(64.) Assumption: umask applies (we don't change it.

```
int make_new_log(char *fname){
    struct stat sb;
    int res=-1;
    if(stat(fname,&sb)==-1){
      res=open(fname, O_WRONLY|O_CREAT|O_TRUNC,
               (S_IRUSR|S_IWUSR|S_IRGRP|
                S_IWOTH);
    }
    return res;
}
```

Or access or open. ✓

```
extern int errno
#define SIZE 128
char *whereami(){
    char *buf = (char*) malloc(SIZE*sizeof(char));
    int len=SIZE;
    getcwd(buf, len);
    while(!buf && errno==ERANGE){
      len+=SIZE;
      buf=(char*) realloc(buf, len);
      getcwd(buf, len);
    }
```

(65.)

```
    /* now clip */
    if( errno == 0 || errno == ERANGE)
    buf = (char*) realloc( buf, strlen(buf) +1);

    return buf;
}
```

66)

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

    struct stat sroot, sb;
    if ( stat( "/" ,&sroot ) == -1) return 0;
    if(stat( path, &sb ) == -1) return 0;

    if ((sroot.st_ino == sb.st_ino) &&
        (sroot.st_dev == sb.st_dev)) return 1;

    else return 0;
}
```

67.

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

    struct stat sb;
    DIR * current;
    struct dirent *dirb;
    char pwd[pwd];
    current = opendir (path);
    if (! current) return 0;
```

```
pwd = getcwd(pwd, PATH_MAX);
if(!pwd) return 0;
if(chdir(path) == -1) return 0;

while((dirb = readdir(current)) != NULL)){
    if (strcmp(dirb->d_name, ".") ||
        strcmp(dirb->d_name, "..") continue;
    if(stat(dirb->d_name, &sb) == -1) return 0;
    if(S_ISDIR(sb.st_mode) return 0;
}
if(chdir(pwd) == -1) return 0;
return 1;
}
```

68.

```
int rm_treee(const char *victim){
  struct stat sb;                    <-- if(stat(victim,&sb)==-1)
  char pwd[PATH_MAX];                    return 0;
  getcwd(pwd, PATH_MAX);             if(S_ISDIR(sb.st_mode)){
  if(!pwd) return 0;
  DIR *current;
  struct dirent dirb;
  current = opendir(victim);
  if(!current) return 0;
  if(chdir(victim) == -1) return 0;
  while((dirb = readdir(current)) != NULL){
    if(!strcmp(dirb->d_name, ".") ||
       !strcmp(dirb->d_name, "..")) continue;
    if(!rm_treee(dirb->d_name)) return 0;}
  if(chdir(pwd) == -1) return 0;
if(closedir(current)==-1) return 0;
  if(rmdir(victim) == -1) return 0;
  else{
  if(unlink(victim) == -1) return 0;
}

return 1;
}
```

```
int countfiles(const char *path)
int res=0 struct stat sb;   <-- if(stat(victim, &sb)==-1) return 0;
    char pwd[PATH_MAX];
    getcwd(pwd, PATH_MAX); if(S_ISDIR(sb.st_mode)){
    if(!pwd) return -1;        int buf=0;
    DIR *current;
    struct dirent *dirb;
    current = opendir(victim);
    if(!current) return -1;
    if(chdir(victim) == -1) return -1;
    while((dirb=readdir(current)) != NULL){
        if(!strcmp(dirb->d_name, ".") ||
           !strcmp(dirb->d_name, "..")) continue;
    if(buf=countfiles(dirb->d_name) == -1)
        result += buf;          return -1;
        if(chdir(pwd) == -1) return -1;
if(closedir(current) == -1) return 0;

    else{
    result += 1;
    }

    return result;
}
```

1.

```
struct stat f1, f2;
if(stat(this, f1) == -1) return 0;
if(stat(that, f2) == -1) return 0;
if( f1.st_dev == f2.st_dev &&
    f1.st_ino == f2.st_ino) return 1;
else return 0;
```

