CS360 Final Exam: May 7, 1994

Question 1: 20 Points

Behold the man page for `jpopen()`:

```
NAME
     jpopen, jpclose - open or close a pipe (for I/O) from and to a process

SYNOPSIS
     int jpopen(char *command)
     int jpclose(int fd)

DESCRIPTION
      The argument to jpopen() is a pointer to a null-terminated shell command.  jpopen()
      executes the command and returns a file descriptor.  The command is hooked up to its
      calling process in the following way: Standard input of the command should come from
      standard output of the calling process.  Standard output of the command should be
      readable by the calling process using the return value of jpopen() as a file descriptor.

      jpclose() should be called on the return value of jpopen().  This will wait for the
      command to complete, close all related open files, and restore the calling process's
      standard output fd to what it was before jpopen() was called.   jpclose() returns
      the status of the command executed by jpopen().

      You may assume that jpopen() is never called twice without an intervening jpclose().
```

Implement `jpopen()` and `jpclose()` in C.

The correct incantation of `exec` for calling a shell command is:

```
execlp("/bin/sh", "sh", "-c", command, (char *)0);
```

Question 2: 25 Points

You've written a totally cool game program in C called `game`. It has a simple interface – it is interactive, with the user giving input on standard input, and receiving output on standard output. When it's done, it writes information to the high score file `/red/homes/game/high-scores`. You would like to export this program so that as many people as possible can play it. If possible, you would like everyone who ever plays the game to update the same high score file. You are considering the following four ways of doing this:

1. Exporting the executable.
2. Keeping one executable in a publically available place with the `setuid` bit set.
3. Writing an iterative `game` server to which users connect using `telnet`.
4. Writing a concurrent `game` server to which users connect using `telnet`.

For each method above, discuss the pros and cons in terms of:

- How a user runs the program.
- How the high score file stays maintained.
- Security.
- System resources.
- Availability.

There is a better method that combines the best of all the above. Explain this design.

Question 3: 25 Points (10/5/10)

```
typedef void *jmp_buf[10];

int setjmp(jmp_buf env)
void longjmp(jmp_buf env, int val)
```

Part 1: Describe `setjmp()` and `longjmp()`. What do they do, and how are they used?

Part 2: What is the output of the program below:

```
#include <setjmp.h>

f(jmp_buf b)
{
  longjmp(b, 1);
}

main()
{
  jmp_buf b;
  int a;

  a = setjmp(b);
  printf("a=%d\n", a);
  if (a == 0) f(b);
  printf("a=%d.  Done\n", a);
}
```

Part 3: Below is an assembly code implementation of `setjmp()`. The machine is as described in class, with 10 general purpose registers, all of which hold 4 bytes, and 3 special registers: the stack pointer (`sp`), frame pointer (`fp`), and program counter (`pc`). As described in class, "`jsr a`" pushes (`pc`+4) and `fp` on the stack, sets the `fp` to the `sp`, and sets the `pc` to `a`. Thus, in procedure `a`, the old frame pointer is in `[fp+4]`, and the saved program counter is in `[fp+8]`. The procedure's arguments start at `[fp+12]`. "`ret`" restores the saved frame pointer, and saved program counter, and pops them off the stack. The return value of the procedure should be in register `r0`.

```
setjmp:
    ld [fp+12] -> %r0       / Put jmp_buf into r0
    st %r1 -> [r0]          / Save the general purpose registers
    st %r2 -> [r0+4]        / (except r0)
    st %r3 -> [r0+8]
    st %r4 -> [r0+12]
    st %r5 -> [r0+16]
    st %r6 -> [r0+20]
    st %r7 -> [r0+24]
    st %r8 -> [r0+28]
    st %r9 -> [r0+32]
    ld [fp+4] -> %r1        / Save the old fp into jmp_buf[9]
    st %r1 -> [r0+36]
    ld [r0] -> %r1          / Restore the value of r1
    mv 0 -> %r0             / Return 0
    ret
```

Implement `longjmp()` in assembly code. You may assume that `longjmp(buf, 0)` is never called.

If you cannot figure out how to implement `longjmp()` given the `setjmp()` above, you can alter `setjmp()` so that your `longjmp()` will work, and only lose a few points. If you are still stuck, think about the state of the machine when the program in **Part 2** above calls `longjmp(b)`. What has to be done in order for the `longjmp(b)` to return to the right place? Think about what "`ret`" does, and how you can get it to work for you. If you still can't implement `longjmp()`, describe as best you can what has to be done so you can get some partial credit.

Question 4: 30 points: 10/10/10

Look at the implementation of the procedures `malloc8()` and `free8()` on the next page.

Part 1: `Malloc8()` and `free8()` allocate and free 8-byte chunks of memory. Describe how they work. Use English and give an overview of what happens when `malloc8()` and `free8()` are called. I do not want to see anything like "`malloc8` first checks to see if `freelist8` is `NULL`, and if so sets `freelist8` to the return value of `make_dl()` and ..."

Part 2: Discuss fragmentation in `malloc8()` and `free8()`. Suppose you are writing a program that is going to allocate and free megabytes worth of 8-byte chunks. Moreover, suppose Unix's `malloc()` calls `sbrk()` in chunks of 8192 bytes. What will be the fragmentation if you call `malloc8()` and `free8()` as opposed to `malloc(8)` and `free()`? For this you can assume that `Dlists` take up 12 bytes per node, and `rb-trees` take up 56 bytes per node.

Part 3: What is the worst case running time of a call to `malloc8()`, and a call to `free8()`. Explain why the red black tree is needed.