CS360 Final Exam: 15 December, 1994

You have 2 hours

There are 85 total points

Answer all questions

Note that the questions are in increasing order of difficulty. Read over all the questions before plunging in so that you can budget your time wisely.

Question 1: 15 Points (three parts)

The grading for this question is going to be very strict.

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 (7 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 2: 15 Points (one part)

In the `csh` (and in the Bourne shell), the semi-colon (`;`) is an operator that lets you put more than one command on a line. For example,

```
UNIX> make a.out; date; a.out > output
```

Is equivalent to:

```
UNIX> make a.out
UNIX> date
UNIX> a.out > output
```

Write a program that executes:

```
UNIX> a.out < input | sort > output ; ls
```

Your program may call any subroutines/system calls listed on the cheat sheet. It may *not* call `system()` or `popen()`.

Question 3: 25 Points (four parts)

Part 1 (5 points)

Look at the following implementation of `malloc()` and `free()`:

```
char *malloc(int n)
{
  return sbrk(n);
}

void free(char *p)
{
  return;
}
```

This code will compile and run decently with many applications. However, it will not run well with all applications. Specifically, there are two big problems with this implementation. Describe them and why they are problems. (If you see more than two problems go ahead and describe them – you won't get extra credit, but you might be more likely to describe the two I'm looking for. However, if you say incorrect things, like "This code doesn't work", you will be penalized).

Part 2 (10 points)

Describe (in English, not using code) how Unix's `malloc()` and `free()` work.

Part 3 (3 points)

```
typedef struct {
  char c1;
  char c2;
} twochar;
```

You are writing a program that calls `malloc(sizeof(twochar))` over a million times. Sporadically, these calls are interleaved with calls of `free(tc)`, where `tc` is a `twochar *`. Why is this going to present a problem for Unix's implementation of `malloc()` and `free()`?

Part 4 (7 points)

Suppose you decide to rewrite `malloc()` and `free()` so that your code from Part 3 works better. How are you going to design it? Include how both `malloc()` and `free()` will work. Explain why this fixes the problem. (Note, I'm not asking for code – just an explanation).

Question 4: 30 Points (four parts)

Suppose Dow Jones Industrial Inc. decides to support a stock market server. It works as follows. At Dow Jones, there is a person typing in current stock prices. The server serves a socket on port 200 on the machine **stocks.dowjones.com**. Whenever a client attaches to that socket, the client will be sent whatever stock prices the person is currently typing in. This continues until the client quits.

Obviously, the client should be able to attach to this server by calling `telnet`:

```
UNIX> telnet stocks.dowjones.com 200
...
```

You are a summer intern at Dow Jones Industrial, and your job is to write this server. Fortunately, because you have taken CS360, this will only take you a few hours, and you can spend the rest of the summer getting paid while you pretend to work.

At the heart of this server is the subroutine `broadcast_info`:

```
broadcast_info(Dlist clients)
{
  char line[100];
                                                /* Read in stock price  */
  while(fgets(stdin, line, 100) != NULL) {      /* from standard input */

    for (tmp = clients->flink; tmp != clients; tmp = tmp->flink) {
      write(tmp->val, line, strlen(line)) {     /* Write stock prices */
    }                                           /* to every client */
  }
}
```

Part 1: 7 points

If a client quits, a `SIGPIPE` signal is generated when the server tries to write to the client's socket. Modify the above subroutine to catch the `SIGPIPE` signal, and to deal with it correctly. You can modify the above procedure in whatever way you see fit. This includes adding global variables.

Part 2: 8 points

Having clients join is a trickier matter. Let us suppose that our server process has a pipe `p` open to its parent process, and is serving a socket `s`. The parent knows whenever a client wants to join (you'll see how in part 3). When a client wants to join, the parent sends a newline to the server over the pipe. The server can then do an `accept_connection(s)` and know that the `accept_connection(s)` will return with a connection.

Update `broadcast_info()` to work with this setup. By "work", I mean that it should be doing two things at once – waiting for lines to be entered on standard input (these lines are broadcast to clients), and waiting for new clients to be announced on the pipe.

It should now look as follows:

```
broadcast_info(int *p, int s, Dlist clients)
{
 ...
}
```

`Broadcast_info()` should still catch `SIGPIPE` and deal with it correctly.

Part 3: 10 points

Here's the rest of the code for the server (this includes the parent process). Explain how this code works. Specifically, how are things initialized? What happens when clients join (with `telnet`). What happens when clients quit?

```
main()
{
  int mainsock, secondsock, i, j;
  int p[2];
  Dlist clients;
  char c;

  pipe(p);

  if (fork() == 0) {
    secondsock = serve_socket("stocks.dowjones.com", 5000);
    clients = make_dl();
    close(p[1]);
    broadcast_info(p, secondsock, clients);
    exit(0);
  } else {
    mainsock = serve_socket("stocks.dowjones.com", 200);
    close(0);
    close(p[0]);
    while(1) {
      i = accept_connection(mainsock);
      if (fork() == 0) {
        close(p[1]);
        j = request_connection("stocks.dowjones.com", 5000);
        while(read(j, &c, 1) != 0) write(i, &c, 1);
        exit(0);
      } else {
        close(i);
        write(p[1], "\n", 1);
      }
    }
  }
}
```

Part 4: 5 points

This program has a problem with zombie processes. Explain this problem, and then fix it.