Error Handling
Low-Level I/O
Signals

Spring 2022

Some useful hints

- Most header files in `/usr/include`
- If in doubt: `echo | gcc -E -Wp,-v -`
- For man pages, typically section 2: `man -s 2 fcntl`

Error Handling

- Potential errors/mistakes have to be anticipated and corresponding corrective action (if possible) should be adopted.

- Instead of using an `fprintf()`, the call `perror()` could be used:

  ```
  void perror(char *estring)
  ```

  - The above prints out the string pointed to by `estring` denoting a specific kind of a mistake (choice of the programmer).

- Should we include the header file `#include <errno.h>` the variable `errno` will have as its value an integer corresponding to the latest error that occurred.

C program with Error Handling

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(){
   FILE *fp=NULL;  char *p=NULL; int stat=0;

   fp=fopen("a_non_existent_file","r");
   if (fp == NULL) {
    printf("errno = %d \n", errno);
    perror("fopen");
    }

   p=(char *)malloc(2147483647);
   if (p==NULL) {
    printf("errno = %d \n",errno);
    perror("malloc");
    }
   else {
    printf("Carry on\n");
    }

   stat=unlink("/etc/motd");
   if (stat == -1) {
    printf("errno = %d \n",errno);
        perror("unlink");
    }

   return(1);
}
```

Running the errors_demo.c executable

```
antoulas@sazerac:~/src$ gcc errors_demo.c
antoulas@sazerac:~/src$ ./a.out
errno = 2
fopen: No such file or directory
Carry on
errno = 13
unlink: Permission denied
antoulas@sazerac:~/src$
```

Low-Level Input/Output

- The `stdio` library enables the average user carry out I/Os without worrying about buffering and/or data conversion.
- The `stdio` is a user-friendly set of system calls.
- Low-level I/O functionality is required when
  1. the amenities that `stdio` are not desirable (for whatever reason) in accessing files/devices, or
  2. interprocess communication (IPC) occurs with the help of pipes/sockets.

Low-Level I/Os

- In low-level I/O, file descriptors that identify files, pipes, sockets and devices are small integers.
  - The above is in contrast to what happens in the `stdio` where respective identifiers are file pointers (for formatted I/O).

- Designated (fixed) file descriptors:

  0 : standard input
  1 : standard output
  2 : standrad error (for error diagnostics).

- The above file descriptors 0, 1, and 2 correspond to pointers to the `stdin` `sdtout` and `stderr` files of the `stdio` library.

- The file descriptors are parent-“inherited” to any child process that the parent in question creates.

The open() system call

```
int open(char *pathname, int flags [, mode_t mode])
```

- The call opens or creates a file with absolute or relative `pathname` for reading/writing.
- `flags` designate the way (i.e., a number) with which the file can be accessed; the value for `flags` may be constructed by a bitwise-inclusive `OR` of flags from the following set:
  - `O_RDONLY`: open for reading only.
  - `O_WRONLY`: open for writing only.
  - `O_RDWR`: open for both reading and writing.
  - `O_APPEND`: write at the end of the file.
  - `O_CREAT`: create a file if it does not already exists.
  - `O_TRUNC`: size of file is to be truncated to 0, if file exists.

The `open()` system call

- required: `#include <fcntl.h>`
  ⇒ `fcntl.h` defines all these (and more) flags.
- The not-compulsory `mode` parameter is an integer that designates the desired access primitives during the creation of a file (access rights not allowed from the `umask` are not allowed).
- `open` returns an integer that designates the file created and in case of no success, it returns -1.

createfile.c

```
#include <stdio.h>  // to have access to printf()
#include <stdlib.h> // to enable exit calls
#include <fcntl.h>  // to have access to flags def
#define PERMS 0644  // set access permissions

char *workfile="mytest";

main(){
    int filedes;

    if ((filedes=open(workfile,O_CREAT|O_RDWR,PERMS))==-1){
        perror("creating");
        exit(1);
        }
    else {
        printf("Managed to get to the file successfully\n");
        }
    exit(0);
}
```

Running the executable for createfile.c

