csc/cpe 357 Midterm

Archive

Name: ______________________

Section: ______________________

Rules:

- Do all your own work. Nothing says your neighbor has any better idea what the answer is.
- You *may not* use any other materials.
- This exam is copyright ©2018 Phillip Nico. Unauthorized redistribution is prohibited.

Suggestions(mostly the obvious):

- When in doubt, state any assumptions you make in solving a problem. **If you think there is a misprint, ask me.**
- **Read the questions carefully.** Be sure to answer all parts.
- Identify your answers *clearly.*
- Watch the time/point tradeoff: Nevermind. There are no points.
- Problems are not necessarily in order of difficulty. They are in the order in which they fit.
- Be sure you have all pages. Pages other than this one are numbered "*n* of 21".

Encouragement:

- Good Luck!

Problem	Possible	Score
1	●	
2	●	
3	●	
4	●	
5	●	
6	●	
7	●	
8	●	
9	●	
10	●	
11	●	
12	●	
13	●	
14	●	
15	●	
16	●	
17	●	
18	●	
19	●	
20	●	

Problem	Possible	Score
21	●	
22	●	
23	●	
24	●	
25	●	
26	●	
27	●	
28	●	
29	●	
30	●	
31	●	
32	●	
33	●	
34	●	
35	●	
36	●	
37	●	
38	●	
39	●	
40	●	

Problem	Possible	Score
41	●	
42	●	
43	●	
44	●	
45	●	
46	●	
47	●	
48	●	
49	●	
50	●	
51	●	
52	●	
53	●	
54	●	
55	●	
56	●	
57	●	
58	●	
59	●	
60	●	

Problem	Possible	Score
61	●	
62	●	
63	●	
64	●	
65	●	
66	●	
67	●	
68	●	
69	●	
70	●	
71	●	
72	●	
73	●	
74	●	
75	●	
76	●	
77	●	
78	●	

Total:	**0**	

0.1 General knowledge

1. () The UNIX system utility `cp` will refuse to copy a file if the source and destination files are the same. (How does it know?) No, really, how does it know?

   Write a C function called `same_file` that takes two pathnames and returns true (nonzero) if both paths specify the same file and false otherwise (if either one doesn't exist (or is inaccessible), or they are not the same).

   ```
   int same_file(const char *src, const char *dst) {
   ```
   Optional extra space for problem 1.
   ```
   }
   ```

2. () Shells like tcsh and bash implement home directory expansion where they replace the string "`~user`" with the path of *user*'s home directory. So, for example, "cd ~pn-cs357" would change directories to pn-cs357's home. Write a function, `cdTilde()` that takes a user's name and changes the current working directory to that user's home directory. `cdTilde()` returns 0 on success, and −1 on failure.

3. () Consider the following two programs:

Program A

```
#include <unistd.h>
#include <stdio.h>

/* Macro SIZE is defined at compile-time */

int main(int argc, char *argv[]){
  int n;
  char buffer[SIZE];
  while((n=read(STDIN_FILENO,buffer,SIZE))>0) {
    write(STDOUT_FILENO,buffer,n);
  }
  return 0;
}
```

Program B

```
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]){
  int c;

  while( EOF != (c=getchar()) )
    putchar(c);

  return 0;
}
```

Program A was compiled once with `SIZE` defined to be 1 and a second time with `SIZE` defined to be 8192. Program B was compiled as is. The resulting executables, in random order, were named `larry`, `curly`, and `moe`.

Now, consider the following three experiments run with these three programs:

Exp.	**Command**	**Runtime (min:sec)**

—	`% ls -l BigFile`
	`-rwx------ 1 pnico pnico 42114758 Oct 2 2002 BigFile`
		Who cares?

1	`% larry < BigFile > /dev/null`	0:00.04
2	`% curly < BigFile > /dev/null`	0:01.88
3	`% moe < BigFile > /dev/null`	2:06.90

Given this information, match each of the program configurations below to the proper executable *and explain why.* (The reason is worth more than the identification.)

Configuration	**Executable**	**Explaination**
Program A, `SIZE` = 1	☐ larry ☐ curly ☐ moe	
Program A, `SIZE` = 8192	☐ larry ☐ curly ☐ moe	
Program B	☐ larry ☐ curly ☐ moe	

4. () If you forget the password to your hornet account and the sysadmin refuses to tell you what it was, is he just being lazy? Why or why not?

5. () What is the fundamental difference between a system call and a library function?

