UNIX Tutorial Sheet #2

Topics Covered:

1. READ AND WRITE SYSTEM CALL
2. FILE SEEK (lseek FUNCTION CALL)
3. DUPLICATING FILE DESCRIPTOR (dup and dup2 SYSTEM CALLS)
4. SAVING AND DESTROYING FILES(sync, fsync, fdatasync SYSTEM CALLS)
5. TRUNCATING FILES(truncate and ftruncate SYSTEM CALLS)
6. SOLVED EXAMPLES and EXERCISE

----

1. Read Function/ System Call:

Data is read from an open file with the **read** function.

```
#include<sys/types.h>
#include<sys/uio.h>
#include <unistd.h>
ssize_t read(int filedes, void *buf, size_t nbytes);
```

First argument is File Descriptor attached with file opened for reading, second is buffer address where read data will be received, third is maximum bytes to be read.

If the read is successful, the number of bytes read is returned. If the end of file is encountered, 0 is returned. If any error occurs then read function will return -1.

The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

2. Write Function/ System Call:

Data is written to an open file with the **write** function.

```
#include<sys/types.h>
#include<sys/uio.h>
#include <unistd.h>
ssize_t write(int filedes, const void *buf, size_t nbytes);
```

The return value is usually equal to the nbytes argument; otherwise, an error has occurred.

It will write the nbytes pointed by buffer in file handled by file descriptor filedes.

For a regular file, the write starts at the file's current offset. If the O_APPEND option was specified when the file was opened, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

The *write* call does not actually write the contents to the disk. It just transfers data to an intermediate place called buffer cache and returns. At some later point (which can be ANY arbitrary time), the kernel transfers the data from the buffer cache to the actual place in disk where the file resides. The user process may never be able to find out when the data was actually transferred to the disk. This delayed writing accounts for the speeds up the *write* call. This is because read or write to buffer caches are very fast compared to read or write to disk. This technique is referred to as **delayed writing**.

Solved Exercise 1:

Program to copy a file using read and write. It reads from standard input and writes to standard output. Execute this program below:

```
#include<sys/types.h>
#include<sys/uio.h>
#include <unistd.h>
#define BUFFSIZE 4096
int main()
{
     int n;
     char buf[BUFFSIZE];
     while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
     if (write(STDOUT_FILENO, buf, n) != n)
     err_sys("write error");
     if (n < 0)
     err_sys("read error");
     exit(0);
}
```

In this example, we use the two defined names, STDIN_FILENO and STDOUT_FILENO, from <unistd.h>. The program doesn't close the input file or output file. Instead, the program uses the feature of the UNIX kernel that closes all open file descriptors in a process when that process terminates.

Solved Exercise 2:

A Complete IO Program: Using open(), creat(), read(), write() and close()

/* This program reads contents from an existing file "some.txt" and copies the contents into another newly created file "new.txt" */

```
# include<stdio.h>
# include<fcntl.h>
# include<unistd.h>
# include<string.h>
# include<sys/types.h>
# include<sys/uio.h>

int main(int argc, char **argv)
{
     int read_bytes,n,fd,fd1;
     char buf[128];
     char *pathname="some.txt";

     /* In the open call permissions are not mandatory */
     fd=open(pathname,O_RDONLY);
     if(fd== -1)    {
          printf("Error opening %s\n",pathname);
          return 1;
     }

     while(1){
          read_bytes=read(fd,buf,sizeof(buf));
          if(!read_bytes)
               break;
          if(read_bytes==-1) {
               printf("Error reading file %s\n",pathname);
               return 2;
          }
          /* The permissions in creat call can be set either
          using octal or symbolic representation*/
          fd1=creat("new.txt",777);
          n=write(fd1,buf,read_bytes);



          if(n== -1) {
               printf("Error writing to stdout\n");
               return 3;
          }
     }
     close(fd);
     close(fd1);
     return 0;
}
```

3. leek Function/System Call:

Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written. By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified. An open file's offset can be set explicitly by calling lseek function. There is no actual I/O performed.

#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);
First argument in above function call is file descriptor, a handle to file opened. Second argument is the offset i.e. how many bytes to move either forward or backward. Third argument is from which position to move in the file. It means it will add offset to the whence.

If call is successful it will return new offset otherwise it will return -1 (error situation)

The interpretation of the offset depends on the value of the whence argument.

1. If whence is SEEK_SET, the file's offset is set to offset bytes from the beginning of the file.
2. If whence is SEEK_CUR, the file's offset is set to its current value plus the offset. The offset can be positive or negative.
3. If whence is SEEK_END, the file's offset is set to the size of the file plus the offset. The offset can be positive or negative.

