[2] 1a. Do a remote login using Secure Shell to aviary. Use the `ps` command to find out what processes you are running. Record the process ID (PID) and name of your shell process (`sh`, `bash`, `csh`, `tcsh`, etc.). Now, use the information about processes in the `/proc` directory to trace and record the hierarchy of processes, starting with your shell process and going all the way back to process 1 (the `init` process). Record the PID and program-name for each of the processes in this hierarchy. To find the necessary information, and assuming that the process-id of your shell happens to be 29496, then you should begin by exploring the contents of the file `/proc/29496/status` (using `less`).

In your answer to part a), the child process of `init` is a listener process that waits for incoming `ssh` requests and forks a new process to handle each one. However, there are *two* more processes between the listener and your shell. This is an example of the concept of *privilege separation*. CITI Technical Report 02-2, entitled *Preventing Privilege Escalation*, (http://static.usenix.org/legacy/events/sec03/tech/full_papers/provos_et_al/provos_et_al_html/) discusses privilege separation, and the application of this technique to OpenSSH. Read the paper and answer the following questions:

[1] 1b. In the introduction, the author states that it is "particularly important to protect against programming errors" in an application like OpenSSH. Give a brief (one sentence) explanation why this is so.

[2] 1c. What is the name of the principle the author states as the motivation for this technique? Find a definition of this principle (other than the one cited in the paper itself). Cite the definition and the source.

[1] 1d. What are the names the author uses for the privileged parent process and the unprivileged child process used to implement privilege separation?

[3] 1e. In the Security Analysis section, the author claims that 75% of any as-yet-undiscovered programming errors in OpenSSH would not result in an attacker gaining privileged access to a system. How did he arrive at this figure, and what assumption does he make regarding programming errors?

[3] 2a. If one is to run the program below in UNIX, what is the maximum number of processes that can be created (i.e. if *all* the resulting `fork()` calls are successful)? Draw the corresponding process family tree. Label the original process A, and the resulting child processes B, C, etc. **Hint**: a child process created by fork begins execution *after* the fork call that created it. If you label each executable statement (1, 2, 3…), it will make it easier to determine where in the program each child will begin execution.

```
main() {
      fork();
      fork();
      fork();
}
```

[3] 2b. Repeat for the program shown below.

```
main() {
      int i;
      i = fork();
      if(i)
        i = fork();
      if(!(i))
        i = fork();
}
```

[5] 3. Linux processes (or "tasks") commonly have much lower creation overhead (during fork) than conventional Unix processes. This is primarily because the fork system call uses a technique known as "copy on write". Use the web to learn what copy on write is and then write a brief paragraph describing it as it would be used during a fork system call. Why does copy on write save overhead?

4. For the programming question you will add functionality to your shell program from Lab 1 (in C language under Linux). You may use your own solution to Lab 1 or the sample solution as a starting point. Each function you add to your shell is worth part of your mark for this question; they are listed from easier to harder. Implement as much as you are able, but make sure that **all the functions are in the same program**. Use good program design methodology. A 'skeleton' program will be posted to help you get started.

[2] PATH environment: the shell will search for commands in all the directories listed in your PATH environment variable, so that you don't need to specify the complete path when typing a command. Consult the manual page for exec (`man 3 exec`) to find the correct function.

[6] Command-line arguments: your shell will accept an arbitrary number of arguments on the command-line. This will require C-style string processing to tokenize the input from the user (see `man strtok` or `man strtok_r`); we'll assume that only blanks and tabs will delimit tokens. The easiest way to work with an arbitrary number of arguments is to store them in an array of strings; again, you should find the version of the exec function that supports this. You can set some sort of reasonable upper limit on the number of arguments (say, 20).

[10] The `cd` command: this command changes the current working directory of the shell; as such it can't be performed by a child process (since a child process can only change its own working directory, not its parent's working directory). The `chdir` function (`man chdir`) is used to change the current directory. Your version of `cd` should work similarly to a "real" shell's version. For example, if no arguments are supplied, it should change to the "home" directory (in this case, the directory containing your shell program); if one argument is supplied, it should attempt to change to that directory, and if more than one argument is supplied, an error message should be displayed.

[12] Wildcard expansion (globbing): your shell will accept arguments with wildcard characters and generate all paths that match the specified patterns (e.g. `ls *.c`). The `glob` function (`man 3 glob`; see `man 7 glob` for a description of globbing)

implements the matching; you need to figure out how to get this to work within your shell.

A script for testing your shell will be made available shortly. A 'skeleton' program will also be made available.

## Total: 50 marks

## Handing in

You will hand in your answers to the written questions and programming questions using the D2L Dropbox for Assignment 1.

a) Written questions: Your answers to the first 3 questions should be prepared electronically. Acceptable formats are: Word document, Adobe reader document (.pdf) or text file (.txt).
b) Programming question: Hand in a single C source file. You may also hand in a README.txt file explaining any unusual aspects of your program, but it is not required.

While your final programs must run on the Linux machines identified on the course homepage, you are free to develop your code on other platforms if you like. You alone, however, are responsible for dealing with any differences in the compilers, tools, etc. that you may encounter. Please remember to leave plenty of time to port your code to Linux. Normally this takes less than 30 minutes but sometimes it may take several hours if your code is still a little buggy. (e.g. The C compiler on Solaris sets all un-initialized memory to zeroes. Hence, bad pointers are treated as NULLs and thus, sometimes incorrect code works there but not with other compilers and operating systems.)