2.

```
int cdTilde(const char *username){
struct passwd *pw = getpwnam(username+1);
if(!pw) return -1;
return chdir(pw->pw_dir);
}
```

Program has 42774758 bytes

3.

pA -> moe -> with SIZE 1 -> 42774758 iterations which include ~~--~~ system calls (a lot of time)

pB -> larry -> with SIZE 8192 -> 42774758/8192 iterations and system calls (way less)

pC -> curley -> with getchar (buffered IO) it needs less system calls (buffers input in the background) but still has 42774758 iterations to get each char from that buffer.

(4.) No password is generated with one way algorithm
↳ sys admin can only look up hashed password or change the password.

(5.) Fundamental difference -> system calls can not be replaced (at least it is really hard). Library functions can be replaced. Because system calls directly invoke requests in the kernel. Lib functions are mostly abstractions of common programming tasks.

(6.) No it is not. Only root can change the uid of a process. -> File is created with uid/gid of process that creates it.

(7.) a) hard link links name to i-node number (i-nodes are only unique within a filesystem). Different devices can have same i-nodes. A symlink however links a name to a name (-> works across filesystems because they are unique).

b) „hard links keep the universe together." e.g. the contents of the i-nodes stay the same (even if new link to i-node and old one removed). with a symlink these can quickly be chaos. (files can be renamed and new files with that name created) -> symlink points to wrong file.

(8.) mv on a filesystem just needs to link new location to i-node and unlink old location. (contents of file are not moved). To mv across filesystems we need to move the contents of the file (because the i-node is only unique within a filesystem).

(9.) hard links link a name to an i-node. i-node numbers are only unique within a filesystem. (different filesystems can have the same i-node). → can not link across filesystems.

(10.) For mv within filesystem we only need to link new location to i-node and unlink old one. (Data of file does not need to be moved). To different filesystem data needs to be moved. If we do not have read permission on file → we can't move it.

(11.) The ctime is the last time the i-node was changed. changing the m- & a-time which are stored in the i-node require means changing the i-node → c-time gets updated to when the change happend.

12. It is not! To open/create a file you need write & execute permission on the directory.
→ same permissions are needed to delete a file.
→ you can delete.

13. same as 12.

14. Syslog calls invoke direct requests to the kernel. recreating that behaviour is very hard / impossible. Lib functions are used to simplify common programming tasks.
→ can be easily replaced.

15. see 5.

16. can create a loop in the filesystem.

17. call utime on the mail file.
↳ access time → unread since
↳ modification time → new mail recieved

18. no more i-nodes available

. to create a file a filesystem needs enough space & i-nodes i-nodes is limited. number of

19. The „testfile" has holes in it.

20. Shared is a hashed version of the password that was generated via a one way algorithm (need pw & salt) to get that.

27. program is a bundle of instructions designed to perform a certain task.

process is a running instance of a program (that's where it gets interesting)

22. rm -> i-node link number -> ctime is inode change time

23. to rm file we only need write/execute perm on dir (user certainly has them)

	5	3	4
umask:	101	011	100
~umask:	010	100	011
& permission: mask	111	101	001
↳ permissions:	010	100	001
a) ⇒	-w-	r--	--x
	user	group	other

b) open can only set permission of newly created file
↳ existing file keeps old permission.

35. euid = 4 ⟷ fuid = 4
egid = 7

uid matches → only user permissions apply
⇒ no x perm for user ⇒ x fails
(perm denied)

36. euid = 4 ⟷ fuid = 12
egid = 7 ⟷ fgid = 7
→ uid no match ⇒ gid match ⇒ only group permissions are looked at. ⇒ group has x perm
⇒ perm granted ⇒ x succeeds

37. euid = 10 ≠ fuid = 12

egid = 7 ≠ fgid = 10

=> other permissions are looked at => perm granted

=> process can read file.

