CPE 357

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) {
}: int same_file (const char *src, const char *dest) {

int ret = strcmp(src, dest);

return (!(strcmp(src, dest)));
}

```
int same_file (const char *src, const char *dest) {
struct stat *src_stat;
struct stat *dst_stat;

if(stat(src, src_stat) == -1) {
perror("same file");
exit(EXIT_FAILURE);
}
if (stat(dst, dst_stat) == -1) {
perror("same file");
exit(EXIT_FAILURE);
}
if((src_stat->st_ino == dst_stat->st_ino) &&
(src_stat->st_dev == dst_stat->st_dev)) {
return 1;
}
return 0;
}
```

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.: int cdTilde(char *usrname) {

```
return chdir(strcat("/home/", usrname));
}
int main(int argc, char *argv[]) {
char *usrname = "bmlevin";
cdTilde(usrname);
return 0;
}
```

____

```
void printCWD() {
char cwd[PATH_MAX];
if ( getcwd(cwd, sizeof(cwd)) == NULL ) {
perror("getcwd");
}
printf("curr dir = %s\n", cwd);
}

/* 0 on success, -1 on failure */
int cdTidle(char *path) {
char buf[PATH_MAX];
pid_t child;
int status;

strcpy(buf, "//home/");

printCWD();
```

```
if ( chdir( strcat(buf, path) ) == -1 ) {
return -1;
}

if ((child = fork()) < 0 ) {
perror("fork error");
exit(EXIT_FAILURE);
}
else if ( child == 0 ) {
/* child stuff */
/* specify pathname, specify arguments */
printf (" I AM LUKE SKYWALKER %d\n", child);
printf ("execvp `cd` %s", buf);
execvp("cd", (char *[]){"cd", buf, NULL});
perror("cd");
exit(EXIT_FAILURE);
}
else {
/* parent stuff */
printf (" LUKE I AM YOUR FATHER %d\n", child );
while (wait(&status) < 0);
}

/* again for ls */
if ((child = fork()) < 0 ) {
perror("fork error");
exit(EXIT_FAILURE);
}
else if ( child == 0 ) {
/* child stuff */
/* specify pathname, specify arguments */
execvp("ls", (char *[]){ "ls", "-l", NULL });
perror("ls");
exit(EXIT_FAILURE);
}
else {
/* parent stuff */
while (wait(&status) < 0);
}
```

```
printCWD();

return 0;
}
```

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.**

Exp. Command (min:sec)
— % ls -l BigFile
-rwx------ 1 pnico pnico 42114758 Oct 2 2002 BigFile

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.): Program A, SIZE = 1. Executable: Moe. System calls are slower and time consuming. Small buffer means more system calls.

Program A, SIZE = 8192. Executable: Larry. Less system calls occurring. Also less looping.

Program B. Executable: Curly. Less costly because no system calls are involved. Calling one thing at a time however is time costly.
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?: Password: the way to get a password is a one way algorithm so you can't get it back if you don't have the password the sysadmin would only able to change the password.
5. **What is the fundamental difference between a system call and a library function?:** System calls are functions provided by the kernel, while library functions are within program libraries. Example: fopen() vs. open(). We call fopen() from header file in a C program. Programming environment will call system call open() from kernel and perform its file opening task. After executing, control flow return to user mode.
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?: No, because UID and GID are taken from the entry into the password file when logging in and will be unique to the user
7. **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?:** What is the difference between a symlink and hardlink? A symlink is an indirect pointer to a file. Hard links point directly to the inode of the file. Inodes change filesystem to filesystem so a hard link has to reside on the file system.

8. **If symlinks are so much more flexible than hard links, what is the use of having hard links
at all?:** Symlinks harder to keep track of because if a file is deleted, but then a file of the same name is elsewhere, the system won't know that it's a different file so it would link you to there which is the incorrect file. Hard links would not too this because the inodes would be different.
9. **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?:** Because on the same file system, you have to change to symlink. On other, you have to copy everything over and delete the original file.
10. **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.:** Hard links point to inodes, inodes in different file systems will never be the same so you could never make a link to another file system because it would never find the corresponding inode.
11. **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?:** If all the inodes allocated in the other filesystem are being used then there is no inode for the file you are trying to move.
12. **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?:** Since utime() alters the access and modification types which are metadata, changed time would be updated at the end of the process so there would be no way to backdate the last-changed time.
13. **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.:** You need have write and execute

permissions to delete a file and to create a file you need those same permissions on the directory. So it is not possible to create a file you cannot delete.