Solved Exercise 3:

Program to test whether standard input is capable of seeking or not.

```
#include <unistd.h>
#include<stdio.h>
int main(){
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
    printf("cannot seek\n");
    else printf("seek OK\n");
    exit(0);
}
```

Solved Exercise 4:

Write a program that prints a file backwards.

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

```
int main(int argc, char *argv[]) {
    int source, dest, n;
    char buf;
    int filesize;
    int i;

    if (argc != 3) {
        fprintf(stderr, "usage %s <source> <dest>", argv[0]);
        exit(-1);
    }

    //read permission for user on source
    if ((source = open(argv[1], 0400)) < 0) {
       fprintf(stderr, "can't open source");
        exit(-1);
    }
    //rwx permission for user on dest
    if ((dest = creat(argv[2], 0700)) < 0) {
        fprintf(stderr, "can't create dest");
        exit(-1);
    }

    //filesize is lastby +offset
    filesize = lseek(source, (off_t) 0, SEEK_END);
    printf("Source file size is %d\n", filesize);

    //read byte by byte from end
    for (i = filesize - 1; i >= 0; i--) {
          lseek(source, (off_t) i, SEEK_SET);
          n = read(source, &buf, 1);
          if (n != 1) {
               fprintf(stderr, "can't read 1 byte");
               exit(-1);
          }
          n = write(dest, &buf, 1);
          if (n != 1) {
               fprintf(stderr, "can't write 1 byte");
               exit(-1);
          }
     }
     write(STDOUT_FILENO, "DONE\n", 5);
     close(source);
     close(dest);
     return 0;
}
```

4. Duplicating File Descriptors:

UNIX provides the capability to have one open file descriptor available as two (or more) separate file descriptors: dup() call. It is also possible to take an open file descriptor and cause it to be available on a specific file unit number (that is already not in use): dup2() call.

An existing file descriptor is duplicated by either of the following functions.

dup function/System Call

```
#include <unistd.h>
int dup(int filedes);
```

filedes argument(old file descriptor) is the file descriptor to be duplicated. On successful execution it will return new file descriptor otherwise if error occurs it will return -1.

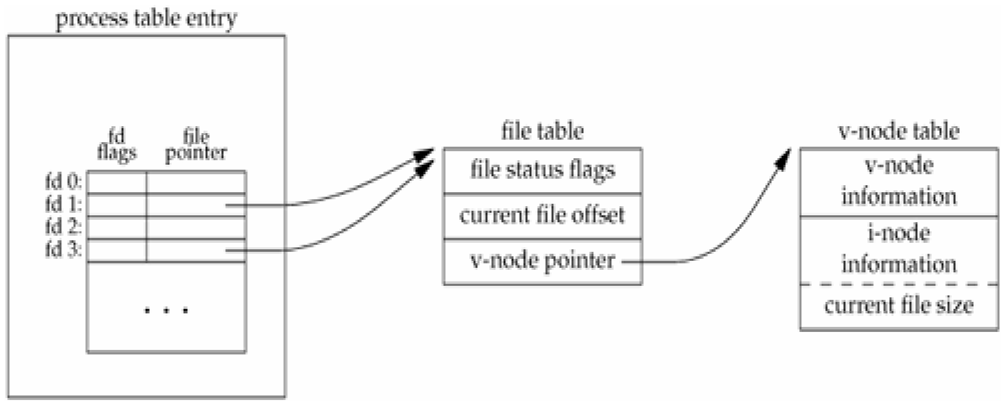
The new file descriptor returned by dup will be the lowest-numbered available file descriptor.

The new file descriptor that is returned as the value of the functions shares the same file table entry as the filedes argument (old file descriptor).

Solved Exercise 5:

If process executes newfd = dup(1), then how the kernel data structure look like.

Solution:



Solved Exercise 6

As The *dup()* call provides the programmer the flexibility to use more than one file descriptor on the same file, without having to worry about synchronizing file offsets every time. The following program demonstrates the same:

```
/* Using two open calls on the same file */

# include<string.h>
# include<stdio.h>
# include<fcntl.h>

int main(int argc,char **argv){
int fd1,fd2;

fd1=open("xyz.txt",O_RDWR);
fd2=open("xyz.txt",O_RDWR);

printf("fd1 file offset before write: %d\n",lseek(fd1,0,1));
write(fd1,"hello",strlen("hello"));
printf("fd2 file offset before write: %d\n",lseek(fd2,0,1));
write(fd2,"world",strlen("world"));

return 0;
}
```

The output for the above program would be:

```
$ fd1 file offset before write: 0
$ fd2 file offset before write: 0

$ cat xyz.txt
$ world
```

Note that the actual data “hello world” is not written successfully into the file because update of the file offset by fd1 is not reflected by fd2.This is because fd1 and fd2 each have a separate file description (and hence different copy of offset values).