```
antoulas@sazerac:~/src$ gcc createfile.c
antoulas@sazerac:~/src$ ./a.out
Managed to get to the file successfully
antoulas@sazerac:~/src$ ls -l
total 20
-rwxr-xr-x 1 ad ad 8442 2010-04-06 21:50 a.out
-rw-r--r-- 1 ad ad  375 2010-04-06 21:49 createfile.c
-rw-r--r-- 1 ad ad  506 2010-04-06 16:24 errors_demo.c
-rw-r--r-- 1 ad ad    0 2010-04-06 21:50 mytest
antoulas@sazerac:~/src$ cat > mytest
This is Kon Tsakalozos
antoulas@sazerac:~/src$ ./a.out
Managed to get to the file successfully
antoulas@sazerac:~/src$ ls
a.out  createfile.c  errors_demo.c  mytest
antoulas@sazerac:~/src$ more mytest
This is Kon Tsakalozos
antoulas@sazerac:~/src$
```

Setting modes with symbolic names

S_IRWXU	00700 owner has read, write and execute permission
S_IRUSR	00400 owner has read permission
S_IWUSR	00200 owner has write permission
S_IXUSR	00100 owner has execute permission
S_IRWXG	00070 group has read, write and execute permission
S_IRGRP	00040 group has read permission
S_IWGRP	00020 group has write permission
S_IXGRP	00010 group has execute permission
S_IRWXO	00007 others have read, write and execute permission
S_IROTH	00004 others have read permission
S_IWOTH	00002 others have write permission
S_IXOTH	00001 others have execute permission

Working with access modes

```c
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
```

1. If the call to `open()` is successful, the file is opened for reading/writing by the user.
2. Those in the “group” and “others” can read the file.

The creat() call

```
int creat(char *pathname, mode_t mode);
```

- The creat is an alternative way to create a file (istead of using open()).
- pathname is any UNIX pathname giving the target location in which the file is to be created.
- mode helps set up the access rights.
- creat will always truncate (an existing file before returning its file descriptor).

```
filedes = creat("/tmp/tsak",0644);
```

is equivalent to:

```
filedes = open("/tmp/tsak", O_WRONLY|O_CREAT|O_TRUNC, 0644);
```

The `read()` call

```
ssize_t read(int filedes, char *buffer, size_t n)
```

- Reads at most `n` bytes from a file, device, end-point of a pipe, socket that is designated by `filedes` and place the bytes on `buffer`.

- The call returns the number of bytes *successfully read*, 0 if we are past the last byte-already read, and -1 if a problem occurs.

- When do we read less bytes?

1. The file has less characters left to be read.
2. The operation is "interrupted" by a signal.
3. Reading on pipe/socket takes place and a character becomes available (in which case a while-loop is needed to read all characters).

Using the read() call (count.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define BUFSIZE 27

main(){
    char buffer[BUFSIZE]; int  filedes; ssize_t nread; long total=0;

    if ((filedes=open("anotherfile", O_RDONLY))== -1){
        printf("error in opening anotherfile \n");
        exit(1);
        }

    while ( (nread=read(filedes,buffer,BUFSIZE)) > 0 )
        total += nread;
    printf("Total char in anotherfile %ld \n",total);
    exit(0);
}
```

Running the executable:

```
antoulas@sazerac:~/src$ ./a.out
Total char in anotherfile 936
antoulas@sazerac:~/src$
```

- What happens if `char *buffer=NULL;` is used instead of `char buffer[BUFSIZE];` ??

The write() and close() system calls

```
ssize_t write(int filedes, char *buffer, size_t n);
```

- The call writes at most n bytes of content from the buffer to the file that is described by filedes.
- write returns the *number of bytes successfully written out* to the file or -1 in case of failure.
- use the write call with: #include <unistd.h>

```
int close(int filedes);
```

- releases the file descriptor filedes; returns 0 in case of successful release and -1 otherwise.
- use the close call with: #include <unistd.h>

Working with open, read, write and close calls

Write a program that appends the content of a file at the very end of the content of another file.

```c
#include  <stdio.h>
#include  <string.h>
#include  <stdlib.h>
#include  <fcntl.h>
#include  <unistd.h>
#include  <sys/stat.h>
#define BUFFSIZE 1024

int main(int argc, char *argv[]){
  int n, from, to; char buf[BUFFSIZE];
  mode_t fdmode = S_IRUSR|S_IWUSR|S_IRGRP| S_IROTH;

  if (argc!=3) {
    write(2,"Usage: ", 7); write(2, argv[0], strlen(argv[0]));
    write(2," from-file to-file\n", 19); exit(1); }

  if ( ( from=open(argv[1], O_RDONLY)) < 0 ){
    perror("open"); exit(1); }

  if ( (to=open(argv[2], O_WRONLY|O_CREAT|O_APPEND, fdmode)) < 0 ){
    perror("open"); exit(1); }

  while ( (n=read(from, buf, sizeof(buf))) > 0 )
    write(to,buf,n);
  close(from); close(to); return(1);
}
```