38.
```
int is_program(char *fname){
    struct stat sb;
    if(stat(fname, &sb)==-1) return 0;
    if(!S_ISREG(sb.st_mode)) return 0;
    return (sb.st_mode & (S_IXUSR | S_IXGRP
                          S_IXOTH));
```

39.
```
int AplusX(const char *path){
    struct stat sb;
    if(stat(path, &sb)==-1) return -1;
    if(!S_ISREG(sb.st_mode)) return -1;
```
everyone already has execute permission
```
    if( sb.st_mode & S_IXUSR && sb.st_mode & S_IXGRP
        && sb.st_mode & S_IXOTH) return 0;
    if(sb.st_mode & (S_IXUSR | S_IXGRP | S_IXOTH))
        return chmod(path, (sb.st_mode | S_IXUSR | S_S_I + GRP | S_IXOTH));
    else return -1;
```

(40.)
```
int Aminus w (const char *path){
    struct stat sb;
    if(stat(path, &sb)==-1)return -1;
```

noone has w → perm to begin with

```
    if(!S_ISREG(sb.st_mode))return -1;
    if(!(sb.st_mode & (S_IWUSR | S_IWGRP | S_IWOTH))
        return 0;
    return chmod(path, sb.st_mode & ~(S_IWUSR | S_IWGRP |
                                      S_IWOTH));
```

(41.)
```
int canIread(const char *path){
    struct stat sb;
    if (stat(path, &sb) == -1) return 0;
    uid_t euid = geteuid();
    gid_t egid = getegid();
```

first check uid perm

```
    if (sb.st_uid == euid)
        return (sb.st_mode & S_IRUSR);
    else if (sb.st_gid == gid)
        return (sb.st_mode & S_IRGRP);
    else
        return (sb.st_mode & S_IROTH);
```

(42.) same as before

#define bits 8

(44.)
```
const int bits[]{
int i;
unsigned char res = 0;
for (i=0; i<bits; i++) {
    res = (res | (bits[i] << (bits - i - 1)));
return res;
```

(45.)
```
{
while (bits->bit == 0 && bits->next)
    bits = bits->next;
int res = 0;
int overflow = 0;
while (bits) {
if (overflow > 31) return -1;
res = (res << 1 ) | bits->bit;
```

```
        bits = bits->next;
        overflow++;
    }
    return res;
```

46.

```
int i;
for (i=0; i<31; i++) {
    if((num >> (31-i)) & 1))
        buffer[i] = '1';
    else buffer[i] = '0';
}
bits[32] = '\0';
return buffer;
```

47.

```
#define MAX sizeof(int) *8;
if (!s) return 0;
int result = 0;
int i;
int overflow;
while (s[i] == '0') i++;
while (s[i] != '\0') {
```

```
if (overflow >(MAX - 7)) return - 1;
    if(s[i] == '1') result=(result<<1) | 1;
    else  result = result << 1;
    i++;
    overflow++;
return result;
```

48.

```
count = 0;
int i;
for (i = 0; i < 32; i++) {
    if ((a >> i) & 1) count++;
}
return (count % 2);
```

49.

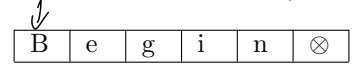
```
count = 0;
int i;
for(i = 0; i < 7; i++) {
    if ((b >> i) & 1) count++;
}
if count % 2:
    b = (b) | (1 << 7);
```

```
else
    b = (b) & ~(7<<7);
return b;
```

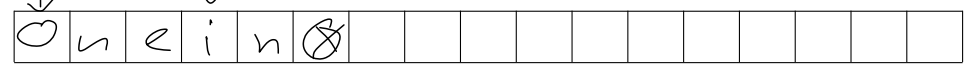
0.4 Follow the bouncing ball

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:

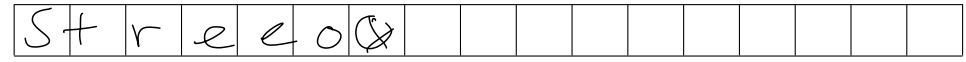



```
int main(int argc, char *argv[]){
  char *filename = "tf";
  char *newfile  = "other";
  int one, two, three;

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

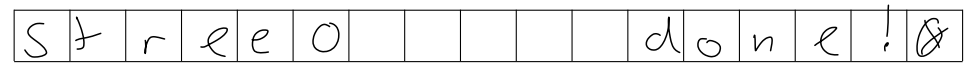



```
  two   = dup(one);
  link(filename,newfile);
  three = open(newfile,O_WRONLY);
  write(two,"two",3);
  write(three,"three",5);
```


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



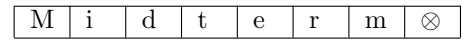

```
  write(three,"done!",5);
```



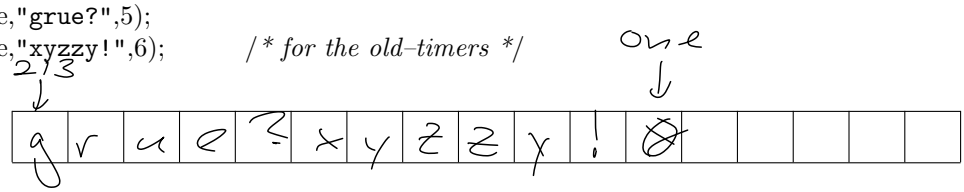

```
  close(one);
  close(two);
  close(three);

  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).

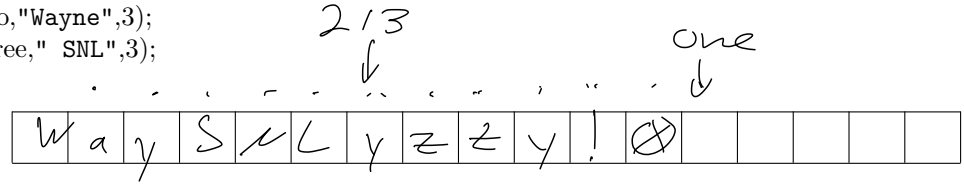
The initial contents of the file, "`tf`", are:



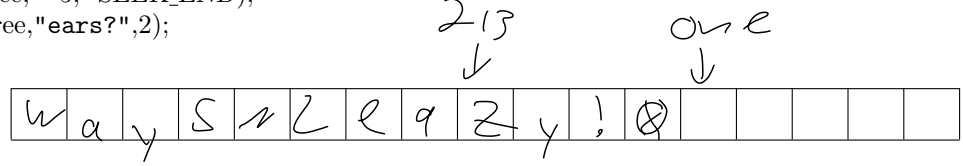
```
char *filename = "tf";
char *backup  = "bob";
int one, two, three;

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



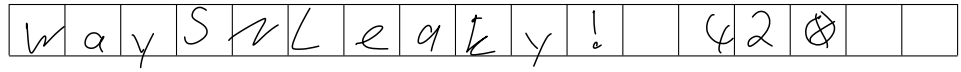

```
symlink(filename,backup);
two   = open(backup,O_WRONLY);
three = dup(two);
write(two,"Wayne",3);
write(three," SNL",3);
```




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




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




```
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);
```

```
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: 

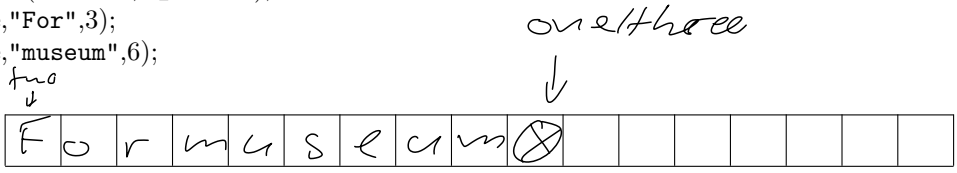
```
close(one);
close(two);
close(three);
```

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).