6. () Is it possible for an ordinary user (not root) to create a file owned by some other user id? How about another group id? Why or why not?

7. () About those links . . .

   (a) (5pts.) A hard link can only be created to a file on the same disk partition, while a symbolic link can link any file anywhere. Why is this the case?

   (b) (5pts.) If symlinks are so much more flexible than hard links, what is the use of having hard links at all?

8. () Given a UNIX filesystem composed of several smaller filesystems (different disk partitions, e.g.), why would you expect using `mv` to move a file to another directory on the same partition to be faster than moving it to a directory on another partition?

9. () It is not possible to make a hard link on one disk partition (filesystem) to a file that resides on another partition. Given what you know about links (and setting aside the question of whether this would be a good idea in the first place) explain why such a thing must be impossible.

10. () It is possible (trust me) for there to be a file that a user can move within a filesystem, but not across filesystems. Assuming the user has write permission for both the source and destination directories, how (under what circumstances) could this be? **Explain.**

11. () The `utime(2)` system call allows one to set the last access and last modification times on a file to any representable time (very useful for sneaking in late homeworks), but it cannot backdate the last changed time (bummer!). Why not?

12. () Is it possible for a user to use `open(2)` or `creat(2)` to create a file s/he cannot delete? Explain how this is possible or why it is not possible.

13. () In a Unix system, is it possible for there to be a file that the owner of the file cannot remove? Why or why not?

14. () A programmer dissatisfied with the behavior of a C library function can redefine it without limiting the capabilities of the program. (That is, there is nothing the program could have done before the redefinition that it could not do afterwards.) A system call, however cannot be replaced without limiting the program. Why is this?

15. () What is the fundamental difference between a system call and a library function?

16. () Anyone can make a hard link to a file, but only root is permitted to make hard links to a directory. What is the danger in creating a hard link to a directory?

17. () Some versions of the `finger(1)` command output "New mail received..." and "Unread since..." with the corresponding dates and times. Given that *user*'s mail is stored in `/var/mail/`*user*, how can the program determine these times and dates?

18. () The UNIX command `df(1)` reports the amount of free space available on a particular filesystem. Now, consider the following typescript:

```
% df .
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda1                46636      9034     35194  21% /extra
% fortune > myfile
myfile: No space left on device.
%
```

`df(1)` clearly shows 35Mb available on the disk. This isn't a lot of space, but surely it's enough for a fortune. Explain what has happened here.

19. () The UNIX command `df(1)` reports the amount of free space available on a particular filesystem. Now, consider the following typescript:

```
% df .
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda1                46636      8856     35372  21% /extra
% ls -l
[...]
-rw-------    1 pnico    pnico    52428751 Feb 19 20:26 testfile
[...]
%
```

`df(1)` clearly shows the disk capacity to be 45.5MB, yet `testfile` occupies 50MB of disk space according to `ls(1)`. Has the the storage management problem finally been solved by writing the bits really small? Explain what is really happening here.

20. () The system call `getpwent(2)` allows a program to sequentially read through all the password information for all the users known to the system. It returns a pointer to a `struct passwd`. One of the fields of this struct is:

```
char *pw_passwd; /* user password */
```

Is this safe? **Explain.**

21. () What is the difference between a *program* and a *process?*

22. () The command `ls -lc` shows the change time (ctime) for a file. Given the following sequence of commands, why does removing `file` cause `link`'s ctime to change?

```
% fortune > file
% ln file link
% ls -lc file link
-rw-------.  2 pnico pnico 44 Nov 3 21:55 file
-rw-------.  2 pnico pnico 44 Nov 3 21:55 link
...some time later ...
% date
Tue Nov 3 21:56:59 PST 2009
% rm file
% ls -lc link
-rw-------.  1 pnico pnico 44 Nov 3 21:57 link
%
```

23. () If process owned by root receives a `SEGV` while its current working directory is a user's home directory, the corefile will wind up in that directory:

```
% ls -l ~
-rw------- 1 root root 22945792 Mar 13 2005 core.13411
...
```

Explain why—even with the ownership and permissions as shown above—the user will always be able to remove this inconvenient corefile.