Execution Outcome:

```
antoulas@sazerac:~/src$ ls
anotherfile   count.c       dupdup2file    mytest
               writeafterend.c
a.out         createfile.c  errors_demo.c  mytest1
buffeffect.c  dupdup2.c     filecontrol.c  readwriteclose.c
antoulas@sazerac:~/src$ more mytest
This is Konstantinos Tsakalozos
antoulas@sazerac:~/src$ more mytest1
that I use to show something silly
use to show something silly
to show something silly
antoulas@sazerac:~/src$ ./a.out
Usage: ./a.out from-file to-file
antoulas@sazerac:~/src$ ./a.out mytest mytest1
antoulas@sazerac:~/src$ cat mytest1
that I use to show something silly
use to show something silly
to show something silly
This is Konstantinos Tsakalozos
antoulas@sazerac:~/src$
```

Using open read, write and close calls

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(){
  int fd, bytes, bytes1, bytes2;
  char buf[50];

  mode_t fdmode = S_IRUSR|S_IWUSR;

  if ( ( fd=open("t", O_WRONLY | O_CREAT, fdmode ) ) == -1 ){
        perror("open");
        exit(1);
        }

  bytes1 = write(fd, "First write. ", 13);
  printf("%d bytes were written. \n", bytes1);
  close(fd);

  if ( (fd=open("t", O_WRONLY | O_APPEND)) == -1 ){
         perror("open");
        exit(1);
        }

  bytes2 = write(fd, "Second Write. \n", 14);
  printf("%d bytes were written. \n", bytes2);
  close(fd);
  if ( (fd=open("t", O_RDONLY)) == -1 ){
        perror("open");
        exit(1);
        }

  bytes=read(fd, buf, bytes1+bytes2);
  printf("%d bytes were read \n",bytes);
  close(fd);

  buf[bytes]='\0';
  printf("%s\n",buf);
  return(1);
}
```

Running the program..

```
antoulas@sazerac:~/src$ ls
anotherfile  count.c       errors_demo.c  readwriteclose.c
a.out        createfile.c  mytest
antoulas@sazerac:~/src$ ./a.out
13 bytes were written.
14 bytes were written.
27 bytes were read
First write. Second Write.
antoulas@sazerac:~/src$ ls
anotherfile  count.c       errors_demo.c  readwriteclose.c
a.out        createfile.c  mytest         t
antoulas@sazerac:~/src$
```

Copying a file with variable buffer size

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#define  SIZE           30
#define  PERM           0644

int mycopyfile(char *name1, char *name2, int BUFFSIZE){
        int infile, outfile;
        ssize_t nread;
        char buffer[BUFFSIZE];

        if ( (infile=open(name1,O_RDONLY)) == -1 )
                return(-1);

        if ( (outfile=open(name2, O_WRONLY|O_CREAT|O_TRUNC, PERM)) == -1){
                close(infile);
                return(-2);
                }

        while ( (nread=read(infile, buffer, BUFFSIZE) ) > 0 ){
                if ( write(outfile,buffer,nread) < nread ){
                        close(infile); close(outfile); return(-3);
                        }
                }
        close(infile); close(outfile);
```

Copying a file with variable buffer size

```
        if (nread == -1 ) return(-4);
        else    return(0);
}

int main(int argc, char *argv[]){
        int     status=0;

        status=mycopyfile(argv[1],argv[2],atoi(argv[3]));
        exit(status);
}
```

