CS360 Exam #1: March 10, 1995

Question 1: 15 points

Behold the following piece of code:

```
#include <stdio.h>

main()
{
  int *ip[5];
  int i;

  printf("A\n");
  for (i = 0; i < 5; i++) {
    ip[i] = subr(i);
  }
  printf("B\n");
  for (i = 0; i < 5; i++) {
    printf("*ip[%d] = %d\n", i, *ip[i]);
  }
}
```

For each of implementations **A**, **B**, **C**, **D** and **E** below, show the output of that implementation of `subr()` when executed with the above `main()`. If the output needs explanation, make sure you give it. If you are not sure about exactly what the output will be, say so and why. If there is more than one possible output, say what they are and why. In other words, be as exact as possible.

Implementation A

```
int *subr(int i)
{
  int *ptr;

  ptr = (int *) i;
  return ptr;
}
```

Implementation B

```
int *subr(int i)
{
  int *ptr;

  i = random();
  ptr = (int *) i;
  return ptr;
}
```

random() returns a random integer

Implementation C

```
int *subr(int i)
{
  int *ptr;

  ptr = (int *) malloc(sizeof(int));
  ptr = &i;
  return ptr;
}
```

Implementation D

```
int *subr(int i)
{
  char *j;
  char *k;
  int *ptr;
  int tmp;

  j = (char *) malloc(sizeof(int));
  k = (char *) (&i);
  for (tmp = 0; tmp < sizeof(int); tmp ++) {
    j[tmp] = k[tmp];
  }
  ptr = (int *) (j);
  return ptr;
}
```

Implementation E

```
int *subr(int i)
{
  char *j;
  char *k;
  int *ptr;
  int tmp;

  j = (char *) malloc(sizeof(char)*i
                      +sizeof(int));
  k = (char *) (&i);
  for (tmp = 0; tmp < sizeof(int); tmp ++) {
    j[i+tmp] = k[tmp];
  }
  ptr = (int *) (j+i);
  return ptr;
}
```

Question 2: 20 points

Suppose I've written a single-player game program (like, say, Tetris), called `jtetris`. Anyone here at UT can play the game by running the program:

```
UNIX> /mahogany/homes/plank/cs360/bin/jtetris
```

Of course, I'd like to print out the top ten scores at the conclusion of each gmae. To do this, I'll maintain a file `/mahogany/homes/plank/jt/scores` which contains this information. This file consists of lines of the form:

```
USERNAME  SCORE
```

where `USERNAME` is the username of the user who played the game, `SCORE` is that user's score for that game. This file will have a maximum of ten lines.

Part 1 (5 Points)

What should the protection modes of `jtetris` and `scores` be and why?

Part 2 (9 Points)

On the next page are two implementations of `log_score()`, which prints out the top ten high scores (including the user's if it is high enough), and updates the high score file. `Log_score2()` is a better implementation than `log_score1()`. Why? In your answer, include how `log_score1()` can fail, and how `log_score2()` fixes this problem. Your answer should point out where the atomic actions are in each implementation, and how they work to achieve their goal.

Part 3 (4 Points)

`Log_score2()` suggests a way to perform a certain kind of generalized atomic actions. How does this work, and what are its limitations?

Part 4 (2 Points)

How can `log_score2()` still fail?