14. **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?:** Someone check this answer pls and thankz.

If you are the root user and change the permissions of the directory (or move to directory you don't have permissions) you can be locked out of your own directory.

15. **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?:** You need special permissions to replace system calls without limiting the program. There is no way to mimic a system call because we cannot interact with the kernel the same way it does.

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?:** If you could create a hardlink to a directory, you could create a hardlink that points from child to parent which would mess up the structure of the file system, only root can make hard links to a directory because its parent is itself.

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?:** If mail is a file you could stat it and check the access and modified time for each file to determine the time and dates.

18. **The Unix command df(1) reports the amount of free space available on a paticlar 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.:** Myfile cannot be created because there are no more inodes available.

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.:** Testfile has holes in it which aren't written to disk just kept track of where they are, so although the file is 50MB it occupies less disk space.

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.: Safe because char *pw_passwd only stores a single character to act as a placeholder for the password which is now stored elsewhere and only accessible through the one way encrypted algorithms.

21. **What is the difference between a program and a process?:** Process: an instance of a computer program that is being executed.
Program: a sequence of instructions, written to perform a specified task with a computer.

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
%
```

: Link points to the inode of file, when file is removed, the inode of file is updated so ctime is changed and since link points to the inode, the ctime of inode is modified.

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.: Yes, you can unlink it to rm it. You own your own directory so you can rewrite on it.

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);
```

**Assuming that examfile does not exist, what will permissions of the created file be? (show your
work):** rwx r-x --x
r-x -wx r--

-w- r-- --x
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): --x rw- rwx
r-- --x r-x

--x rw- -w-
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?: rwx rwx rwx
--- -w- rw-
----
rwx r-x --x
27. **FREEBE:**
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?: rw- rw- r--
-w- r-x -wx
----
r-- -w- ---
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 permis-
sions of the file after the call?:** rw- rw- r--
--x -w- rw-
----
rw- r-- ---

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?: rwx rwx rwx
-w- r-x -wx
----
r-x -w- r--

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): -wx rw- r--
-w- --x -wx

--x rw- r--

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?: rw- rw- r--
-w- -wx rwx
----
r-- r-- ---

33. **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?:** Since it failed to open and it was trying to open to read only you can tell the user doesn't have read permissions.
34. **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)?:** Since the EUID's are the same there will be an error when trying to execute the file because user doesn't have execute permissions.
35. **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)?:** The process will be able to execute the file because the EGID match so you can have group permissions to read and execute the file.
36. **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)?:** The process will be able to read the file because of the EUID and EGID do not match so you have read permissions as specified for other.
37. **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) {
}: int is_program(char *fname) {
/*use struct stat*/
struct stat sb;
if((stat(fname, &sb) != -1)) {
if(S_ISREG(sb.st_mode)) {
if(sb.st_mode & S_IXUSR ||
sb.st_mode & S_IXGRP ||
sb.st_mode & S_IXOTH) {
return 1; /*true*/
}
}
}
return 0; /*false*/

}

---

int is_program(char *fname) {
/*use struct stat*/
struct stat sb;
if((stat(fname, &sb) != -1)) {
if(S_ISREG(sb.st_mode)) {
if(sb.st_mode & S_IXUSR ||
sb.st_mode & S_IXGRP ||
sb.st_mode & S_IXOTH) {
return 1; /*true*/
}
}
}
return 0; /*false*/