Running the program for various size buffers..

```
antoulas@sazerac:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 8192
real    0m0.012s user   0m0.000s sys    0m0.012s
antoulas@sazerac:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 4096
real    0m0.010s user   0m0.000s sys    0m0.008s
antoulas@sazerac:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 256
real    0m0.071s user   0m0.000s sys    0m0.072s
antoulas@sazerac:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 32
real    0m0.454s user   0m0.012s sys    0m0.444s
antoulas@sazerac:~/src$ time ./a.out /tmp/stuff.ppt /tmp/alex1 1
real    0m13.738s user  0m0.428s sys    0m13.305s
antoulas@sazerac:~/src$
```

lseek call

```
off_t lseek(int filedes, off_t offset, int start_flag);
```

- `lseek` repositions the offset of the open file associated with `filedes` to the argument `offset` according to the directive `start_flag` as follows:
  1. SEEK_SET: The offset is set to `offset` bytes; usual actual integer value = 0
  2. SEEK_CUR: The offset is set to its current location plus `offset` bytes; usual actual integer value = 1
  3. SEEK_END: The offset is set to the size of the file plus `offset` bytes. usual actual integer value = 2

```
off_t newposition;
...
newposition=lseek(fd, (off_t)-32, SEEK_END);
```

Positions the read/write pointer 32 bytes BEFORE the end of the file.

The `fcntl()` system call

```
int fcntl(int filedes, int cmd);
```

```
int fcntl(int filedes, int cmd, long arg);
```

```
int fcntl(int filedes, int cmd, struct flock *lock);
```

- provides (some) control over already-opened files; headers required: `<sys/types.h>`, `<unistd.h>`, `<fcntl.h>`.
- `fcntl()` performs one of the operations described below on the open file descriptor `filedes`. The operation is determined by `cmd` – values for the `cmd` appear in the `<fcntl.h>`.
- Value of *3rd param* (`arg`) depends on what `cmd` does.
- Among other operations, `fcntl()` carries out two commands:
  1. `F_GETFL`: Read file status flags; `arg` is ignored.
  2. `F_SETFL`: Set file status flags to value specified by `arg`.

A routine for checking the flags of an open file

```c
#include <fcntl.h>

int filestatus(int filedes){
    int myfileflags;

    if ( (myfileflags = fcntl(filedes,F_GETFL)) == -1){
        printf("file status failure\n"); return(-1);
        }
    printf("file descriptor: %d ",filedes);
    switch ( myfileflags & O_ACCMODE ){ //test against the open file flags
    case O_WRONLY:
        printf("write-only"); break;
    case O_RDWR:
        printf("read-write"); break;
    case O_RDONLY:
        printf("read-only"); break;
    default:
        printf("no such mode");
    }
    if ( myfileflags & O_APPEND ) printf("- append flag set"); printf("\n");
    return(0);
}
```

⇒ `&` : bitwise AND operator

⇒ `fcntl` can be used to acquire record locks (or locks on file segments).

calls: `dup`, `dup2`

```
int dup(int oldfd);
```

returns the lowest-numbered unused descriptor as the new descriptor.

```
int dup2(int oldfd, int newfd);
```

makes `newfd` be the copy of `oldfd` - note:

1. If `oldfd` is not a valid file descriptor, then the call fails, and `newfd` is not closed.
2. If `oldfd` is a valid file descriptor, and `newfd` has the same value as `oldfd`, then `dup2()` does nothing, and returns `newfd`.

- After a successful return from one of these system calls, the old and new file descriptors may be used *interchangeably*.

Example of dup and dup2

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(){
  int fd1, fd2, fd3;
  mode_t fdmode = S_IRUSR|S_IWUSR|S_IRGRP| S_IROTH;

  if ( ( fd1=open("dupdup2file", O_WRONLY | O_CREAT | O_TRUNC, fdmode ) ) == -1
       ){
    perror("open");
    exit(1);
  }
  printf("fd1 = %d\n", fd1);
  write(fd1, "What ", 5);
  fd2=dup(fd1);
  printf("fd2 = %d\n", fd2);
  write(fd2, "time", 4);
  close(0);

  fd3=dup(fd1);
  printf("fd3 = %d\n", fd3);
  write(fd3, " is it", 6);
  dup2(fd2, 2);
  write(2,"?\n",2);
  close(fd1); close(fd2); close(fd3);
  return 1;
}
```

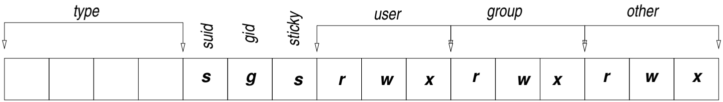
Execution Outcome:

