Problems 1-8: Short answer questions. Answer each question clearly, precisely, and refer to specific system calls when appropriate. Write your answer in the space provided.

1. Why is it considered a bad idea to write programs that make more system calls than are necessary?
2. If you know the name of a file, how can you find the name of the user that owns that file?
3. List four system calls that return file descriptors. For each system call, explain what the file descriptor refers to.
4. What is the relationship between the systems calls *exit()* and *wait()*?
5. What is the purpose of the `fork()` system call? What value does it return, and what does that value indicate?
6. Why is the system call that deletes a file called `unlink()` rather than `delete()`?
7. What is the meaning of `onlcr` mode, and why is it useful?
8. What is a signal? Give an example and explain how it is useful.

Problems 9-13: Compare and contrast. Each of these problems mentions two related concepts, system calls, or operations. For each pair, explain briefly and clearly (a) what they have in common, (b) when you would use the first item, and (c) when you would use the second item.

9. `popen()` vs `fopen()`
10. `program` vs `process`
11. `accept()` vs `connect()`
12. `blocking input` vs `non-blocking input`
13. `symbolic link` vs `hard link`

Part Three

'*Getting System Status*' Some system calls request actions, and some request information. For example, the time() system call requests information, while the chdir system() system call requests an action. Each of the following system calls does both: initiates an action and requests information. For each of these system calls, explain what information is returned by the call, and describe how the returned value is useful.

14. read(), 15. lseek(), 16. signal(), 17. alarm()

Part Four

18. An enhancement to your small shell - the `trap` built-in

The shell you wrote for homework was supposed to ignore interrupts but allow its children processes to catch them. The 'real' shell is more flexible. It has a *trap* built-in command that allows the user to set the shell to respond to signals. Here is what the manual says about trap:

```
trap [arg] [n] ..
   Arg is a command to be read and executed
   when the shell receives signal(s) n .  If
   arg is absent then all trap(s) n are re-
   set to their original values.  If arg is
   the null string then this signal is ig-
   nored by the shell and by invoked com-
   mands.  If n is 0 then the command arg is
   executed on exit from the shell, other-
   wise upon receipt of signal n as numbered
   in signal (2).  Trap with no arguments
   prints a list of commands associated with
   each signal number.
```

Describe how you would modify your shell to handle the *trap* built-in command. You do not need to include code or low-level details. You must cover at least three topics: 1) any data structures you will add to implement this built-in, (2) how the code for the trap built-in will operate, (3) how setting traps affects execution of commands.

19. *Biff Lives!* Mail Notification Email is now a standard, relied-upon means of communication. Some people who depend on email want to be notified when new messages appear, just as people who depend on the telephone expect to hear the telephone ring when someone calls.

This is the question: devise a program that notifies a user when new mail has arrived. In this problem, you will explore three solutions to this problem.

On most Unix machines, mail is delivered by appending the message to a file called `/usr/spool/mail/logname` where `logname` is the logname of the user. Some systems use `/usr/mail/logname` and some use other directories. The full pathname of the user's mailfile is usually stored in an environment variable called $MAIL.

Method I: The Shell Watches [5 points] Many Unix shells are programmed to check for the arrival of new mail. The shells usually print out a message like "You have new mail" just before they print out a new prompt. The shell checks for new mail every *n* seconds, where *n* is a variable set by the user.

a) Explain what additions you would need to make to your shell to implement the above-described feature. What system calls would you need to use?
b) Why is this not an ideal solution? That is, why might you not hear about new mail if you are using this method?

Method II: A Dedicated Process Watches [5 points] One solution to the problem mentioned in (b) is to run a separate background process, running in the background just as your `watch` program did.

c) Write a simple stand-alone program (in C or C++) that can be run in the background and will notify a user when new mail arrives. The program should take as a command-line argument the number of seconds between checks for new mail.

d) What are the drawbacks of this method?

Method III: A Single Program Watches for Everyone [7 points] The Unix program called `biff` (named after a dog said to bark when the mailman arrived) uses a somewhat bizarre method. When you run the biff command, it sets the 'user execute bit' in the special file that represents your terminal. That is, if you are logged in at `/dev/ttyq3`, then the protection mode for `/dev/ttyq3` will be **-rwx-w--w-** . There is no meaning to the idea of a terminal being executable; therefore the bit has no practical use for file access in this case.

A single background program watches for incoming mail for all users who had run the `biff` command. The `biff` watcher program checks for new mail at an interval set when it starts. By examining all terminal devices in the `/dev` directory, the program can determine which users have new mail and can notify the user.

e) Write an outline for the biff watcher program. Mention in your description which system calls you need to make to implement this program. Make your algorithm clear. Be sure you explain how the program knows if new mail has arrived and how the program gets the notification to the user.