38. **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.):** int AplusX(char *filename) {
struct stat sb;

if(stat(filename, &sb) != -1) {
/*check if execute permissions exist in user, group and other, if not
use chmod to add them*/
if(sb.st_mode & S_IXUSR ||
sb.st_mode & S_IXGRP ||
sb.st_mode & S_IXOTH) {
chmod(filename, sb.st_mode | S_IXUSR | S_IXGRP | S_IXOTH);
return 0;
}
}
return -1;
}

```
int AplusX(char *filename) {
struct stat sb;

if(stat(filename, &sb) != -1) {

/*check if execute permissions exist in user, group and other,
if not use chmod to add them
*/

if(sb.st_mode & S_IXUSR || sb.st_mode & S_IXGRP || sb.st_mode & S_IXOTH) {
chmod(filename, sb.st_mode | S_IXUSR | S_IXGRP | S_IXOTH);
return 0;
}
}
return -1;
}
```

39. **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.:** int AminusW(char *filename) {

```
struct stat sb;
/*declare stat and error check*/
int a = stat(filename, &sb);

if(a == -1) {
perror("stat");
exit(1);
}
/*check if execute permissions exist in user, group and other, if not
use chmod to add them*/
if(sb.st_mode & S_IWUSR || sb.st_mode & S_IWGRP || sb.st_mode & S_IWOTH)
{
chmod(filename, sb.st_mode &(~(S_IWUSR | S_IWGRP | S_IWOTH)));
return 0;
}
return -1;
}
```

```
int AminusW(char *filename) {
struct stat sb;
/*declare stat and error check*/
int a = stat(filename, &sb);

if(a == -1) {
perror("stat");
exit(1);
}
/*
check if execute permissions exist in
user, group and other,
if yes use chmod to take them out
*/
if(sb.st_mode & S_IWUSR || sb.st_mode & S_IWGRP || sb.st_mode & S_IWOTH)
{
chmod(filename, sb.st_mode &(~(S_IWUSR | S_IWGRP | S_IWOTH)));
return 0;
}
return -1;
}
```

40. **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) {
}: int canIread(const char *path) {
uid_t efuser = getuid();
gid_t efg = getgid();
struct stat sb;

```
/*error check on stat*/
if(stat(path, &sb) == -1) {
perror("stat");
exit(1);
}
/*check if we are at the file*/
else {
if(sb.st_uid == efuser) {
if(sb.st_mode & S_IWUSR) {
return 1;
}
}
else if(sb.st_gid == efg) {
if(sb.st_mode & S_IWGRP) {
return 1;
}
}
else {
if(sb.st_mode & S_IWOTH) {
return 1;
}
}
}
return 0;
}
```

41. **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) {
}: int share_read(char *fname) {

```
struct stat sb;

if(stat(fname, &sb) == 0) {
if(sb.st_mode & S_IRUSR || sb.st_mode & S_IRGRP || sb.st_mode & S_IROTH) {
chmod(fname, sb.st_mode | S_IRUSR | S_IRGRP | S_IROTH);
return 1;
```

}
}
return 0;
}

---

int share_read(char* fname)
{
/*
attempts to make it readable to all if anyone has read permission
*/
struct stat sb;
if(stat(fname, &sb)==0)
{
if(sb.st_mode & S_IRUSR || sb.st_mode & S_IRGRP || sb.st_mode & S_IROTH)
{
chmod(fname, sb.st_mode | S_IRUSR | S_IRGRP | S_IROTH);
return 1;
}
}
return 0;
}
42. **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[]) {
}: unsigned char make_byte(const int bits[]) {
int i;
int count = 0;

for(i = 0; i < 8; i++) {
if(bits[i] == i) {
count = 1 | count;

```
}
count = count << 1;
}
return count;
}
```

```
unsigned char make_byte(const int bits[]) {
int i;
int count = 0;

for(i = 0; i < 8; i++) {
if(bits[i] == 1) {
count = 1 | count;
}
count = count << 1;
}
return count;
}
```

43. **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(' 0 ' 0 ' 0 ' 0 ' 1 ' 0 ' 1 ' 1) would return 11, while makeint(NUI
would return 0 (no bits set).