```
antoulas@sazerac:~/src$ ls
anotherfile   count.c       dupdup2file    mytest
a.out         createfile.c  errors_demo.c  readwriteclose.c
buffeffect.c  dupdup2.c     filecontrol.c
antoulas@sazerac:~/src$ ./a.out
fd1 = 3
fd2 = 4
fd3 = 0
antoulas@sazerac:~/src$ ls
anotherfile   count.c       dupdup2file    mytest
a.out         createfile.c  errors_demo.c  readwriteclose.c
buffeffect.c  dupdup2.c     filecontrol.c
antoulas@sazerac:~/src$ cat dupdup2file
What time is it?
antoulas@sazerac:~/src$
```

Accessing inode information with stat()

- `int stat(char *path, struct stat *buf);`
  `int fstat(int fd, struct stat *buf);`
  returns information about a file; `path` points to the file (or `fd`) and the `buf` structure helps "carry" all derived information.
- such information includes:
  1. `buff→st_dev`: ID of device containing file
  2. `buff→st_ino`: inode number
  3. `buff→st_mode`: the last 9 bits represent the access rights of owner, group, and others. The first 4 bits indicate the type of the node (after a bitwise-AND with the constant S_IFMT, if the outcome is S_IFDIR, the node is a catalog, if outcome is S_IFREG, the mode is a regular file etc.)
  4. `buff→st_nlink`: number of hard links
  5. `buff→st_uid`: user-ID of owner
  6. `buff→st_gid`: group ID of owner
  7. `buff→st_size`: total size, in bytes
  8. `buff→st_atime`: time of last access
  9. `buff→st_mtime`: time of last modification of content
  10. `buff→st_ctime`: time of last status change

st_mode is a 16-bit quantity



type | suid | gid | sticky | user | group | other

				s	g	s	r	w	x	r	w	x	r	w	x

1. 4 first bits indicate the type of the file (16 possible values - less than 10 file types are in use now: regular file, dir, block-special, char-special, fifo, symbolic link, socket).
2. the next three bits set the flags: `set-user-ID`, `set-group-ID` and the `sticky` bits respectively.
3. next three groups of 3 bits a piece indicate the *read/write/execute* access right for the the groups: `owner`, `group` and `others`.
4. masking can be used to decipher the permissions each file system entity is given.

stat-ing inodes

- The fields `st_atime`, `st_mtime` and `st_ctime` designate time as number of seconds past since 1/1/1970 of the Coordinated Universal Time (UTC).
- The function `ctime` helps bring the content of the fileds `st_atime`, `st_mtime` and `st_ctime` in a more readable format (that of the `date`). The call is:

```
char *ctime(time_t *timep);
```

- `stat` returns 0 if successful; otherwise, -1
- Header files needed: `<sys/stat.h>` and `<sys/types.h>`
- `int fstat(int fd, struct stat *buf);` is identical to `stat` but it works with *file descriptors*.
- `int lstat(char *path, struct stat *buf);` is identical to `stat`, except that if path is a *symbolic link*, then the link itself is stat-ed, **not** the file that it refers to.

Definitions in <sys/stat.h>

```
#define      S_IFMT      0170000      /* type of file*/
#define      S_IFREG     0100000      /* regular */
#define      S_IFDIR     0040000      /* directory */
#define      S_IFBLK     0060000      /* block special */
#define      S_IFCHR     0020000      /* character sspecial */
#define      S_IFIFO     0010000      /* fifo */
#define      S_IFLNK     0120000      /* symbolic link */
#define      S_IFSOCK    0140000      /* socket */
```

Testing for a specific type of a file is easy using code fragments of the following style:

```
if ( (info.st_mode & S_IFMT) == S_IFIFO )
    printf("this is a fifo queue.\n");
```

Accessing information from inode

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>

int main(int argc, char *argv[]){
        struct stat statbuf;

        if (stat(argv[1], &statbuf) == -1)
                perror("Failed to get file status");
        else {
                printf("Time/Date  : %s",ctime(&statbuf.st_atime));
                printf("---------------------------------\n");
                printf("entity name: %s\n",argv[1]);
                printf("accessed   : %s", ctime(&statbuf.st_atime)+4);
                printf("modified   : %s", ctime(&statbuf.st_mtime));
                }
        return(1);
}
```

Running the program..

```
antoulas@sazerac:~/src-set004$ ./samplestat git.pdf
Time/Date  : Mon Mar 21 10:12:30 2016
---------------------------------
entity name: git.pdf
accessed   : Mar 21 10:12:30 2016
modified   : Mon Mar 21 10:11:55 2016
antoulas@sazerac:~/src-set004$
```