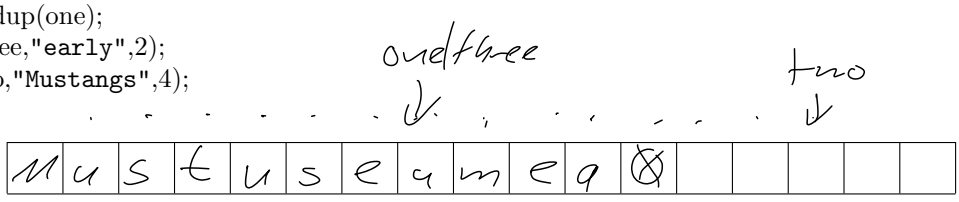
The initial contents of the file, "`tf`", are:

tf:	W	i	n	d	o	w	s	?	⊗

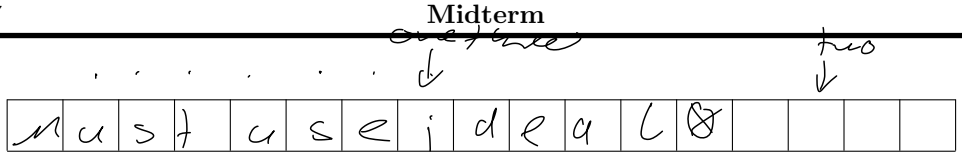
```
char *filename = "tf";
int one, two, three;

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




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




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




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

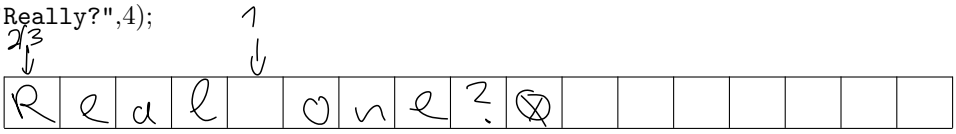

```
close(one);
close(two);
close(three);

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.**

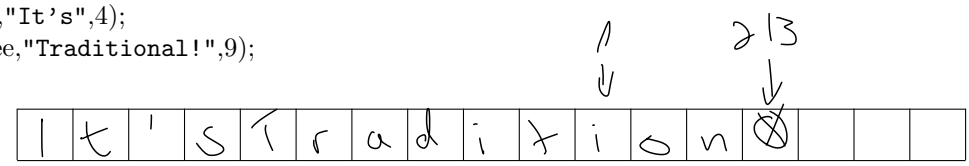
The initial contents of the file, "`tf`", are:

`tf:`	T	h	i	s		o	n	e	?	⊗

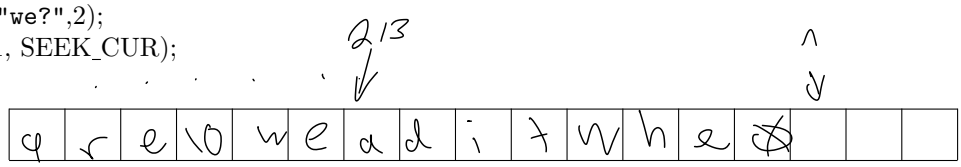
```
char *filename="tf";
int one, two, three;

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




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




```
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);
```




```
lseek(two,0,SEEK_SET);
```

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

tf: 

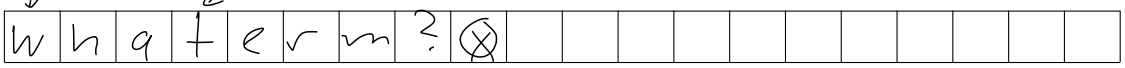
```
close(one);
close(two);
close(three);
```

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).

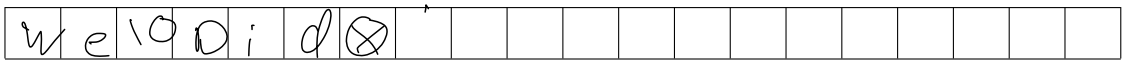
    The initial contents of the file, "`tf`", are:

`tf:`	M	i	d	t	e	r	m	?	⊗

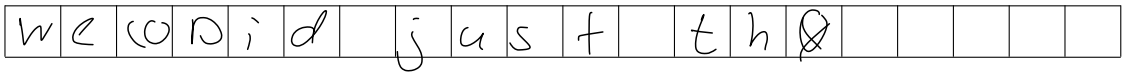
```
int one, two, three;
char *filename="tf";

