CS360 Midterm Exam – February 25, 2010 – James S. Plank

Answer all questions.
Please answer on separate paper – do not put your answers ont the test.
Don't forget to put your name and email address on your answer sheets.

Question 1

Behold the following procedure:

```
int write_double_array(int fd, int *ids, double *vals, int n)
{
  int i;

  for (i = 0; i < n; i++) {
    if (write(fd, ids+i, sizeof(int)) != sizeof(int)) return -1;
    if (write(fd, vals+i, sizeof(double)) != sizeof(double)) return -1;
  }
  return 0;
}
```

Part A: Suppose *n* is large (over 1 million). Why is this procedure going to run exceptionally slowly? I don't want a one-sentence answer here – give me three or four sentences, and maybe even an equation!

Part B: Rewrite **write_double_array()** so that it runs much faster. You are *not* allowed to use the standard I/O library, and your procedure must have the same output as the original. Limit your extra memory usage to no more than 10,000 bytes.

Part C: Give me two examples of why **write_double_array()** would return -1.

Question 2

Explain what an atomic action is, and how the O_EXCL flag is used to perform an atomic action. Again, I want multiple sentences and detail – what is the action? How is the flag used to perform it? Why would we not be able to perform the action without the flag?

Question 3

Write the procedure **find_distinct_files()**, which has the following prototype:

```
int find_distinct_files(char *directory);
```

This procedure should print the number of distinct files in the given directory. Two files are distinct if they are not hard links to each other. Your program may ignore symbolic links. To be clear, you want the number of distinct files *in* the given directory, not *reachable* from the given directory.

Question 4

Suppose I have written a program **apstring** that takes one command line argument *string*, and appends *string* (with a newline) to the file **/home/plank/stringfile.txt**. Suppose I want to let other users run this program. Explain how I do this with and without the setuid bit. In your explanation, tell me the exact steps that I should do, and how the setuid bit works. Tell me why using the setuid bit is better than not using it.

CS360 Midterm Exam, Page 2

Question 5

Below is a program in **readfile.c**. Given the sequence of commands to the right, specify what the output will be with every **readfile** call. If you think that the output will depend on the machine/compiler, say so, but give the output that you know will happen.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

main(int argc, char **argv)
{
  char buffer[10];
  int i, fd;

  strcpy(buffer, "123456789");
  fd = open(argv[1], O_RDONLY);
  i = read(fd, buffer, 10);
  printf("%d %d %s\n", fd, i, buffer);
  exit(0);
}
```

```
UNIX> echo ABCD > f1.txt
UNIX> cat f1.txt
ABCD
UNIX> ln f1.txt f2.txt
UNIX> cp f2.txt f3.txt
UNIX> ln f3.txt f4.txt
UNIX> chmod 0 f4.txt
UNIX> echo ABCDEFGHIJKLMN > f5.txt
UNIX> cat f5.txt
ABCDEFGHIJKLMN
UNIX> readfile f1.txt
                             what is the output?
UNIX> readfile f2.txt
                             what is the output?
UNIX> readfile f3.txt
                             what is the output?
UNIX> readfile f4.txt
                             what is the output?
UNIX> readfile f5.txt
                             what is the output?
UNIX> readfile f6.txt
                             what is the output?
UNIX> readfile < f1.txt
                             what is the output?
```

Question 6

Behold the following program, in **umask_fun.c:**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

main(int argc, char **argv)
{
  int fd;
  char *s = "Hi\n";

  umask(022);

  fd = open("f1.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
  write(fd, s, 3); close(fd);

  fd = open("f2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0777);
  write(fd, s, 3); close(fd);

  fd = open("f3.txt", O_WRONLY | O_CREAT | O_TRUNC, 0777);
  write(fd, s, 3); close(fd);

  umask(276);

  fd = open("f4.txt", O_WRONLY | O_CREAT, 0666);
  write(fd, s, 3); close(fd);

  fd = open("f5.txt", O_WRONLY | O_CREAT, 0777);
  write(fd, s, 3); close(fd);

  fd = open("f6.txt", O_WRONLY | O_CREAT, 0777);
  write(fd, s, 3); close(fd);

  umask(444);

  fd = open("f7.txt", O_WRONLY, 0666);
  write(fd, s, 3); close(fd);

  fd = open("f8.txt", O_WRONLY, 0777);
  write(fd, s, 3); close(fd);

  fd = open("f9.txt", O_WRONLY, 0777);
  write(fd, s, 3); close(fd);
}
```

```
UNIX> ls -l f?.txt | awk '{ print $1, $5, $9 }'
-rw------- 5 f2.txt
-r-------- 5 f3.txt
-rw-r--r-- 5 f5.txt
-rw---x--x 5 f7.txt
-r-xr-xr-x 5 f8.txt
UNIX> sh -c 'umask_fun > /dev/null 2>&1'
UNIX> ls -l f?.txt | awk '{ print $1, $5, $9 }'
```

Above are three Unix commands. The first gives a long listing of the files **fx.txt**, printing only the permissions, sizes and names of the files (they are all 5 bytes in size). The second command runs **umask_fun**, suppressing standard output and standard error so that you don't see them. The third command is identical to the first.

Tell me what the output of the third command is.

Useful Typedefs and Prototypes

You don't need to worry about putting the proper include files in the code that you write.

```
struct stat {
          mode_t   st_mode;     /* File mode (see mknod(2)) */
          ino_t    st_ino;      /* Inode number */
          nlink_t  st_nlink;    /* Number of links */
          uid_t    st_uid;      /* User ID of the file's owner */
          gid_t    st_gid;      /* Group ID of the file's group */
          off_t    st_size;     /* File size in bytes */
          time_t   st_atime;    /* Time of last access */
          time_t   st_mtime;    /* Time of last data modification */
          time_t   st_ctime;    /* Time of last file status change */
         /* Plus some other stuff */
};

struct dirent {
        char d_name[256];    /* name must be no longer than this */
        /* Plus other stuff */
};

int     open(const char *path, int flags, mode_t mode);
int    close(int d);
size_t  read(int d, const void *buf, size_t nbytes);
size_t write(int d, const void *buf, size_t nbytes);
int     stat(const char *path, struct stat *sb);
mode_t umask(mode_t numask);

DIR           *opendir(const char *filename);
struct dirent *readdir(DIR *dirp);
long           telldir(DIR *dirp);
void           seekdir(DIR *dirp, long loc);
void         rewinddir(DIR *dirp);
int           closedir(DIR *dirp);

extern JRB make_jrb();   /* Creates a new rb-tree */

extern JRB jrb_insert_str(JRB tree, char *key, Jval val);
extern JRB jrb_insert_int(JRB tree, int ikey, Jval val);
extern JRB jrb_insert_dbl(JRB tree, double dkey, Jval val);
extern JRB jrb_insert_gen(JRB tree, Jval key, Jval val, int (*func)(Jval,Jval));

/* The following return NULL if there is no such node in the tree */

extern JRB jrb_find_str(JRB root, char *key);
extern JRB jrb_find_int(JRB root, int ikey);
extern JRB jrb_find_dbl(JRB root, double dkey);
extern JRB jrb_find_gen(JRB root, Jval, int (*func)(Jval, Jval));
```