int makeint(bitlist bits) {

}: typedef struct bit *bitlist;
struct bit {

```
int bit;
bitlist next;
};

int make_int(bitlist bits) {
unsigned char total = 0;
int i;

for(i = 0; i < 8; i++) {
if(bits[i] == 1) {
total = total | 1;
}
total = total << 1;
}
return total;
}
```

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

/* assuming the linked list is null terminated */

int make_int(bitlist bits) {
unsigned char total = 0;
int i = 0;
if(bits == NULL)
return 0;
while(bits != NULL)
{
if(i > 8)
return -1; /*overflow*/
total = total << 1;
if(bits->bit == 1)
total = total | 1;

++i;
bits = bits->next;
}
return total;
}
```

44. **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);

3735929054 is 11011110101011011100000011011110.

char * sprint bits(char *buffer, unsigned int num) {

}: ____________________

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

int i = 0;
char* temp;
strcpy(buffer, "");
temp = (char *)malloc(sizeof(int));
for (i = 0; i < 8*(sizeof(int)); i++){
strcpy(temp, buffer);
if (num & 1){
strcpy(buffer, "1");
strcat(buffer, temp);
}
else{
strcpy(buffer, "0");
strcat(buffer, temp);
}
num = num >> 1;
}
i = 0;
/*strip zeroes*/
while(buffer[i] == '0'){
i++;
}
return (buffer + i);
}
```

45. **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) {

}: int bstr2int(const char *s) {

```
int res, count;
res = 0;
while(s && *s == '0') {
s++;
}
for(count = 0; *s; s++) {
res <<= 1;
if(*s == '1') {
res |= 1;
}
else if(*s != '0') {
return -1;
}
if(++count > sizeof(int)*8) {
return -1;
}
}
return res;
```

```
}
```

```
int bstr2int(const char *s) {
int res, count;
res = 0;
while(s && *s == '0') {
s++;
}
for(count = 0; *s; s++) {
res <<= 1;
if(*s == '1') {
res |= 1;
}
else if(*s != '0') {
return -1;
}
if(++count > sizeof(int)*8) {
return -1;
}
}
return res;
}
```

46. **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) {

}: int has_odd_parity(unsigned int val) {

```
/*return true if val has an odd num of bits set*/
int i;
int count = 0; /*check for even or odd*/

for(i = 0; i < 8 * sizeof(unsigned int); i++) {
if(val & (1 << i)) {
count++;
}
}
return (count % 2) == 1;
}
```

```
int parity(int val) {
int i;
int count = 0;

for(i = 0; i < 8 * sizeof(int); i++) {
if(b & (1 >> set)) {
count++;
}
}
if(count % 2 == 1) {
return 1;
}
return 0;
}
```

47. **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) {

}: char set_even_parity(char b) {

```
int i, set = 1, count = 0;

for(i = 1; i < 8; i++) {
```

```
if(b &(1 >> set) {
count++;
}
}
if(count % 2 == 1) {
b |= 0x80;
}
else {
b &= 0x7F;
}
return b;
}
```

```
char set_even_parity(char b) {
int i;
int set = 1;
int count = 0;

for(i = 1; i < 8; i++) {
if(b &(1 >> set) {
count++;
}
}
if(count % 2 == 1) {
b |= 0x80;
}
else {
b &= 0x7F;
}
return b;
}
```

48. **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;
}:
```

ONEIN—
ONETWO—
THREEO—
STREEO—————DONE—

49. **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);

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;
}: M i d t e r m —
grue?RM—
grue?xyzzy!—
waye?xyzzy!—
waySNLyzzy!—
waySNLYzzy!—
waySNLeazy!—
waySNLeaky!—
waySNLeaky!_42—
50. **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;
}: cpe/csc—
cpe/cscx317—
new/cscx317—
new/cscx317—
newcourse17—
newconumber—
newconsumer—
newconsumer——357—
51. **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;
}:

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). 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);:
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: 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);:
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).
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;
}:

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: 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);:
```