Implementations of `log_score1()` and `log_score2()`

```
#include <fcntl.h>
#include <stdio.h>
#include "rb.h"

#define SF "/mahogany/homes/plank/jt/scores"

log_score1(char *username, int score)
{
  FILE *f;
  char s[100], un[10];
  Rb_node r, tmp;
  int sc, nscores, fd;

  r = make_rb();
  rb_inserti(r, score, username);

  f = fopen(SF, "r");
  if (f != NULL) {
    while(fscanf(f, "%s %d", un, &sc) != EOF) {
      rb_inserti(r, sc, strdup(un));
    }
  }
  fclose(f);

  fd = open(SF, O_APPEND | O_SYNC | O_WRONLY |
              O_CREAT | O_TRUNC, 0644);
  if (fd < 0) { perror(SF); exit(1); }

  printf("High Scores:\n\n");
  nscores = 0;
  for (tmp = rb_last(r);
       tmp != r && nscores < 10;
       tmp = rb_prev(tmp))
  {
    sprintf(s, "%-8s %8d\n",
           tmp->v.val, tmp->k.ikey);
    printf("%s", s);
    write(fd, s, strlen(s));
    nscores++;
  }
  close(fd);
}
```

```
#include <fcntl.h>
#include <stdio.h>
#include "rb.h"

#define SF "/mahogany/homes/plank/jt/scores"
#define CF "/mahogany/homes/plank/jt/cntl"
#define WCE (O_WRONLY | O_CREAT | O_EXCL)
#define WCT (O_WRONLY | O_CREAT | O_TRUNC)

log_score2(char *username, int score)
{
  FILE *f;
  char s[100], un[10];
  Rb_node r, tmp;
  int sc, nscores, fd;

  r = make_rb();
  rb_inserti(r, score, username);

  for (fd = open(CF, WCE, 0644);
       fd < 0;
       fd = open(CF, WCE, 0644)) ;
  close(fd);

  f = fopen(SF, "r");
  if (f != NULL) {
    while(fscanf(f, "%s %d", un, &sc) != EOF) {
      rb_inserti(r, sc, strdup(un));
    }
  }
  fclose(f);

  fd = open(SF, WCT, 0644);
  if (fd < 0) { perror(SF); exit(1); }

  printf("High Scores:\n\n");
  nscores = 0;
  for (tmp = rb_last(r);
       tmp != r && nscores < 10;
       tmp = rb_prev(tmp))
  {
    sprintf(s, "%-8s %8d\n",
           tmp->v.val, tmp->k.ikey);
    printf("%s", s) ;
    write(fd, s, strlen(s));
    nscores++;
  }
  close(fd);

  unlink(CF);
}
```

Question 3: 15 points

At the bottom of this page is a `main()` procedure for the program `jfind`, which works as follows. It accepts exactly one command line argument `fn`. It then prints out the relative path names of any files named `fn` which are reachable from the current directory. `jfind` ignores soft links.

For example, suppose I am in the directory `/mahogany/homes/plank/cs360`:

```
UNIX> jfind lecture
./notes/01-11/lecture
./notes/01-13/lecture
./notes/08-26/lecture
./notes/08-24/lecture
./notes/08-31/lecture
./notes/01-25/lecture
./notes/01-27/lecture
./notes/02-01/lecture
./notes/02-03/lecture
./notes/02-08/lecture
./notes/02-10/lecture
./notes/02-15/lecture
./notes/02-17/lecture
./notes/02-22/lecture
./notes/02-24/lecture
./notes/03-01/lecture
./notes/03-03/lecture
UNIX>
```

Your job is to write the procedure `findfile()`.

All of the system and library calls that you may need for this program can be found on the cheet sheet. You may assume that no single filename is greater than 256 characters. However, you should not make any other assumptions (i.e. you should not assume that the relative path name is less than a fixed size, and you should not assume that the number of nested directories is less than a fixed number. Your program also should not have any bad memory leaks).

(Of course, if you're in a panic and don't have enough time, then go ahead and make the above assumptions that I tell you not to make — it's more important to write down something that works in most cases than to freeze up and write down nothing because you can't figure out how to prevent memory leaks...)

On the next page is a skeleton for `findfile()` that you can use to hand in your answer.

```
#include <stdio.h>

main(int argc, char **argv)
{
  if (argc != 2) {
    fprintf(stderr, "usage: jfind filename\n");
    exit(1);
  }
  findfile(argv[1], ".");
}
```

Below is a skeleton for `findfile()`. Feel free to fill it in and hand this sheet as the answer to Question 1.

```
/* You should not need any include files other than these, but if you do go ahead and add them  */
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include "dlist.h"

findfile(char *fn, char *dir)
{

}

main(int argc, char **argv)
{
  if (argc != 2) {
    fprintf(stderr, "usage: jfind filename\n");
    exit(1);
  }
  findfile(argv[1], ".");
}
```

From rb.h:

```
typedef struct rb_node {
  union {
    struct {
      struct rb_node *flink;
      struct rb_node *blink;
    } list;
  } c;
  union {
    int ikey;
    char *key;
  } k;
  union {
    char *val;
  } v;
} *Rb_node;
```

- `Rb_node make_rb()`: Mallocs and returns a new rb-tree.
- `void rb_inserti(Rb_node t, int ikey, char *val)`: Creates and inserts a new rb-node into the tree `t` with key `ikey` and val `val`. The key is an integer.
- `Rb_node rb_find_ikey_n(Rb_node t, int ikey, int *found)`: Finds and returns a node in the tree `t` with integer key `ikey`. If it finds such a node, then it sets `*found` to 1. If not, it sets `*found` to 0.
- `#define rb_nil(t) (t)`
- `#define rb_first(t) (t->c.list.flink)`
- `#define rb_next(t) (t->c.list.flink)`
- `#define rb_last(t) (t->c.list.blink)`
- `#define rb_prev(t) (t->c.list.blink)`
- Tree Traversal Macro:

```
#define rb_traverse(ptr, lst) \
    for(ptr = rb_first(lst); ptr != rb_nil(lst); \
                             ptr = rb_next(ptr))
```

