Test 1 (March 1994) Answers

Question 1

At the arrow, the state of the directory is as follows: Besides the standard "." and ".." directories, there are three files: `f1`, `f2`, and `f3`. `F1` and `f2` are hard links to the same file, and `f3` is a soft link to `f1`. In other words, there is one file on disk, whose contents are `"This is file 1\n"`. There is one inode that governs this file, and two links `f1` and `f2` to this inode. The soft link is maintained by Unix to point to `f1`, but isn't included as a link in the file's inode.

The first `rm` command removes the link for the file `f1`. This does nothing to the contents of the file, but does decrement the link count in the inode to a value of one.

The second `rm` command removes the link for the file `f2`. As this is the last link to the inode (this is discovered by the link count in the inode being 1), the inode itself is destroyed, and the disk space that the file occupies is freed. Note that the soft link `f3` still exists, but if it were to be referenced, Unix would flag an error since `f1` is gone.

The last `rm` command removes the soft link `f3`. The directory is now empty.

Question 2

A good example of an atomic action is opening a file using

```
open(fn, O_CREAT | O_EXCL, ...)
```

This means to open and create the file only if it does not exist already. The atomicity comes in this system call doing two things: checking for the file existing, and opening it only if it doesn't. We need this atomicity, for if we had to do this with two system calls (i.e. do a `stat` followed by an `open`, or an `open` followed by an `open`), then the state of the system might change in between the two calls, meaning that we might create and open a file that already exists. Making this functionality atomic is the only way to assure correctness in this case.

Another example of atomicity was when you perform a `write` after opening a file with:

```
open(fn, O_WRONLY | O_APPEND | O_SYNC, ...)
```

Here, the atomicity is in performing two actions – this time seeking for the end of file, and then writing there – at once. Otherwise, the end of file could change after the seek, leading to a `write` that is not an append.

Question 3

`File_to_dlist_2` is more efficient because it uses the standard I/O library, which buffers input. `File_to_dlist_1` makes the system call `read()` once for every integer in the file. `File_to_dlist_2` makes a procedure call which copies the input from a buffer, and only calls `read()` when the buffer is empty. As the buffer is larger than one integer (usually something like 4Kbytes), it saves the user on the order of 1000 system calls from `file_to_dlist_1`.

In order to write an efficient `file_to_dlist_3`, we must provide our own buffering. By this, we read in the number of integers, and then `malloc` space for all of them. Then we read all of them from the file in one system call, and then create the dlist. When we're done, we call `free` to free up the allocated memory. The code is below:

```
Dlist file_to_dlist_3(char *fn)
{
  int n_ints;
  Dlist d;
  int i;
  int fd;
  int *buffer;

  fd = open(fn, O_RDONLY);

  d = make_dl();
  read(fd, &n_ints, sizeof(int));
  buffer = (int *) malloc(sizeof(int) * n_ints);
  read(fd, buffer, sizeof(int) * n_ints);
  for (i = 0; i < n_ints; i++) {
    dl_insert_b(d, buffer[i]);
  }
  close(fd);
  free(buffer);
  return d;
}
```

Some of you tried to use a fixed-size buffer for this. That was harder to do, but not impossible. Something like this would work:

```
#define BSIZE 4096
Dlist file_to_dlist_3(char *fn)
{
  int n_ints, to_read;
  Dlist d;
  int i;
  int fd;
  int buffer[BSIZE];

  fd = open(fn, O_RDONLY);

  d = make_dl();
  read(fd, &n_ints, sizeof(int));
  while (n_ints > 0) {
    if (n_ints > BSIZE) {
      to_read = BSIZE;
    } else {
      to_read = n_ints;
    }
    read(fd, buffer, sizeof(int) * to_read);
    for (i = 0; i < to_read; i++) {
      dl_insert_b(d, buffer[i]);
    }
    n_ints -= to_read;
  }
  close(fd);
  return d;
}
```

Question 4

The left one is incorrect, and the right one is correct, but `main()` will not work with it. First, the left `get_line()` returns a pointer to temporary storage. In other words, when `main()` gets the return value of `get_line()`, that return value is no longer valid, as the temporary storage has gone away. This will be a hard bug to track,

The right version of `get_line()` is ok, but `main()` uses it incorrectly. Specifically, `get_line()` returns a pointer to a global variable `s`. Right after each call to `get_line()`, `s` holds the next line. Unfortunately, each time `main()` calls `rb_insert()`, it will call it with the value of `s`, which means that at the end of input, all nodes of the red-black tree will point to the same character string: `s`. This is a memory bug made by a bad assumption in the procedure `main()`. It has been described in a few lectures. To fix the bug, we must `malloc` space for each line in the red-black tree. The correct `main()` is below:

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

char *get_line();

main()
{
  Rb_node r, tmp;
  char *line;
  char *l_copy;

  r = make_rb();

  for (line = get_line(); line != NULL; line = get_line()) {
    l_copy= (char *)malloc(sizeof(char) * (strlen(line)+1));
    strcpy(l_copy, line);
    rb_insert(r, l_copy, NULL);
  }
  rb_traverse(tmp, r) {
    printf("%s", tmp->k.key);
  }
}
```