Solved Exercise 7:

```
/* Using open followed by a dup call */

int main(int argc,char **argv){
int fd1,fd2;

fd1=open("xyz.txt",O_RDWR);
fd2=dup(fd1);

printf("fd1 file offset before write: %d\n",lseek(fd1,0,1));
write(fd1,"hello",strlen("hello"));
printf("fd2 file offset before write: %d\n",lseek(fd2,0,1));
write(fd2,"world",strlen("world"));
return 0;
}
```

This time the output would be:

```
$fd1 file offset before write: 0
$fd2 file offset before write: 5

$ cat xyz.txt
$ hello world
```

When a dup() call is used, the two file descriptors (fd1 and fd2) share the same file description and hence update by one is reflected in the other.

dup2 function/System Call

```
#include <unistd.h>
int dup2(int filedes, int filedes2);
```

With dup2, we specify the value of the new descriptor with the filedes2 argument. If filedes2 is already open, it is first closed. If filedes (old file descriptor) equals filedes2 (new file descriptor), then dup2 returns filedes2 without closing it.
The file descriptor returned by dup2() has the following in common with oldfd:

- Same open file
- Same file pointer (that is, both file descriptors share one file pointer)
- Same access mode (read, write or read/write)
- Same file status flags (refer to stat() system call)

dup2() is frequently used to re-direct file input\output. That is to reset the standard file input or output. The file descriptor STDIN_FILENO (standard input) does not always have to be the terminal (console). It can be made to refer to any other file (or device). The redirection is made simpler because dup2() automatically takes care of closing the old file descriptor.

6. Saving and Destroying Files

The *write()* call does not immediately write to the disk but instead only writes to the buffer cache. The kernel eventually writes all the delayed-write blocks to disk, normally when it needs to reuse the buffer for some other disk block. To ensure consistency of the file system on disk with the contents of the buffer cache, the sync, fsync, and fdatasync functions are provided.

sync function

The sync function simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place.

```
#include<unistd.h>
void sync(void);
```

sync() tells the kernel to flush the buffer cache. Kernel flushes the buffer cache at the earliest possible after the sync() call is issued. sync() is heavy handed in the sense that all the data in the buffer cache, written by so many different processes to so many different files get copied to the disk. Usually the sync() command is executed before the UNIX system is shutdown or a removable device is unmounted.

fsync function

The function fsync refers only to a single file, specified by the file descriptor filedes, and waits for the disk writes to complete before returning. The intended use of fsync is for an application, such as a database, that needs to be sure that the modified blocks have been written to the disk. A frequent call to sync() would affect other users too much. Because every time sync() is called, the entire contents of the buffer cache has to be copied into the disk and this is a very time consuming operation. A solution to this problem is to use the fsync() call.

```
#include<unistd.h>
int fsync(int fd); /* Returns 0 on error or -1 on error */
```

This function accepts a file descriptor as an argument and all cached changes only for that file is written out on the disk.

fdatasync function

The fdatasync function is similar to fsync, but it affects only the data portions of a file. With fsync, the file's attributes are also updated synchronously.

```
#include<unistd.h>
int fdatasync(int filedes);
```

7. Truncating files

The open and create calls when used with the appropriate flags can be used to truncate a file. However if a file needs to be truncated after it’s opened, the following calls could be used.

truncate and ftruncate System Calls

truncate call does not work on a open file. It truncates a file to a specified length without actually opening a file. ftruncate call works with open files.

```
# include<unistd.h>
int truncate(
  const char *path, /* Pathname */
  off_t length);    /* New Length */

int ftruncate(
   int filedes,     /* File Descriptor */
   off_t length);   /* New Length */
```

Oddly enough, the truncate() and ftruncate() calls can also be used to expand the file, if the new length is greater than the original length of the file. The expanded portion of the file would be filled with bytes of zero.

Usage:

```
int x,y;
int fd=open(“./pqr/lmn/file”,O_RDWR);

x= truncate(“./abc/xyz/filename”,0);
y= ftruncate(fd,10);
```

Self Study Topics and Exercises

1. Scatter Read and Gather Write: readv() and writev() system calls
2. Ignoring the file offset : pread() and pwrite() system calls
3. File status: stat() and fstat() system calls
4. Write a C program using UNIX system calls that would mimic the “grep” command , i.e., print the lines of a file which contain a given search word. Wildcard characters need not be considered
5. Write a C program using UNIX system calls that would mimic the “head” and “tail” commands. The commands should be supported with the option “-n”
6. Write your own version of the scanf() function. You are allowed to change the existing signature of the scanf() function. But the semantics of scanf() should not change (obvious!)
7. Write a C program to Print the file flags of specified file descriptor.
8. Write a C program to Turn on one or more of the file status flags for a descriptor.
9. fcntl() function.
10. Write a program to create a hole in a file using lseek function.
11. umask function.