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

tf: 

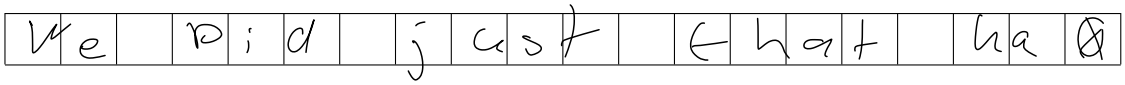
```
two   = open(filename,O_WRONLY|O_TRUNC);
three = dup(one);
write(three,"Didn't",3);
write(two,"We",3);
```

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);
```

Networks

17.3: Declaration specifies the attributes (such as the data type) of a set of identifiers.
If the declaration also causes storage to be allocated, it is called a definition.

17.6: 1. Race between call to stat and the call to unlink, where the file can change.
2. Stat follows a symlink if symlink points to a regular file we would remove the symlink and not the socket file. → use lstat.

Signals:

10.7: pause returns -1 and program exits (errno is set to EINTR)
→ pause returns whenever a signal is caught.

10.3:

Process Lifecycle:

Chapter 7:

7.2:

Process fork() & wait()

8.5:

MYARG2

8.6:

```
main
:
:

if (fork() == 0);
    exit(0);
exit(0);
```

64.
```
int make_new_log(char *fname){
    struct stat sb;
    if(stat(fname, &sb) > -1) return -1;
    return open(fname, O_WRONLY, (S_IRUSR|S_IWUSR|S_IWOTH));
```

Unreliable Signal Handling:

-> signal handler is called with the signal number

-> (maybe) the handler is reset to SIG_DFL

-> (maybe) the further deliver of the same signal is blocked

-> (maybe) slow system calls are interrupted, returning error and setting errno to EINTR

-> reliable signal handling. ACC this is done with certainty

-> provides support for more timing, but makes the interface more complicated.

Task: set up SIGINT & blocking over a critical section.

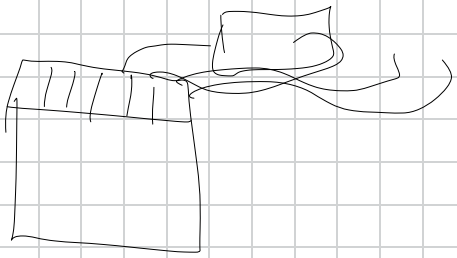
```
void sigint handler(int num){
    printf("Hello :)\n");
}
int main(int argc, char* argv[]){
    sigset_t new, old;
    struct sigaction sa;

    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = sigint handler;

    sig emptyset(&new);
    sig add set(&new, SIGINT);
    sigprocmask(SIG_BLOCK, &new, &old);

if(sigaction(SIGINT, &sa, NULL) == -1)
        perror("sigaction");
        exit(1);

    printf("important code!\n");

    sigprocmask(SIG_SETMASK, &old, NULL);

    return 0;
}
```

Rohit

```
int main(int argc, char *argv[]) {
    printf("Hello World!\n");
    pid_t child = fork();
    if (child == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    else if (child == 0) {
        printf("PID: %d\n", getpid());
        exit(EXIT_SUCCESS);
    }
    else {
        printf("PID: %d\n", getpid());
        if (wait(NULL) == -1) {
            perror("wait");
            exit(EXIT_FAILURE);
        }
        printf("Bye!\n");
    }
    return 0;
}
```

