csci *e215*		
	final *exam*	
		may 11 *1992* *usa*

Your Name Here: _____

Instructions: You have two hours for this exam. This exam is 'open notes', but not 'open book'. Please write your answers on the pages in this exam booklet. No scrap paper or additional sheets will be accepted. Watch your time, be concise, think before you write, look before you leap, listen before you accept, and *always* check the return value of a system call. Good luck.

prob	points	got	section
1	6		
2	6		
3	6		
4	6		
5	6		
6	6		
7	6		
8	6		
9	6		
10	6		
11	6		
12	6		
13	6		
1	4		
2	4		
3	4		
4	4		
b	6		

Problems 1-7: For each system call given, briefly describe its purpose, the arguments it takes, and the value(s) it returns.

name	**purpose**	**arguments**	**return value(s)**
open()			
chdir()			
dup()			
ioctl()			
socket()			
exit()			
alarm()			

Problems 8-14: For each question, write a short, accurate answer in the space provided.

8. Briefly describe *buffered output*? What is it, how does one do it, and why is it done?

9. What is *errno*? What is it for? How is it used?

10. What is the difference between *cooked* input and *raw* input? Why would you use one rather than the other? Does this difference apply to non-terminal files?

11. When a Unix program wants to run another program as a sub-process, it calls fork() then exec(). Why not have one system call to *run_sub_program()*? Name two useful activities a process can perform between the fork() and the exec().

12. The remote execution server demonstrated in class redirects standard output to a socket before executing the shell. Explain what happens to messages the shell sends to standard error. How can these error messages be sent to the socket along with the standard output?

```
case 0  :
   len = read(fd,cmdbuf,512);  /* get the command */
   cmdbuf[len] = '\0';         /* terminate it    */
   close(1);                   /* close stdout    */
   dup(fd);                    /* attach fd to 1  */
   close(fd);                  /* tidy up         */
   execlp("sh", "sh", "-c", cmdbuf);
   oops( "execlp" );
```

13. File names can represent regular files, directories, devices, sockets, symbolic links, and fifos. What system call, include file, and C operators do you use to determine what sort of thing a file name represents?

14. This problem explores the question of how the shell handles signals.

First, the overview. A shell can run interactively or from a script. Operating interactively, it reads a command, executes it, and returns to the user for the next command. Reading from a script, it repeatedly takes commands from a file, executing them in sequence.

When the user of the shell presses the Interrupt key, often Ctrl-C, the shell is sent a signal, SIGINT. The user may press other keys to generate other signals, or may use the *kill* command to send a signal to the shell.

The shell runs programs and waits until they finish. A program run by the shell may receive an signal. For example, the user may run *a.out*, and press the interrupt key. For another example, the user may run *a.out*, and that program may attempt to access memory outside its region and be sent a SIGSEGV.

We shall limit our attention to SIGINT and consider four combinations:

case	*shell mode*	*state*
1	interactive	shell running
2	script	shell running
3	interactive	program running, shell waiting
4	script	program running, shell waiting

Part a: For each of these four cases, write short, clear answers to each question.

Case 1: An interactive shell is waiting for input. It catches SIGINTs, flushes input, and issues a new prompt (this is what the Bourne Shell does). Why is this preferable to exiting upon receipt of an interrupt?

Case 2: A shell is running from a script. It is processing a *read* command, digesting comments, parsing an if command, or some other shell level command. The shell receives SIGINT. In this case, the Bourne Shell exits. Why is this preferable to catching the interrupt and simply moving on to the next command?

Case 3: An interactive shell is running a program and waiting for it to finish. How should this interactive shell handle SIGINT ? Why?

Case 4: A shell is running from a script. It fork()s and exec()s the program *a.out*. While a.out is running, the user presses Ctrl-C to stop a.out. A.out stops. What should the shell do when it detects that a.out has been interrupted? Why? How does it know that a.out was stopped by signal and did not just exit() normally?

Write your answers to the questions in cases 1-4 below and on the next page. There is *one more part* to this question. It is on page 6.

(More on the Back)

part b: The shell operates with certain defaults. You have discussed these defaults in part a. A user may specify other actions by using the *trap* built-in command. 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 reset to their original values. If arg is the null string then this signal is ignored by the shell and by invoked commands. If n is 0 then the command arg is executed on exit from the shell, otherwise 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.