0.2 Permissions

24. () A user with a umask of `0534` attempts to create a file with the call:

```
open("examfile", O_RDWR | O_CREAT, S_RWXU | S_IRGRP | S_IXGRP | S_IXOTH);
```

(a) Assuming that `examfile` does not exist, what will permissions of the created file be? (show your work)

25. () A user with a umask of `0415` attempts to create a file with the call:

```
open("examfile", O_RDONLY | O_APPEND | O_CREAT, S_IXUSR | S_IRGRP | S_IWGRP | S_IRWXO );
```

Assuming that `examfile` does not exist, what will permissions of the created file be? (show your work)

26. () If a user with a umask of `026` attempts to create a file with the call:

```
open("examfile", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU | S_IRWXG | S_IRWXO);
```

Given that `examfile` does not exist, what will permissions of the created file be?

27. () If a user with a umask of `0253` attempts to create a file with the call:

```
open("examfile", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU | S_IRWXG | S_IRWXO);
```

Given that `examfile` does not exist, what will permissions of the created file be?

28. () If a user with a umask of `0257` attempts to create a file with the call:

```
open("examfile", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
```

Given that `examfile` does not exist, what will permissions of the created file be?

29. () A user with a umask of `0126` attempts to create a file with the call:

```
open("examfile", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
```

(a) Assuming that `examfile` does not exist, what will permissions of the created file be? (show your work)

(b) Assuming the file *does* exist and the call to `open()` succeeds, what can you say about the permissions of the file after the call?

30. () If a user with a umask of `0253` attempts to create a file with the call:

```
open("examfile", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU | S_IRWXG | S_IRWXO);
```

Given that `examfile` does not exist, what will permissions of the created file be?

31. () A process with a umask of `0213` attempts to create a file with the call:

```
open("examfile", O_RDONLY | O_APPEND | O_CREAT, S_IWUSR | S_IXUSR | S_IRGRP | S_IWGRP | S_IROTH );
```

Assuming that `examfile` does not exist, what will permissions of the created file be? (show your work)

32. () If a user with a umask of `0237` attempts to create a file with the call:

```
open("examfile", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
```

Given that `examfile` does not exist, what will permissions of the created file be?

33. () If a user with a umask of `0237` attempts to create a file with the call:

```
open("examfile", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IXOTH);
```

Given that `examfile` does not exist, what will permissions of the created file be?

34. () Assuming the that the file in problem 31 *does* exist and the call to `open()` fails, what can you say about the permissions of the file after the call?

35. () Assume a process with effective user id 4 and effective group id 7 tries to execute a file with user id 4, group id 9, and permissions `rw-r-x--x`. What will happen (and why)?

36. () Assume a process with real user id 12, effective user id 4, real group id 23, and effective group id 7 tries to execute a file with user id 12, group id 7, and permissions `rw-r-xr--`. What will happen (and why)?

37. () Assume a process with real user id 12, effective user id 10, real group id 23, and effective group id 7 tries to read a file with user id 12, group id 10, and permissions `rwxr-xr--`. What will happen (and why)?

38. () Write a C function that takes a filename as its parameter and returns true if the given file exists, is an ordinary file, and somebody has execute permission for it, and false otherwise.

```
int is_program(char *fname) {
}
```

39. () Write a function, `AplusX()` that takes a path name as a parameter and grants execute permission to all (user, group, and other) if anybody has execute permission for the named file. On success, `AplusX()` should return 0. On failure it should return −1 and leave `errno` alone. (So whoever called `AplusX()` can react appropriately to whatever failed.)

40. () Write a function, `AminusW()` that takes a path name as a parameter and upon successful completion, guarantees that no one (user, group, or other) has write permission for the given file. On success, `AminusW()` should return 0. On failure it should return −1 and leave `errno` alone. (So whoever called `AminusW()` can react appropriately to whatever failed.) `AminusW()` should not fail unless its goal is actually unachievable.

Optional extra space for problem 40.

}

41. () Write a C function that takes a pathname (relative or absolute) and returns true the file exists and the caller has permission to read it. Do not use `access(2)`.

    Think before writing this one.

    ```
    int canIread(const char *path) {
    ```
    Optional extra space for problem 41.

    ```
    }
    ```

42. () Write a C function called `share_read` that takes a filename as its parameter and attempts to make it readable to all if anyone has read permission. It should return true if the given file exists and (now) has proper permissions (either nobody had read permission, or everyone has read permission). It should return false otherwise.

    ```
    int share_read(char *fname) {
    ```
    Optional extra space for problem 42.

    ```
    }
    ```

43. () Write a C function called `safe_from_me()` that takes a path as a parameter and returns true if given path represents an ordinary file whose contents the calling program is unable to alter and false otherwise (path represents something other than a file, the calling program can mutate the file, the file doesn't exist, etc.). Write robust code.