From dirent.h / the opendir man page:

```
#include <dirent.h>

DIR *opendir(dirname)
char *dirname;

struct dirent *readdir(dirp)
DIR *dirp;

int closedir(dirp)
DIR *dirp;


struct  dirent {
        long            d_off;          /* offset of next disk dir entry */
        unsigned long   d_fileno;       /* file number of entry */
        unsigned long   d_reclen;       /* length of this record */
        unsigned long   d_namlen;       /* length of string in d_name */
        char            d_name[255+1];  /* name (up to MAXNAMLEN + 1) */
};
```

From string.h:

You should already know what these do:

- `char *strcmp(char *s1, char *s2);`
- `char *strcpy(char *s1, char *s2);`
- `char *strdup(char *s);`
- `int *strlen(char *s);`

From dlist.h:

```
typedef struct dlist {
  struct dlist *flink;
  struct dlist *blink;
  void *val;
} *Dlist;
```

- `Dlist make_dl()`: Mallocs and returns a new dlist
- `void dl_insert_b(Dlist d, void *val)`: Creates a new dlist node with val `val` and inserts it into the list before node `d`. If `d` is the head of the list, the new node is inserted at the end of the list.
- `void dl_delete_list(Dlist d)`: Deletes and frees the dlist whose head is the node `d`. Note that it does *not* free the `val` fields.
- Tree Traversal Macro:

```
#define dl_traverse(ptr, lst) \
    for(ptr = lst->flink; ptr != lst; ptr = ptr->flink)
```

From the stat man page

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(char *path, struct stat *buf)

int lstat(char *path, struct stat *buf)

struct stat {
    dev_t     st_dev;
    ino_t     st_ino;
    mode_t    st_mode;
    nlink_t   st_nlink;
    uid_t     st_uid;
    gid_t     st_gid;
    dev_t     st_rdev;
    off_t     st_size;
    time_t    st_atime;
    time_t    st_mtime;
    time_t    st_ctime;
    long      st_blksize;
    long      st_blocks;
}
```

The following macros test whether a file is of the specified type. The value `m` is the value of `st_mode`. Each macro evaluates to a non-zero value if the test is true or to zero if the test is false.

- `S_ISDIR(m)`: Test for directory file.
- `S_ISREG(m)`: Test for regular file.
- `S_ISLNK(m)`: Test for a symbolic link.

From the unlink man page

```
int unlink(char *path)
```

DESCRIPTION: `unlink()` removes the directory entry named by the pathname pointed to by `path` and decrements the link count of the file referred to by that entry. If this entry was the last link to the file, and no process has the file open, then all resources associated with the file are reclaimed.

From the open man page

```
#include <fcntl.h>

int open(char *path, int flags[ , int mode ] )

     path points to the pathname of a  file.   open()  opens  the
     named  file  for reading and/or writing, as specified by the
     flags argument, and returns a descriptor for that file.  The
     flags  argument may indicate the file is to be created if it
     does not already exist (by specifying the O_CREAT flag),  in
     which  case  the file is created with mode mode as described
     in chmod(2V) and modified by the process' umask  value  (see
     umask(2V)).  flags values  are  constructed  by  ORing flags
     from the following list (one and only one of the first three
     flags  below  must be used):

     O_RDONLY       Open for reading only.

     O_WRONLY       Open for writing only.

     O_RDWR         Open for reading and writing.

     O_SYNC         When  opening  a  regular  file,  this  flag
                    affects  subsequent  writes.   If  set,  each
                    write(2V) will wait for both  the  file  data
                    and file status to be physically updated.

     O_APPEND       If set, the seek pointer will be set  to  the
                    end of the file prior to each write.

     O_CREAT        If the file exists, this flag has no  effect.
                    Otherwise, the file is created, and the owner
                    ID of the file is set to the  effective  user
                    ID  of the process.

     O_TRUNC        If the file exists and is a regular file, and
                    the  file  is  successfully  opened O_RDWR or
                    O_WRONLY, its length is truncated to zero and
                    the  mode  and  owner are unchanged.

     O_EXCL         If O_EXCL and O_CREAT are  set,  open()  will
                    fail if the file exists.
```

Syntax of close and write

```
int close (int fd)

int write(int fd, char *buf, int nbyte)
```