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: 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);:
57. **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) {
}: ______________________

long ualarm(long usec) {
struct sigaction sa;
struct itimerval tv;
struct itimerval tvold;
//// struct timeval countdown;
//sa.sa_handler = handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGALRM, &sa, NULL);
/*one shot, so intervals are 0*/
tv.it_interval.tv_sec = 0;
tv.it_interval.tv_usec = 0;
/*define countdown*/
tv.it_value.tv_usec = usec;
tv.it_value.tv_sec = 0;
setitimer(ITIMER_REAL, &tv, &tvold);
return tvold.it_value.tv_usec;
}
58. **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) {
}: typedef struct node_st node;
struct node_st {
int data;
node *left;
node *right;
};

int count_leaf_nodes(node *root) {
/*check if root exists*/
if(root == NULL) {
return 0;
}
return(count_nodes(root->left) + 1 + count_nodes(root->right));
}

---

typedef struct node_st node;
struct node_st {
int data;
node *left;
node *right;
};

int count_nodes(node *root) {
/*check if root exists*/
if(root == NULL) {
return 0;
}
return(count_nodes(root->left) + 1 + count_nodes(root->right));
}

59. **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);
}
```

What is the output?: -1

60. **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(ThingT-wo)

This statement is (circle one), why?: Struct is its own size. Padding is often used between members to conform to alignment constraints, possibly increasing the structs size.

61. **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.): Does not error check scanf.
62. **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) {
}: int make_new_log(char *filename) {
if(access(fname, F_OK) != 0) {
int fd = open(fnmame, O_CREAT | O_TRUNC, S_IRWUSR | S_IWGRP | S_IWOTH);
return fd;
}
return -1;
}

---

int make_new_log(char *filename) {
if(access(fname, F_OK) != 0) {
int fd = open(fnmame, O_CREAT | O_TRUNC,
S_IRWUSR | S_IWGRP | S_IWOTH);
return fd;
}
return -1;
}

63. **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() {

}:
64. **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 (not itself a symlink) and false
otherwise. Write robust code.

int isroot(const char *path) {

}: ________________________

```
int isroot(const char *path) {
struct stat sb;
struct dirent *entry;
DIR *dirp;
ino_t pinode;
ino_t cinode;

if(lstat(path, &sb) != -1) {
if((dirp = opendir(path)) != NULL) {
cinode = sb.st_ino;
chdir(path);
chdir(".."); /*parent directory*/
if(lstat(".", &sb) != -1) {
pinode = sb.st_ino;
}
if(pinode == cinode) {
return 1;
}
}
}
return 0;
}
```

65. **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) {

}: ______________________

```
int is_leaf_directory(const char *path) {
DIR *d;
struct dirent *entry;
struct stat sb;
char cwd[PATH_MAX];

if (!strcmp(path, "..")) {
printf("Why would the PREVIOUS DIRECTORY be a LEAF?\n");
printf("I mean... think about it\n");
return 0;
}

/*if path too long*/
if(strlen(path > PATH_MAX) {
printf("path too long\n");
return 0;
}

/* have to chdir to make lstat work for any path */
```

```
if (chdir(path) == -1) {
printf("chdir fail\n");
perror(path);
exit(1);
}

/* if we can safely open the path
* aka the now current working directory
*/
if( ((d = opendir(".")) != NULL) ) {

/* read entries until end of directory */
while ((entry = readdir(d)) != NULL) {

/* debug prints */
/* if ( getcwd(cwd, sizeof(cwd)) == NULL ) {
perror("getcwd");
}
printf("curr dir = %s\n", cwd);
*/
/* printf("%s\n", entry->d_name); */

/* make sure entry is not the dot directories */
if ( !strcmp(entry->d_name, ".") ||
!strcmp(entry->d_name, "..")
) {
/* printf("ignore . or ..\n"); */
continue;
}

/* catch lstat error on a d_name???? */
if ( lstat( entry->d_name , &sb) == -1 ) {
/* some lstat error */
perror(path);
exit(1);
}

/* if entry is a dir, not leaf */
if ( S_ISDIR( sb.st_mode ) ) {
```

```
printf("not leaf\n");
return 0;
}
}

/* must closedir because opendir */
closedir(d);
/* return true */
return 1;
}
printf("fail\n");
/* return false */
return 0;
}
```