    ```
    int safe_from_me(const char *path) {
    ```
    Optional extra space for problem 43.

    ```
    }
    ```

0.3 Bit manipulation

44. () Write a C function called `make_byte()` that takes array of eight integers as its parameters and returns the byte formed by setting each bit in the byte according to the boolean value of each array element. Bits should be filled in from the high-order end (That is, `make_byte({1,1,0,0,0,1,0,1})` is `0xc5`).

    You may assume you are given a valid array pointer.

    ```
    unsigned char make_byte(const int bits[]) {
    ```
    Optional extra space for problem 44.

    ```
    }
    ```

45. () Write a C function called `makeint()` that takes a linked list of bit values ($\{0, 1\}$) in the structure defined below, strips off any leading zeros, and then builds an int out of the remaining values. Returns the resulting integer on success or -1 if the number cannot be represented (due to overflow). You may assume sizeof(int) is 4.

    ```
    typedef struct bit *bitlist;
    struct bit {
      int bit;
      bitlist next;
    };
    ```

    For example, `makeint(`$\rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 1$`)` would return 11, while `makeint(NULL)` would return 0 (no bits set).

    ```
    int makeint(bitlist bits) {
    ```

Optional extra space for problem 45.

}

46. () Write a C function called `sprint_bits()` that takes a char pointer and unsigned integer as a parameters and writes the binary representation of the integer into the string pointed to by the char *. `sprint_bits()` returns a pointer to head of the resulting string. E.g.:

```
unsigned int num = 0xDEADC0DE;
char bitstr[33];
sprint_bits(bitstr, num);
printf("%u is %s.\n", num, bitstr);
```

will print:

```
3735929054 is 11011110101011011100000011011110.
```

`char * sprint_bits(char *buffer, unsigned int num) {`

Grading Notes

Do:

2	generate mask (or test sign or remainder)	
5	mask bits	
	2	mask
	1	correct bit
	2	increment
2	terminate string	
1	return string	

Optional extra space for problem 46.

}

47. () Write a C function called `bstr2int()` that takes a valid C string (possibly NULL) consisting exclusively of the digits "0" and "1", strips off any leading zeros, and then builds an int out of the remaining values. Returns the resulting integer on success or −1 if the number cannot be represented (due to overflow). Recall that the size of an int on the current platform can be determined through `sizeof(int)`.

   For example, `bstr2int("00001100")` would return 12, while `bstr2int(NULL)` would return 0 (no bits set). Write robust code.

   `int bstr2int(const char *s) {`

Optional extra space for problem 47.

}

48. () Write a function `parity()` that takes an integer and returns 1 if the integer has an odd number of bits set, 0 otherwise. Also, if it matters, note that the value being passed is signed, and you may assume sizeof(int) is 4. Write robust code.

   `int parity(int val) {`

   }

49. () A common way of detecting transmission errors in data is the use of *parity bits.* One bit, usually the most significant, is either set or cleared to ensure that the resulting byte has even an even or odd number of bits set before being sent. If this is not the case on receipt, the resulting byte is assumed to have been corrupted in transit.

   Write a function `set_even_parity()` that takes a byte as a parameter and returns the same byte with the most significant bit either set or cleared to create even parity. Write robust code.

   `char set_even_parity(char b) {`

Optional extra space for problem 48.

}

0.4 Follow the bouncing ball

50. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Clearly mark the current end of the file (as with ⊗ below).

    The initial contents of the file, "`tf`", are:

`tf:`	B	e	g	i	n	⊗

```
int main(int argc, char *argv[]){
  char *filename = "tf";
  char *newfile  = "other";
  int one, two, three;

  one = open(filename,O_RDWR);
  write(one,"One",3);
```

`tf:`																	

```
  two   = dup(one);
  link(filename,newfile);
  three = open(newfile,O_WRONLY);
  write(two,"two",3);
  write(three,"three",5);
```

`tf:`																	

```
  lseek(one,   0,  SEEK_SET);
  lseek(three, 5,  SEEK_END);
  write(one,"String",3);
```

`tf:`																	

```
  write(three,"done!",5);
```

`tf:`																	

```
  close(one);
  close(two);
  close(three);

  return 0;
}
```

51. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Clearly mark the current end of the file (as with ⊗ below).