Accessing Catalog Content

- The catalog content (ie, pairs of *inodes* and *node names*) can be accessed with the help of the calls: `opendir`, `readdir` and `closedir`.
- Accessing of a catalog happens via a pointer `DIR *` (similar to the `FILE *` pointer that is used by the `stdio`).
- Every item in the catalog is weaved around a structure called `struct dirent` that includes the following two elements:
  1. `d_ino`: inode number;
  2. `d_name[]`: a character string giving the filename (null terminated)
- Using these calls, it is not feasible to change the content of the directory or its structure.
- Required header files: `<sys/types.h>` and `<dirent.h>`

calls: opendir, readdir, closedir

- `DIR *opendir(char *name)`:
  1. Opens up the catalog termed name and returns a pointer type DIR for accessing the catalog.
  2. If there is a mistake, the call returns NULL
- `struct dirent *readdir(DIR *dirp);`
  1. the call returns a pointer to a dirent structure representing the next directory entry in the directory pointed to by dirp
  2. if for the current entry, the field d_ino is 0, the respective entry has been deleted.
  3. returns NULL if there are no more entries to be read.
- `int closedir(DIR *dirp);`
  1. closes the directory associated with dirp
  2. function returns 0 on success. On error, -1 is returned, and errno is set appropriately.

Example

```
#include  <stdio.h>
#include  <sys/types.h>
#include  <dirent.h>

void    do_ls(char dirname[]){
DIR     *dir_ptr;
struct  dirent *direntp;

if ( ( dir_ptr = opendir( dirname ) ) == NULL )
        fprintf(stderr, "cannot open %s \n",dirname);
else {
        while ( ( direntp=readdir(dir_ptr) ) != NULL )
                printf("inode %d of the entry %s \n", \
                           (int)direntp->d_ino, direntp->d_name);
        closedir(dir_ptr);
        }
}

int main(int argc, char *argv[]) {
if (argc == 1 ) do_ls(".");
else while ( --argc ){
                printf("%s: \n", *++argv ) ;
                do_ls(*argv);
        }
}
```

Execution Outcome

```
antoulas@sazerac:~/src-set004$ ./openreadclosedir
inode 11403323 of the entry myreadlink
inode 11403324 of the entry myctime
inode 11403322 of the entry .
inode 11403325 of the entry dupdup2
inode 11403326 of the entry signal-example
inode 10883777 of the entry count
inode 11403328 of the entry myalarm1.c
inode 11403310 of the entry errors_demo
inode 11403330 of the entry signal-ignore.c
inode 11403331 of the entry morewithls.c
inode 11403332 of the entry myalarm.c
inode 11403393 of the entry openreadclosedir.c
inode 10883835 of the entry t
inode 11403335 of the entry myreadlink.c
inode 11403336 of the entry samplestat.c
inode 11403305 of the entry ..
inode 11403337 of the entry signal-exampleD
inode 10883705 of the entry createfile
inode 11403339 of the entry jj.ps

.....
antoulas@sazerac:~/src-set004$ ./openreadclosedir
```

Creating a program that behaves as ls -la

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
                          /* eight distinct modes */
char *modes[]={"---","--x","-w-","-wx","r--","r-x","rw-","rwx"};
void list(char *);
void printout(char *);

main(int argc, char *argv[]){
struct stat mybuf;

if (argc<2) { list("."); exit(0);}

while(--argc){
  if (stat(*++argv, &mybuf) < 0) {
        perror(*argv); continue;
        }
  if ((mybuf.st_mode & S_IFMT) == S_IFDIR )
        list(*argv);      /* directory encountered */
  else  printout(*argv);  /* file encountered      */
  }
}
```

Creating a program that behaves as `ls -la`

```c
void list(char *name){
DIR     *dp;
struct dirent *dir;
char    *newname;

    if ((dp=opendir(name))== NULL ) {
        perror("opendir"); return;
        }
    while ((dir = readdir(dp)) != NULL ) {
        if (dir->d_ino == 0 ) continue;
        newname=(char *)malloc(strlen(name)+strlen(dir->d_name)+2);
        strcpy(newname,name);
        strcat(newname,"/");
        strcat(newname,dir->d_name);
        printout(newname);
        free(newname); newname=NULL;
        }
    closedir(dp);
}
```

Creating a program that behaves as `ls -la`

