0);
16      counter = counter << 1;
17      printf("%d", counter)
18     }
19   }
```

Use the following assumptions to answer the questions:

- All processes run to completion and no system calls will fail.
- `printf()` is atomic and calls `fflush(stdout)` after printing argument(s) but before returning.
- Logical operators such as `&&` evaluate their operands from left to right and only evaluate the smallest number of operands necessary to determine the result.

A. List all possible outputs of the program in the following blanks.

(You might not use all the blanks.)

____________________ ____________________

____________________ ____________________

____________________ ____________________

____________________ ____________________

____________________ ____________________

B. If we modified line 10 of the code to change the > comparison to >=, it would cause the program flow to print out zero counter values. With this change, how many possible outputs are there?

(Just give a number, you do not need to list them all.)

`NEW NUMBER OF POSSIBLE OUTPUTS =` __________

Problem 6. (16 points):

Your friend Goger Ganberg was inspired by shell lab, and has, in an effort to better understand process management, written the following program:

```
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>

extern int do_stuff();

void magic()
{
    if (fork()) exit(0);
    signal(SIGHUP, SIG_IGN);
    setpgid(0, 0);
    chdir("/");
}

int main()
{
    magic();
    return do_stuff();
}
```

The main functionality of the program is implemented in `do_stuff`, defined in another file, but he seems more intent on showing off the `magic` function called beforehand.

Suppose that you run Goger's program in a shell whose process ID is 1000, and the program is assigned process ID 1001.

1. Assume the call to `fork` in this program returns 1002. What will be the PIDs of all processes that run `do_stuff` after `magic` returns, and for each one, what will be the PID of its parent?

2. Suppose `do_stuff` takes a long time to execute. Will the shell next emit a prompt only after `do_stuff` returns, or might it do so sooner? Why?

3. Assume now that `fork` instead returns -1. What will be different from the case in which `fork` succeeds? What will be the same?

The `SIGHUP` signal is delivered to a process when its controlling terminal goes away, and the default action is to exit. For example, if you open a terminal and run `vim` in it, then close the terminal, you would want your now-useless `vim` process to not stick around.

4. What is the purpose of the second and third lines of `magic`?

5. **Bonus question** (1 point): What sort of program would use a function like `magic`? (An example is sufficient.)

6. **Bonus question** (1 point): What else might it be useful for a function such as `magic` to do?