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.

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**

read() and write() system calls

read() synopsis:
*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 :

- -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.

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

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

 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);
 }

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

- 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 $< unistd.h >$.

Note :
lseek() allows the file pointer to be set beyond the existing data in the file. If data are later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes of value 0 until data are written into the gap.

Example:
Write a C program, called reverse, to take a file as input then copies it to another file in reverse order. That is, the last byte becomes the first, the byte just just before the last one becomes the second, etc.
The program call should look like:
reverse fileIn fileOut

There are two solutions, either we use *lseek()* on *fileIn* (read bytes fron the end) or we use *lseek()* on *fileOut*. (write bytes strting at the end)

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

```
// solution 1: Start reading  at the end
//             Use lseek() on fileIn
int main(int argc, char *argv[]){
 int fd1, fd2;
 char buffer; // 1 character buffer
 long int i=0, fileSize=0;

 fd1=open(argv[1], O_RDONLY);
 fd2=open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0755);

 while(read(fd1, &buffer, 1)>0)
   fileSize++;
 while(++i <= fileSize){
   lseek(fd1, -i, SEEK_END);
   read(fd1, &buffer, 1);
   write(fd2, &buffer, 1);
 }
 close(fd1);
 close(fd2);
}
```

```
//solution 2: start writing at end
int main(int argc, char *argv[]){
 int fd1, fd2;
 char buffer; // 1 character buffer
 long int i=0, fileSize=0;

 fd1=open(argv[1], O_RDONLY);
 fd2=open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0755);

 while(read(fd1, &buffer, 1)>0)
   fileSize++;
 lseek(fd2, fileSize-1, SEEK_SET);//pointer to end
 lseek(fd1, 0, SEEK_SET);   // rewind fd1!!
 while(++i <= fileSize){
   read(fd1, &buffer, 1);
   lseek(fd2, -i, SEEK_END);
   write(fd2, &buffer, 1);
 }
 close(fd1);
 close(fd2);
}
```