CHAPTER IV: UNIX FILE INPUT/OUTPUT

B. Boufama

University of Windsor

Introduction

Most Unix I/O can be performed using the system calls :

- *open*: to open a file
- *creat*: to create a new file or rewrite an existing one
- *read*: to read a number of bytes
- *write*: to write a number of bytes
- *lseek*: to explicitly position the file offset
- *close*: to close a file

In contrast to the std I/O functions, the Unix I/O system calls are unbuffered.

Each open file has associated with it a file offset. This is a pointer to the place in the file where the next read or write will start. When a file is first opened, the file offset is set to the beginning of the file, byte zero. Each time that bytes are read, the file offset is automatically advanced to the next unread byte.

File descriptors : The kernel refers to any open file by a file descriptor, a nonnegative integers.
In particular, the standard input, standard output and standard error have descriptors 0, 1 and 2 respectively.
The symbolic constants, defined in $< unistd.h >$, for these values are *STDIN_FILENO*, *STDOUT_FILENO* and *STDERR_FILENO*.

File descriptors range from 0 through *OPEN_MAX*, the total number of files a process can open.

open() system call

Synopsis :
*int open(const char *fName, int oflag[, int mode])*
Returns file descriptor if OK, -1 otherwise.
The argument *oflag* is formed by **OR**'ing together 1 or more of the following constants (in $< fcntl.h >$)

- **O_RDONLY** : Open for reading only
- **O_WRONLY** : Open for writing only
- **O_RDWR** : Open for reading and writing only
- **O_APPEND** : Open for writing after the end of file
- **O_CREAT** : Create a file

Note that the third argument, only used when a file is created, supplies the initial file's permission flag settings, as an octal value.

Examples :

- **if ((d=open("data.txt", O_RDONLY))==-1)**
  fatalError();
- **d=open(name,O_CREAT|O_WRONLY,0700)**
  In this case, 0700 means give all rights to the owner of this file and no permission to any other user.
  Example values for mode :
  - 0400: Allow read by owner
  - 0200: Allow write by owner
  - 0100: Allow execute
  - 0040: Allow read by group
  - 0004: Allow read by others
  - 0777: Allow read/write/execute by all

  Check the utility **umask** and **chmod**

```
F or D    OW GR OT
          rwx rwx rwx
          421 421 421

     777 = 111 111 111
```

128 64 32 16 8 4 2 1

read() and write() system calls

char contentsRead[6];

read() synopsis: long read(sample.txt, contentsRead, 6

*ssize_t read(int fd, void *buf, size_t nbyte);*

Reads as many bytes as it can, up to *nbyte*, and returns the number of bytes actually read.

ssize_t and *size_t* are usually defined as long integer(larger than an integer).

The value returned by *read()* can be:

returns -1 for error, 0 if file empty and number of bytes read

- -1 : in case of an error
- smaller than *nbyte*: the number of bytes left before the end of file was less than *nbyte* or, when reading from a keyboard where up to a line is normally read or, when reading from a network.
- 0 : the end of file has been already reached

write() synopsis:

*ssize_t write(int fd, const void *buf, size_t nbyte);*

write returns *nbyte* if OK and -1 otherwise.

Example: a **copy** utility.

contents from buffer are written to the file

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

int main(int argc, char *argv[]){
 int fd1, fd2;
 char buffer[100];
 long int n1;
```

file to open

```
 if(((fd1 = open(argv[1], O_RDONLY)) == -1) ||
    ((fd2 = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC,
                 0700)) == -1)){
    perror("file problem ");
    exit(1);
 }
```

if it is 0 then nothing is there to read from the file

```
 while((n1=read(fd1, buffer, 100)) > 0)
   if(write(fd2, buffer, n1) != n1){
```

This returns the number of bytes written in the file that is same as read bytes that is why !n1

```
      perror("writing problem ");
      exit(3);
   }
```

```
// Case of an error exit from the loop

 if(n1 == -1){
   perror("Reading problem ");
   exit(2);
 }
 close(fd2);
 exit(0);
}
```

Note that *int close(int fd)* frees the file descriptor fd. If no other file descriptor is associated with a particular open file, the kernel frees all resources that were used for that file.
close() returns 0 when OK and -1 otherwise.
For example, -1 will be returned if *fd* was already closed.

lseek() system call

Synopsis :

retuns the current position of the curson after the whence operation is done

off_t lseek(int fd, off_t offset, int whence);

Return the resulting offset if OK, -1 otherwise.

The return type *off_t* is a long integer.

Similar to *fseek()*, it sets the file pointer(position) associated with the open file descriptor specified by the file descriptor *fd* as follows:

the pointer is set from starting point + number given in the offset

- If whence is SEEK_SET, the pointer is set to offset bytes.
- If whence is SEEK_CUR, the pointer is set to its current location plus offset.
- If whence is SEEK_END, the pointer is set to the size of the file plus offset.

These three constant are defined in $< unistd.h >$.

umask

System call **umask()** and command **umask** allow the settings of the user *mask* that controls newly created file permissions.
Basically, each mask digit is negated, then applied to the default permissions using a logical AND operation.
umask() system call
Synopsis:
mode_t umask(mode_t mask);
umask() sets the calling process's file mode creation mask (umask) to mask & 0777
It returns the previous value of the mask.
Needed header files:
<sys/types.h> and <sys/stat.h>

Examples:

umask(0000): this is a neutral mask, does not stop any perission.
So, when we do **open(file, O_CREAT, 0777)**, we will get the permissions we asked for.

However, if we do
umask(0222): no write permission
So, when we do **open(file, O_CREAT, 0777)**, we will not get write permissions.
Note: we can still use chmod() to overwrite umask.

umask(0xyz): where x, y or z can take any octal value, 0..7, with the following effects

0 is neutral, 1 blocks execute, 2 blocks write, 3 blocks write/execute, 4 blocks read, 5 blocks read/execute, 6 blocks read/write and 7 blocks everything.

Command **umask** does a simlar job as the system call **umask()**

Synopsis:
umask [-S] [**mask**]
When option **-S** is present, accept symbolic representation of mask
When no mask is provided, **umask** returns the current user mask.

Examples: umask -S g+w: allow write permission for my group, if requested.
umask 0000: make your mask neutral
umask 0077: no permission for your group and others.

The user mask acts as a safety measure that disables some permissions when files are created.