```
int main(int argc, char *argv){
  /* command check */
    char *fname;
    pid_t child = fork();
    int status;
    if(child == -1){
        perror("fork");
        exit(EXIT_FAILURE);
    else if(child == 0){
        execl(fname, fname, (char*) NULL);
        perror("execl");
        exit(EXIT_FAILURE);

    else{
        if(wait(&status) == -1){
            perror("wait")
            exit(EXIT_FAILURE);
        }
        if(WIFEXITED(status) == 0){
            printf("child exited with status %d",
                WEXITSTATUS(status));

        else{
            perror("error")
        }
        else{
```

profit (abnormally)



```
int   p_one[2];
int   p_two[2];

pid_t  child1, child2, child3;

if((child1=fork()) == -1){
    perror(fork);
    exit(1);

else if(child1==0){

if( dup2(p_one[1], STDOUT_FILENO) ==-1){
        perror(dup2);
        exit(1);
    close(p_one[1]);
    close(p_one[0]);
    close(p_two[1]);
    close(p_two[0]);

    execlp("ls", "ls", (char *) NULL);
    perror("ls");
    exit(1);
}
```

```
if ((child2 = fork()) == -1)
        perror fork
        exit(1);
}
else if(child2==0)
if (dup2(p_one[0], STDIN_FILENO)==-1) {
    error
}
if (dup2(p_two[1], STDOUT_FILENO)==-1){
    error
}
    close(p_one[1]);
    close(p_one[0]);
    close(p_two[1]);
    close(p_two[0]);

    execlp("sort", "sort", "-r", (char *)NULL);
    perror("sort");
    exit(1);

if ((child3=fork())==-1){ error
else if(child3==0){
if(dup2(p_two[0], STDIN_FILENO)==-1){
    error
```

```
close(p_one[1]);
close(p_one[0]);
close(p_two[1]);
close(p_two[0]);

execlp("move", "move", (char *)NULL);
perror("move");
exit(1);
}
/* parent */

close(p_one[1]);
close(p_one[0]);
close(p_two[1]);
close(p_two[0]);

for(int i=0; i<3; i++)
    if(wait(NULL)==-1)
        error
    }

exit(0);
```

```
struct pollfd {
    int fd;
    int events;
    int revents;
};

int poll(struct pollfd *fds, int nfds, int timeout);

struct pollfd fd[2];
fd[0].fd = STDIN_FILENO
fd[0].events = POLLIN;
fd[0].revents = 0;
fd[1] = fd[0];
fd[1].fd = unmask_socket;
for(;;) {
    poll(fd, 2, -1);
    if (fd[0].revents & POLLIN) {
        /* stdin */
    }
    if (fd[1].revents & POLLIN) {
        /* handle network */
    }
}
```

Networking:

TCP:

Server:

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in sa, newsockinfo, peer;

sa.sin_family = AF_INET;
sa.sin_port = htons(port);

sa.sin_addr.s_addr = htonl(INADDR_ANY);

bind(sockfd, (struct sockaddr*) &sa, sizeof(sa));

int newsock = accept(sockfd, (struct sockaddr *)
                     &peer, sizeof(peer));
```

65.

```
#define LEN 706
char *xbreadmit();

char *path = NULL;
int len = SIZE;

while (1) {
  path = (char*) realloc(path, len * sizeof(char));
  path = getcwd(path, len);
  if (path)
    break;
  else if (errno != ERANGE && errno != 0) {
    return NULL;
  }
  len += SIZE
}
path = realloc(path, strlen(path) + 1);

return path
```

66.

```
int isroot(const char *path) {
struct stat sroot, spath;
if (stat("/", &sroot) == -1) return 0;
if(stat(path, &spath) == -1) return 0;
```

return ((sroot.st_dev == spath.st_dev) &&
(sroot.st_ino == spath.st_ino));