```c
void printout(char *name){
struct stat     mybuf;
char            type, perms[10];
int             i,j;

    stat(name, &mybuf);
    switch (mybuf.st_mode & S_IFMT){
    case S_IFREG: type = '-'; break;
    case S_IFDIR: type = 'd'; break;
    default:      type = '?'; break;
    }

    *perms='\0';

    for(i=2; i>=0; i--){
        j = (mybuf.st_mode >> (i*3)) & 07;
        strcat(perms,modes[j]);
        }
        printf("%c%s%3d %5d/%-5d %7d %.12s %s \n", \
                type, perms, (int)mybuf.st_nlink, mybuf.st_uid, \
                mybuf.st_gid, (int)mybuf.st_size, \
                ctime(&mybuf.st_mtime)+4, name); /* try without 4 */
}
```

```
antoulas@sazerac:~/src-set004$ ./morewithls mydir morewithls.c
drwx------ 10  1000/1000      4096 Mar  9 07:51 mydir/.
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/b
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/e
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/d
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/a
drwx------  4  1000/1000      4096 Mar 12 13:24 mydir/..
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/f
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/h
-rwxr-xr-x  1  1000/1000       750 Mar  9 07:51 mydir/j
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/g
-rwxr-xr-x  1  1000/1000        12 Mar  9 07:51 mydir/k
drwx------  2  1000/1000      4096 Mar  9 07:51 mydir/c
-rwxr-xr-x  1  1000/1000       368 Mar  9 07:51 mydir/i
-rwxr-xr-x  1  1000/1000      1680 Mar 12 13:18 morewithls.c
antoulas@sazerac:~/src-set004$
```

link and unlink

```
int link(char *oldpath, char *newpath)
```

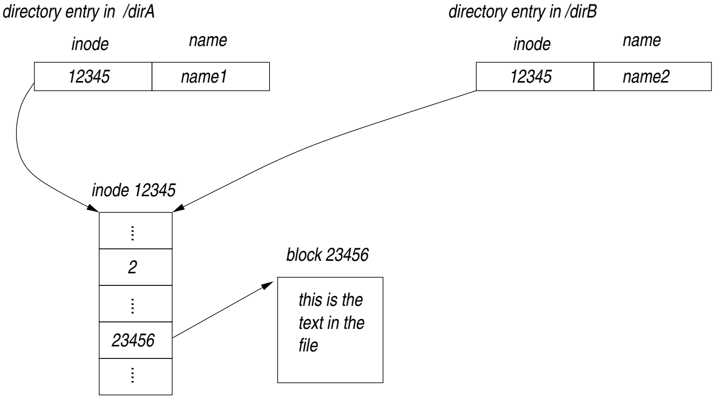
- It creates an new hard link to an existing file. If newpath exists, it will not be overwritten.
- The created link essentially connects the inode of the oldpath with the name of the newpath.

```
int unlink(char *pathname)
```

- Deletes a name from the file system; if that name is the last link to a file and no other process have the file open, the file is deleted and its space is made available.

Example on `link()`

```
#include <stdio.h>
#include <unistd.h>
....
if ( link("/dirA/name1","/dirB/name2")== -1 )
    prerror("Failed to make a new hard link in /dirB");
....
```



chmod, rename calls

```
int chmod(char *path, mode_t mode)
```

```
int fchmod(int fd, mode_t mode)
```

- Change the permissions (on files with `path` name or having an `fd` descriptor) according to what `mode` designates.
- On success, 0 is returned; otherwise -1

```
int rename(const char *oldpath, const char *newpath)
```

- Renames a file, moving it between directories (indicated with the help of `oldpath` and `newpath`) if required.
- On success, 0 is returned; otherwise -1

symlink and readlink calls

```
int symlink(const char *oldpath, const char *newpath)
```

- Creates a symbolic link named `newpath` that contains the string `oldpath`.
- A symbolic link (or soft link) may point to an existing file or to a nonexistent one; the latter is known as a *dangling link*.
- On success, zero is returned. On error, -1 is returned, and `errno` is set appropriately.

```
ssize_t readlink(char *path, char *buf, size_t bufsiz)
```

- Places the content of the symbolic link `path` in the buffer `buf` that has size `bufsiz`.
- On success, `readlink` returns the number of bytes placed in `buf`; otherwise, -1.