    The initial contents of the file, "`tf`", are:

`tf:`	M	i	d	t	e	r	m	⊗

```
  char *filename = "tf";
  char *backup  = "bob";
  int one, two, three;

  one = open(filename,O_RDWR);
  write(one,"grue?",5);
  write(one,"xyzzy!",6);          /* for the old–timers */
```

tf:																	

```
  symlink(filename,backup);
  two   = open(backup,O_WRONLY);
  three = dup(two);
  write(two,"Wayne",3);
  write(three," SNL",3);
```

tf:																	

```
  lseek(one,    12,  SEEK_SET);
  lseek(three, −5,  SEEK_END);
  write(three,"ears?",2);
```

tf:																	

```
  write(three,"kernel",1);
  write(one,"42",2);
```

tf:																	

```
  close(one);
  close(two);
  close(three);

  return 0;
}
```

52. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Even though UNIX has no end-of-file mark, you should clearly mark the current end of the file (as with ⊗ below).

    The initial contents of the file, "tf", are:

tf:	c	p	e	2	5	0	?	⊗

```
  char *filename = "tf";
  int one, two, three;

  one = open(filename,O_RDWR);
  write(one,"cpe/csc",7);
  write(one,"X317",4);
```

tf:																	

```
  two   = open(filename,O_WRONLY);
  three = dup(two);
  write(three,"new",3);
  write(two,"Course",6);
```

tf:																	

```
  lseek(one,    13,  SEEK_SET);
  lseek(three, −6,  SEEK_END);
  write(three,"number after",6);
```

tf:																	

```
  lseek(two,6,SEEK_SET);
  write(three,"summer",3);
  write(one,"357?",3);
```

tf:																	

```
  close(one);
  close(two);
  close(three);

  return 0;
}
```

53. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Even though UNIX has no end-of-file mark, you should clearly mark the current end of the file (as with ⊗ below). Be careful.

    The initial contents of the file, "tf", are:

tf:	P	r	e	s	i	d	e	n	t	?	⊗

```
  char *filename = "tf";
  int one, two, three;

  one = open(filename,O_RDWR);
  write(one,"Vote",4);
  write(one,"Early",5);
```

tf:																	

```
  two   = dup(one);
  three = open(filename,O_WRONLY);
  write(two,"And",1);
  write(three,"Often",5);
```

tf:																	

```
  lseek(one,    12,  SEEK_SET);
  lseek(three, −10,  SEEK_END);
  write(three,"Now wait",2);
```

tf:																	

```
  lseek(three,12,SEEK_SET);
  lseek(two,3,SEEK_SET);
  write(two,"Real Deal?",4);
  write(three,"Gemini Twins?",3);
```

tf:																	

```
  close(one);
  close(two);
  close(three);

  return 0;
}
```

54. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Even though Unix has no end-of-file mark, you should clearly mark the current end of the file (as with ⊗ below). **Be careful.**

    The initial contents of the file, "tf", are:

tf:	P	r	i	m	a	r	y	?	⊗

```
char *filename="tf";
int one, two, three;

one = open(filename,O_RDWR);
write(one,"Already",7);
write(one,"Done",4);
```

tf:																	

```
two   = open(filename,O_WRONLY | O_TRUNC);
three = dup(one);
write(two,"November's a",2);
write(three,"Fairly",4);
```

tf:																	

```
write(two,"too",1);
lseek(one,    12,  SEEK_SET);
lseek(two,     8,  SEEK_CUR);
lseek(three, −10,  SEEK_END);
write(one,"long wait",4);
write(two,"now",4);
write(three,"  ",2);  /* that's two spaces */
```

tf:																	

```
lseek(three,1,SEEK_SET);
lseek(two,11,SEEK_SET);
write(one,"extra",3);
write(two,"years?",4);
```

tf:																	

```
close(one);
close(two);
close(three);
```

55. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Even though UNIX has no end-of-file mark, you should clearly mark the current end of the file (as with ⊗ below). **Be careful.**

The initial contents of the file, "tf", are:

tf:	H	e	r	e	,	P	i	g	g	y	⊗

```
char *filename="tf";
int one, two, three;

one = open(filename,O_RDWR);
write(one,"Oink",4);
write(one,"Eek",3);
```

tf:																	

```
two   = dup(one);
three = open(filename,O_RDWR | O_CREAT, S_IRWXU);
write(two,"Big?",4);
write(three,"Bad!",4);
```

tf:																	

```
write(three,"Pin",3);
```

```
lseek(one, 5, SEEK_SET);
lseek(two,−5, SEEK_CUR);
write(one,"Twine",6);
lseek(three, 2, SEEK_END);
write(two,"glue",4);
write(three,"?",1);
```

tf:																	

```
lseek(two,0,SEEK_SET);
lseek(three,6,SEEK_SET);
write(one,"slip",1);
write(three,"flip?",1);
```

tf:																	

```
close(one);
close(two);
close(three);
```

56. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Even though Unix has no end-of-file mark, you should clearly mark the current end of the file (as with ⊗ below).