57. int is_leaf_director (const char *path) {
struct stat sb; int res = 1;
if (stat (path, &sb) == -1) return 0;
if (! S_ISDIR (sb.st_mode) return 0;
char cwd [PATH_MAX + 1];
getcwd (cwd, PATH_MAX);
if (! cwd) return 0;
cwd [PATH_MAX] = '\0';
if (chdir (path) == -1) return 0;
DIR *current;
struct dirent *dirb;
current = opendir (".");
if (! current) return 0;
while ((dirb = readdir (current)) != NULL) {
if (! strcmp (dirb.d_name, ".") &&
! strcmp (dirb.d_name, "..")) continue;
if (stat (dirb.d_name, &sb) == -1) res = 0;
if (S_ISDIR (sb.st_mode)) res = 0;
}

```
    if ( closedir(current) == -1) res= 0;
    if( chdir(cwd) == -1)  res= 0;

  return res;
}
```

68.

```
int rm_treee(const char *victim){
struct stat sb;
if ( stat(victim, &sb) == -1) return 0;
if (! S_ISDIR(sb.st_mode)){
    if(unlink (victim) == -1) return 0;
    return 1;
}
else{
char *cwd[PATH_MAX+7];
getcwd(cwd, PATH_MAX);
if(!cwd) return 0;
cwd[PATH_MAX] = '\0';
if (chdir(victim) == -1) return 0;
 DIR *current;
 if (opendir(".") == -1) return 0;
 int res = 1;
 struct dirent *dirp;
 while((dirp = readdir(current) != NULL){
```

```
if ( !strcmp(dirp->d_name, ".") ||
     !strcmp(dirp->d_name, "..") continue;
if (stat(dirp->d_name, &sb) == -1) res = 0;
if( S_ISDIR(sb.st_mode)) {
    if (!rm_tree(dirp->d_name)) res = 0;
} else {
    if (unlink(dirp->d_name) == -1) res = 0;
}
}
if( closedir(current) == -1) res=0;
if( chdir("cwd") == -1) res=0;
if( rmdir(victim) == -1) res=0;
return res;
}
}
```

69.

```
int countFiles(const char *path){
    int result = 0;
    struct stat sb;
    if(lstat(path, &sb) == -1) return -1;
    if (S_ISREG(sb.st_mode)) return 1;

    if(S_ISDIR(sb.st_mode)){
        int buf = 0;
        char cwd[PATH_MAX];
        getcwd(cwd, PATH_MAX);
        if(!cwd) return -1;
        if(chdir(path) == -1) return -1;
        DIR *current;
        current = opendir(".");
        if(!current) return -1;
        struct dirent *dirp;
        while((dirp= readdir(current)) != NULL){
            if(!strcmp(dirp->d_name, ".") ||
                !strcmp(dirp->d_name, "..")) continue;
            if(lstat(dirp->d_name, &sb) == -1) return -1;
            if(S_ISREG(sb.st_mode)) result++;
            if(S_ISDIR(sb.st_mode)){
                if((buf = countFiles(dirp->d_name)) == -1) return -1;
                result += buf;
            }
```

```
        if (closedir(current) == -1) return -1;
        if (chdir(cwd) == -1) return -1;
        return result;
    }
    return 0;
}
```

Signal handling:

```
void sigintHandler(int num) {
    printf("Meeee neener\n");
}
int main(int argc, char *argv[]) {
struct sigaction sa, sold;
sa.sa_handler = sigintHandler;
if( sigemptyset(sa.sa_mask) == -1)
        perror("sigemptyset");
        exit(EXIT_FAILURE);
}
sa.sa_mask = 0;
if( sigaction(SIGINT, &sa, &sold) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
}
sigset_t newmask, oldmask;
if(sigemptyset(&newmask) == -1) {perror("sigemptyset")
    exit(1);
}
if(sigaddset(&newmask, SIGINT) == -1)
```

```
        perror("sigaddset");
        exit(EXIT_FAILURE);
    }

    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) == -1) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    printf("Important stuff!\n");

    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```