CS360 Exam #1: March 9, 1994

Question 1: 10 points

Consider the following commands. Describe the state of the directory (in terms of filenames, inodes, and disk storage) at the arrow. Explain what happens to this state during each `rm` command. Include when the disk storage associated with this file is freed. How does Unix know to free the disk storage when it does?

```
UNIX> ls
UNIX> cat > f1
This is file 1
<CNTL-D>
UNIX> ln f1 f2
UNIX> ln -s f1 f3
                                  <----------------------
UNIX> rm f1
UNIX> rm f2
UNIX> rm f3
```

Question 2: 10 points

Give an example of an atomic action as described in class. Why is the atomicity of this action important?

Question 3: 15 points

Below are two implementations of the same function:

```
#include <fcntl.h>
#include "dlist.h"

Dlist file_to_dlist_1(char *fn)
{
  int n_ints;
  Dlist d;
  int i, j;
  int fd;

  fd = open(fn, O_RDONLY);

  d = make_dl();
  read(fd, &n_ints, sizeof(int));
  for (i = 0; i < n_ints; i++) {
    read(fd, &j, sizeof(int));
    dl_insert_b(d, j);
  }
  close(fd);
  return d;
}
```

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

Dlist file_to_dlist_2(char *fn)
{
  int n_ints;
  Dlist d;
  int i, j;
  FILE *f;

  f = fopen(fn, "r");

  d = make_dl();
  fread(&n_ints, sizeof(int), 1, f);
  for (i = 0; i < n_ints; i++) {
    fread(&j, sizeof(int), 1, f);
    dl_insert_b(d, j);
  }
  fclose(f);
  return d;
}
```

Part a: (10 points) Which one will be more efficient if the file being read is large, and why? (Answer the 'why' part with a few sentences, not one or two words)

Part b: (5 points) Write the procedure `file_to_dlist_3`, which is functionally equivalent to these two functions, and is at least as efficient as the best of these two. You may only use `open()`, `close()`, and `read()`, and not `fopen()`, `fclose()` and `fread()`. Explain why this is efficient code. You may assume that the file being read is never bigger than the amount of available physical memory.

Question 4: 12 points

The following code reads lines of text on standard input, and then prints them sorted on standard output. It assumes that `get_line()` returns a pointer to the next line of standard input (terminated by a newline and the NULL character), or `NULL` if it has reached the end of standard input.

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

char *get_line();

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

  r = make_rb();

  for (line = get_line(); line != NULL; line = get_line()) {
    rb_insert(r, line, NULL);
  }
  rb_traverse(tmp, r) {
    printf("%s", tmp->k.key);
  }
}
```

Below are two implementations of `get_line()`. Neither works with the `main()` program above. One is totally broken, and the other is simply used incorrectly by `main()`. State which one (left or right) is broken and why, and fix the `main()` program so that it works with the other one.

It is assumed that no line is longer than 1000 characters, and that the last line will end with a newline.

```
#include <stdio.h>

char *get_line()
{
  char s[1000];
  int i;
  char c;

  i = 0;
  while(1) {
    c = getchar();
    if (c == EOF) return NULL;
    s[i] = c;
    i++;
    if (c == '\n') {
      s[i] = `\0`;
      return s;
    }
  }
}
```

```
#include <stdio.h>
char s[1000];

char *get_line()
{
  int i;
  char c;

  i = 0;
  while(1) {
    c = getchar();
    if (c == EOF) return NULL;
    s[i] = c;
    i++;
    if (c == '\n') {
      s[i] = `\0`;
      return s;
    }
  }
}
```