CS360 Exam #1: October 19, 1994

Question 1: 12 points

Explain buffering file I/O: What is it? Why is it a good idea? Give an example. (You don't have to write code for the example, but may if you'd like)

Question 2: 14 points

Part 1 (3 points): Explain the *setuid* bit. What is it and why does it exist?

Part 2 (6 points): Suppose I perform the following actions:

```
UNIX> cp /bin/rm .    ; chmod 04755 rm          ; sleep 3600 ; /bin/rm rm
UNIX> cp /bin/sh .    ; chmod 04755 sh          ; sleep 3600 ; rm sh
UNIX> cp /bin/chmod . ; /bin/chmod 04755 chmod  ; sleep 3600 ; rm chmod
UNIX> cp /usr/local/bin/gcc . ; chmod 04755 gcc ; sleep 3600 ; rm gcc
```

What these lines do is the following: For each of the programs `rm`, `sh`, `chmod` and `gcc`, it makes a copy of the program in my home directory, sets the *setuid* bit of the program, and waits for an hour before removing the program.

Suppose we define an "unsafe" state to be one in which a malignant user can overwrite or delete some or all of my files, regardless of how they are protected.

Each of these lines leaves me in an unsafe state for an hour. For each line, how can the malignant user delete/overwrite my files?

Part 3 (3 points): The line:

```
UNIX> cp /bin/sh .    ; chmod 04755 sh          ; sleep 3600 ; rm sh
```

has the potential to leave me in an unsafe state *forever*! In other words, before the hour is over, an a malignant user can do something that lets him/her delete/overwrite files even after the hour is over. Explain how he/she can do that.

Part 4 (2 points):

Similarly,

```
UNIX> cp /bin/chmod . ; /bin/chmod 04755 chmod  ; sleep 3600 ; rm chmod
```

can do the same thing. Again, explain how.

Question 3: 14 points

Behold the man page for `time()`:

```
SYNOPSIS
     long time(long *tloc)

DESCRIPTION
     time() returns the time since 00:00:00 GMT,  Jan.  1,  1970,
     measured in seconds.

     If tloc is non-NULL, the return value is also stored in  the
     location to which tloc points.
```

Now, look at the code below (we don't know what `proc1()` does, but we assume that it is bug-free):

```
extern int proc1(int a, int b);
long t1;

b(int j, int k)
{
  int i;

  i = proc1(j, k);
  t1 = time();
  return i;
}
```

Part 1 (3 points): There is a bug in this code. What is it?

Part 2 (6 points): Convert the code for procedure `b()` into assembly code. (We're assuming here that the bug is not caught by the compiler).

Part 3 (5 points): What are all the possible outcomes of running this code? Make sure explain how these outcomes come about. Which ones will make this bug easy to find using a debugger?

Question 4: 10 points

Below is the code for **simpls**, a program that lists the contents of the current directory in no particular order:

```
#include <dirent.h>
#include <stdio.h>

main()
{
  DIR *dirp;
  struct dirent *d;

  dirp = opendir(".");
  if (dirp == NULL) {
    perror("myls: .");
    exit(1);
  }

  for (d = readdir(dirp); d != NULL; d = readdir(dirp)) {
    printf("%s\n", d->d_name);
  }
  closedir(dirp);
}
```

Write a program `lslen` to list the contents of the current directory, sorted by the length of the filename (if two filenames have the same length, then they can be printed in any order). For example, if the contents of a directory are:

```
UNIX> ls
main.aux        main.log        main.toc        title.tex
main.dvi        main.tex        title.jgr
UNIX>
```

Then **simpls** and `lslen` work as follows:

```
UNIX> simpls
.
..
title.jgr
mm
main.tex
main.log
main.aux
tc
title.tex
UNIX> lslen
.
tc
mm
..
main.log
main.aux
main.tex
title.tex
title.jgr
UNIX>
```

To help you out, there is a skeleton of `lslen.c` on the next page. Feel free to write on it and hand that in.

If you are in a panic and feel that you don't have time to do this, describe what you think should be done. That will be better than nothing, and better than garbage.

Below is a skeleton for `lslen.c`. Feel free to fill it in and hand this sheet as the answer to Question 4. You will have to add more `#include`'s and variables.

```
#include <dirent.h>
#include <stdio.h>



main()
{
  DIR *dirp;
  struct dirent *d;

}
```

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)`
- 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);`