Full Name: _____

Roll Number: _____

IT215: Systems Software, Winter 2015-16
Second In-Sem Exam (1 hour 30 minutes)

March 16, 2016

Instructions:

- Make sure your exam is not missing any sheets, then write your name and roll number on the top of this page.
- Clearly write your answer in the space indicated. None of the questions need long answers.
- For rough work, do not use any additional sheets. Rough work will not be graded.
- Assume IA32 machine running Linux.
- The exam has a maximum score of 20 points. It is CLOSED BOOK. Notes are NOT allowed.
- Anyone who copies or allows someone to copy will receive F grade.

Problem 1 (/5):
Problem 2 (/3):
Problem 3 (/3):
Problem 4 (/3):
Problem 5 (/3):
Problem 6 (/3):
TOTAL (/20):

Problem 1. (5, 1 each points):

Circle the *single best* answer to each of the following questions. You will get -0.5 points for a wrong answer, so don't just guess wildly. If you circle more than one answer, you will lose the mark for the corresponding question.

1. Which of the following statements about shell commands is FALSE?

   (a) Built-in commands usually access state that the shell maintains.

   (b) Built-in commands can be used in a pipeline.

   (c) Built-in commands always run in the shell process the user is interacting with.

   (d) External commands always run in processes separate from the shell.

2. What happens when a thread calls `exit()`?

   (a) All threads in the same process terminate.

   (b) Nothing, multithreaded programs must use `pthread_exit`.

   (c) All threads are terminated except those that are waiting for a signal.

   (d) Only the calling thread terminates.

   (e) All threads are terminated except those that are waiting for I/O.

3. Spot the error! When run, the `go` function causes a segmentation fault during the `qsort` call. Assume `comp_fn` is correctly written. Which response best describes the bug that caused the segfault?

```
1   pthread_t tid;
2   void *result;
3   void *func(void *m) {
4       qsort(m, 100000, sizeof(int), comp_fn);
5       return NULL;
6   }
7   void go() {
8       void *mem = calloc(100000, sizeof(int));
9       pthread_create(&tid, NULL, func, mem);
10      free(mem);
11      pthread_join(tid, &result);
12  }
```

   (a) `qsort` cannot be used with heap memory

   (b) Line 11: `pthread_join` should be `pthread_exit`

   (c) Line 8 and 9 need to be swapped

   (d) `qsort` must not be called in a second thread

   (e) Line 10 and 11 need to be swapped

4. Which of the following is preserved across `exec`?

   (a) Signal handlers

   (b) Blocked signals

   (c) a and b

   (d) None of the above

5. 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 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.

Problem 2. (3 points):

Your machine comes with a simple memory allocator that uses an implicit free list. Each block in the heap has a 32-bit header that encodes the block size (including the header and any padding) along with a "used" bit that is set when the block is allocated. The block size is always a multiple of eight to satisfy the alignment requirement, and the "used" bit is stored in the least-significant bit of the header. The header is followed by the payload that the application requested when it called `malloc`.

We will implement a "blockinfo" command. This command takes one argument, a pointer `p`, and then prints information about the block pointed to by `p`. Namely, it prints the size of the block and whether or not the block is currently allocated. You can assume `p` always points to the payload area of an object. Your job is to finish the implementation of `printBlockInfo()`. You need to compute `size`, which is the size of the block, and `alloc` which is 1 if the block is allocated, and 0 otherwise. We have provided macros that you are required to use in implementing the above command. Do not modify the the given macros. Remember to cast your pointers correctly!

```
#define SIZE(w)   ((w) & ~0x7)
#define TAG_USED 1

void printBlockInfo(void *p) {
     int size;
     int alloc;

     // ------------ Your code here ---------------

     int header = ____________________________________;

     size = ____________________________________;

     alloc = ____________________________________;

     printf("   Block size: %d\n", size);
     printf("   Block allocated?: %d\n", alloc);

}
```

Problem 3. (3=1+2 points):

Assume the following program compiles and runs without error.

```
static int counter = 1;

static void *bar(void *ignore) {
    counter--;
    if (counter == 0) {
         fork();
         printf("hello\n");
    }
    return NULL;
}

static void foo()
{
     pthread_t t;
     if (fork() == 0) {
          fork();
          pthread_create(&t, NULL, bar, NULL);
          printf("hello\n");
          pthread_join(t, NULL);
     }
}

int main() {
    foo();
    printf("hello\n");
    return 0;
}
```

A. Under the assumption that all system calls succeed, how many times will the program print "hello"?

B. Draw a graph that illustrates the processes and threads that will be created when the program is run.

Problem 4. (3=1.5+1.5 points):

Consider the programs presented below.