    The initial contents of the file, "tf", are:

tf:	W	i	n	d	o	w	s	?	⊗

```
char *filename = "tf";
int one, two, three;

one = open(filename,O_RDWR);
write(one,"For",3);
write(one,"museum",6);
```

tf:																	

```
two   = open(filename,O_WRONLY);
three = dup(one);
write(three,"early",2);
write(two,"Mustangs",4);
```

tf:																	

```
lseek(two,    14,  SEEK_SET);
lseek(three, −4,  SEEK_END);
write(three,"ideal",5);
```

tf:																	

```
  lseek(one,7,SEEK_SET);
  write(three,"AREA",2);
  write(two,"OS",2);
```

tf:																	

```
  close(one);
  close(two);
  close(three);

  return 0;
}
```

57. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Even though UNIX has no end-of-file mark, you should clearly mark the current end of the file (as with ⊗ below). **Be careful.**

    The initial contents of the file, "tf", are:

tf:	T	h	i	s		o	n	e	?	⊗

```
char *filename="tf";
int one, two, three;

one = open(filename,O_RDWR);
write(one,"Really?",4);
```

tf:																	

```
two   = open(filename,O_RDWR | O_CREAT, S_IRWXU);
three = dup(two);
write(two,"It's",4);
write(three,"Traditional!",9);
```

tf:																	

```
lseek(one,−3, SEEK_END);
write(one,"Where",3);
lseek(two, 0, SEEK_SET);
write(three,"are",4);
write(two,"we?",2);
lseek(one,1, SEEK_CUR);
```

tf:																	

```
lseek(two,0,SEEK_SET);
lseek(three,6,SEEK_SET);
write(one,"Not",2);
write(two,"Clever.",3);
write(three,"Today.",2);
```

tf:																	

```
close(one);
close(two);
close(three);
```

58. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Even though UNIX has no end-of-file mark, you should clearly mark the current end of the file (as with ⊗ below).

The initial contents of the file, "tf", are:

tf:	M	i	d	t	e	r	m	?	⊗

```
int one, two, three;
char *filename="tf";

one = open(filename,O_RDWR);
write(one,"What again?",3);
```

tf:																				

```
two   = open(filename,O_WRONLY|O_TRUNC);
three = dup(one);
write(three,"Didn't",3);
write(two,"We",3);
```

tf:																				

```
lseek(one,    13,  SEEK_SET);
lseek(two,     7,  SEEK_SET);
write(two,"just do",5);
lseek(three,  −1,  SEEK_CUR);
write(three,"this?",2);
lseek(two,5,SEEK_CUR);
```

tf:																				

```
write(two,"hayfever?",2);
write(one,"atchoo?",2);
```

tf:																				

```
close(one);
close(two);
close(three);
```

0.5 Timers

59. () Write a function `ualarm()` that uses the interval timer to deliver a single SIGALRM a given number of microseconds from now.

    If successful, returns the number of microseconds remaining until a currently pending alarm (0, if none). On failure, returns −1 and leaves errno alone to indicate the error. In no case does `ualarm()` print anything. Write robust code.

```
long ualarm(long usec) {
```

Optional extra space for problem 49.

```
}
```

60. () Given the following structure and type definitions for a node in a binary tree:

```
typedef struct node_st node;
struct node_st {
    whocares data;   /* the data */
    node *left;      /* left child or NULL */
    node *right;     /* right child or NULL */
};
```

Write a C function, `count_nodes()`, that takes as an argument the root of one of these trees, possibly empty, and returns the number of nodes in the tree. Write robust code.

```
int count_nodes(node *root) {
}
```

0.6 probably obsolete

61. () **Multiple Choice** (By request) What is the output of the following program:

```
#include <stdio.h>

int f(int *ip) {
  *ip = *ip + 1;
  return *ip;
}

#define g(x) ( x − x )

int main(int argc, char *argv[]) {
  int res;
  int i = 1;
  res = g(f(&i));
  printf("%d\n",res);
}
```

(1pt.) Output?