Program P1:

```
int main() {
   int p[2];

   pipe(p);
   close(p[0]);

   if (fork()) {
      while (1) {
         sleep(1);
         write(p[1], "a", 1);
      }
   } else {
      char c;
      while (1) {
         read(p[0], &c, 1);
      }
   }
}
```

Program P2:

```
int BUFSIZE = 4096;

int main() {
   int fd[2];
   pid_t pid;
   char buffer[BUFSIZE];

   pipe(fd);
   if ((pid = fork()) == 0) {
      printf("child\n");
      write(fd[1], "a", 1);
      exit(0);
   }
   read(fd[0], buffer, BUFSIZE);
}
```

A. Consider the program P1 on the left-hand side of Figure. Here, the parent ought to send a stream of as, one per second. When you run the program, you make the following observations instead: (i) the program returns to the shell almost immediately; (ii) after a while you see that the system load is at 100%. Explain both phenomena.

B. Consider the program P2 on the right-hand side of Figure. Write the output of the program, assuming no errors occur. Can you guarantee the order of the output? If so, why? If not, why not?

Problem 5. (3=1.5+1.5 points):

A student has just started on her shell project and is trying to get foreground processes to run correctly. She has also started to flesh out her `SIGCHLD` handler. Changes so far are shown below.

```
void sigchld_handler(int sig) {
     while (waitpid(-1, NULL, WNOHANG) > 0) { ; }
}

void eval(char *cmdline) {
     char *argv[MAXARGS];
     pid_t pid;

     bg = parseline(cmdline, argv);
     if (!bg) {
          if ((pid = fork()) == 0)
               execvp(argv[0], argv);
          addjob(jobs, pid, FG, cmdline);
          waitfg(pid);
     }
}

void waitfg(pid_t pid) {
     int wpid;
     do {
         wpid = wait(NULL);
     } while (wpid != pid);
     deletejob(jobs, pid);
}
```

After compiling and running her shell, the student is able to enter a command and see its output, but unfortunately she doesn't get her shell prompt back and is unable to enter another command:

A. What is causing the observed bug? Be specific.

B. What would be a good way for the student to fix her code? (Keep in mind that she'll want to go on to complete the project, so the fix should be forward-looking.)

Problem 6. (3=1.5+1.5 points):

Suppose the file `foo.txt` contains the text "`123456`", `bar.txt` contains the text "`abcdef`", and `baz.txt` does not yet exist. Examine the following program, and answer the questions below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main() {
     int fd1, fd2, fd3, fd4, fd5, fd6;
     int status;
     pid_t pid;
     char c;

     /* foo.txt has "123456" */
     fd1 = open("foo.txt", O_RDONLY, 0);
     fd2 = open("foo.txt", O_RDONLY, 0);

     /* bar.txt has "abcdef" */
     fd3 = open("bar.txt", O_RDWR, 0);
     fd4 = open("bar.txt", O_RDWR, 0);

     /* baz.txt doesn't exist initially */
     fd5 = open("baz.txt", O_WRONLY | O_CREAT | O_TRUNC,
                           S_IRUSR | S_IWUSR); /* r/w */
     fd6 = dup(STDOUT_FILENO);
     dup2(fd5, STDOUT_FILENO);

     if ((pid = fork()) == 0) {
         dup2(fd3, fd2);
         read(fd3, &c, 1); printf("%c", c);
         write(fd4, "!@#$%^", 6);
         read(fd3, &c, 1); printf("%c", c);
         read(fd1, &c, 1); printf("%c", c);
         read(fd2, &c, 1); printf("%c\n", c);
         exit(0);
     }
     wait(NULL);
     read(fd1, &c, 1); printf("%c", c);
     fflush(stdout);
     dup2(fd6, STDOUT_FILENO);
     printf("done.\n");
     return 0;
}
```

A. What will the contents of `baz.txt` be after the program completes?

B. What will be printed on `stdout`?

Blank page for rough work.