(a) −1

(b) 0

(c) 1

(d) Huh? (indeterminate)

- (4pts.) Why?

62. () **True/False** (By request) Given the following structure definition:

```
struct thing {
  ThingOne Cat;
  ThingTwo Hat;
};
```

and the statement,

`sizeof(struct thing)` is always the same as `sizeof(ThingOne) + sizeof(ThingTwo)`

- (1) This statement is (circle one):

- (4) Why?

63. () A novice programmer wrote the following fragment of code for a new game destined to be a mega-hit. At the next code review, the project manager looked at the program, announced, "I will not be responsible for this kind of work!" and quit on the spot (in today's economy, no less!)

```
[...]
#define MAX_ANIMAL 1024
void update_tree(NODE node) {
  char animal[MAX_ANIMAL];
  printf("What is it?  ");
  scanf("%s",animal);
  insert_node(node, animal);
  [...]
}
```

What is wrong with the code fragment? (Hint: It compiles fine, and you don't have to make any assumptions about any other code.)

0.7 OTHER

64. () Write a C function that takes a filename as its parameter and does the following: If the given file does not exist, the function should create it, give read/write access to the owner, give only write access to everyone else, and return an open file descriptor (write only) to the file. If the file already exists, the function should leave it alone and return −1.

```
int make_new_log(char *fname) {
}
```

65. () The library function `getcwd(char *buf, size_t size)` copies the absolute pathname of the current working directory into the array pointed to by `buf`. If the current directory would require a buffer of more than `size` characters, `NULL` is returned and `errno` is set to `ERANGE`. If `getcwd()` is unable to determine the current directory, `NULL` is returned and `errno` is set appropriately.

Your mission, whether or not you choose to accept it, is to write a function, `whereami()` that returns a pointer to a newly allocated string containing the absolute path of the current working directory. You may make no assumptions about the maximum path length, the returned string should be no larger than necessary, and you must be careful not to leak any memory. If it is not possible to determine the current working directory, return `NULL`.

(If necessary, you may continue on the following page.)

```
char *whereami() {
```

Optional extra space for problem 59.

```
}
```

66. () Write a C function called `isroot()` that takes a pathname as an argument and returns true (as C sees it) if that path represents a true path to the root directory and false otherwise. Write robust code.

    ```
    int isroot(const char *path) {
    ```

67. () Write a function `is_leaf_directory()` that takes a pathname and returns true if it is a directory and has no subdirectories, and false otherwise (either it has subdirectories or there was an error). You may assume that the macro `PATH_MAX` is defined if it is helpful to you. Write robust code.

    ```
    int is_leaf_directory(const char *path) {
    ```
    Optional extra space for problem 65.
    ```
    }
    ```

68. () Write a C function called `rmtree()` that takes a path as a parameter and recursively removes that directory tree rooted at that point in the filesystem. (Yes, a single file is a tree, too, just not an interesting one.) Return true if it was able to succeed, false otherwise. If it's helpful, you may assume the macro `PATH_MAX` is defined. Write robust code.

    ```
    int rm_treee(const char *victim) {
    ```
    Optional extra space for problem **??**.
    ```
    }
    ```

69. () Write a C function called `countfiles()` that takes a path as a parameter and returns a count of all regular files in the directory tree rooted at that point in the filesystem. (Yes, a single file is a tree, too, just not an interesting one.) If the function encounters errors, return -1. If it's helpful, you may assume the macro `PATH_MAX` is defined. Write robust code.

    ```
    int countfiles(const char *path) {
    ```
    Optional extra space for problem **??**.
    ```
    }
    ```

70. () **Failed to find problem: Exams/midterm/problems/2004-04.1**

71. () **Failed to find problem: Exams/midterm/problems/2004-04.2**

72. () **Failed to find problem: Exams/midterm/problems/2005-04.1**

73. () **Failed to find problem: Exams/midterm/problems/2008-01.3**

74. () **Failed to find problem: Exams/midterm/problems/2010-04.1**

75. () **Failed to find problem: Exams/midterm/problems/2005-04.2**

76. () At each indicated point below, show the contents of the given file. You may assume all system calls return successfully. Clearly mark the current end of the file (as with ⊗ below).

    The initial contents of the file, "`tf`", are:

`tf:`	T	h	s	1	?	R	l	l	y	?	⊗

```
char *filename = "tf";
char *backup  = "bob";
int one, two, three;

one = open(filename,O_RDWR);
```

```
  write(one,"grue?",5);
  write(one,"xyzzy!",6);        /* for the old–timers */
```

tf:																	

```
  symlink(filename,backup);
  two   = open(backup,O_WRONLY);
  three = dup(two);
  write(two,"Wayne",3);
  write(three," SNL",3);
```

tf:																	

```
  lseek(one,    12,  SEEK_SET);
  lseek(three, −5,  SEEK_END);
  write(three,"ears?",2);
```

tf:																	

```
  write(three,"kernel",1);
  write(one,"42",2);
```

tf:																	

```
  close(one);
  close(two);
  close(three);

  return 0;
}
```

77. () Write a C function that takes two path names (relative or absolute) and returns true if they refer to the same file and false if either file does not exist or they are not the same.

```
int samefile(const char *this, const char *that) {
```
Optional extra space for problem ??.
```
}
```

78. () If you forget your password (oops), why can't the sysadmin tell you what it was?

Useful Information

Prototypes (Also look in K&R)

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fildes, off_t offset, int whence);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int close(int fd);
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);
int utime(const char *filename, struct utimbuf *buf);
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
char *getcwd(char *buf, size_t size);
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int chdir(const char *path);
int fchdir(int fd);
DIR *opendir(const char *name);
void rewinddir(DIR *dir);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
off_t telldir(DIR *dir);
struct passwd *getpwnam(const char * name);
struct passwd *getpwuid(uid_t uid);
struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ...,
    char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *filename, char *const argv [],
    char *const envp[]);
int kill(pid_t pid, int sig);
pid_t fork(void);
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
int sigaction(int signum, const struct sigaction *act,
    struct sigaction *oldact);
int sigprocmask(int how, const sigset_t *set,
    sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
void (*signal(int signum, void (*sighandler)(int)))(int);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
int pipe(int filedes[2]);
unsigned int alarm(unsigned int seconds);
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value,
    struct itimerval *ovalue);
int tcgetattr ( int fd, struct termios *termios_p );
int tcsetattr ( int fd, int optional_actions, struct
    termios *termios_p );
int feof( FILE *stream);
int ferror( FILE *stream);
int fileno( FILE *stream);
```

Structures and Macros

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}


struct itimerval {
    struct timeval it_interval;
    struct timeval it_value;
};

struct timeval {
    long tv_sec;
    long tv_usec;
};


struct passwd {
        char    *pw_name;
        char    *pw_passwd;
        uid_t   pw_uid;
        gid_t   pw_gid;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};


struct group {
        char    *gr_name;
        char    *gr_passwd;
        gid_t   gr_gid;
        char    **gr_mem;
};


struct stat
{
    dev_t         st_dev;
    ino_t         st_ino;
    mode_t        st_mode;
    nlink_t       st_nlink;
    uid_t         st_uid;
    gid_t         st_gid;
    dev_t         st_rdev;
    off_t         st_size;
    unsigned long st_blksize;
    unsigned long st_blocks;
    time_t        st_atime;
    time_t        st_mtime;
    time_t        st_ctime;
};
```

Structures and Macros, cont.

```
S_ISLNK(m)   S_ISDIR(m)   S_ISBLK(m)
S_ISREG(m)   S_ISCHR(m)   S_ISFIFO(m)
                          S_ISSOCK(m)
```

Macro	Value	Macro	Value
S_IFMT	0170000	S_IRWXU	00700
S_IFSOCK	0140000	S_IRUSR	00400
S_IFLNK	0120000	S_IWUSR	00200
S_IFREG	0100000	S_IXUSR	00100
S_IFBLK	0060000	S_IRWXG	00070
S_IFDIR	0040000	S_IRGRP	00040
S_IFCHR	0020000	S_IWGRP	00020
S_IFIFO	0010000	S_IXGRP	00010
S_ISUID	0004000	S_IRWXO	00007
S_ISGID	0002000	S_IROTH	00004
S_ISVTX	0001000	S_IWOTH	00002
		S_IXOTH	00001

```
struct dirent
{
    long d_ino;
    off_t d_off;
    unsigned short d_reclen;
    char d_name [NAME_MAX+1];
}
```

```
struct utimbuf {
        time_t actime;
        time_t modtime;
};
```

```
struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t c_cc[NCCS];
}
```

```
WIFEXITED(status)
WEXITSTATUS(status)
WIFSIGNALED(status)
WTERMSIG(status)
WIFSTOPPED(status)
WSTOPSIG(status)
```