UNIX IO

ITER, SOA (Deemed to be University), Bhubaneswar
By
Dr. TRILOK NATH PANDEY
DEPT. OF C.S.E

Text Books

Kay A. Robbins, & Steve Robbins

Unix™ Systems Programming

Communications, concurrency, and Threads

Pearson Education

(Chapter 4)

Unix I/O:

- Unix uses a uniform device interface, through file descriptors, that allows the same I/O calls to be used for terminals, disks, tapes, audio and even network communication.

- There are basically five system calls used for I/O operations: **open, close, read, write and creat**

open System Call

- The open function associates a file descriptor with a file or physical device.
- SYNOPSIS:

```
#include <fcntl.h>
#include <sys/stat.h>

int open(const char *path, int oflag, ...);

(1) If successful, open returns a nonnegative integer
    representing the open file descriptor.
(2) If unsuccessful, open returns -1 and sets errno.
```

- The **path** parameter of open points to the pathname of the file or device.
- The **oflag** parameter specifies status flags and access modes for the opened file.
- A third parameter must be included to specify access permissions if a file is created.

open() Examples

The following code segment opens a file my.dat for reading which belongs to the present working directory.

```
#include <fcntl.h>
#include <sys/stat.h>

int fd1;
fd1 = open("my.dat", O_RDONLY);
```

Modify the above code segment to open the file Users/jogs/ch4/file1.txt for reading

```
Perform OR the O_RDONLY and the O_CREAT flags.

myfd = open("Users/jogs/ch4/file1.txt", O_RDONLY
    |  O_CREAT);
```

O_CREAT is an additional flag which creates the file if the file does exist .

`close()` System Call

The close function has a single parameter, **`fildes`**, representing the open file whose resources are to be released.

```
#include <unistd.h>

int close(int fildes);

(1) If successful, close returns 0.
(2) If unsuccessful, close returns -1 and sets errno.
```

`read` System Call

- UNIX provides sequential access to files and other devices through the **`read`** and **`write`** functions.
- The **`read`** function attempts to retrieve **`nbyte`** bytes from the file or device represented by **`fildes`** into the user variable **`buf`**.
- A large enough buffer must be provided to hold **`nbyte`** bytes of data.
- SYNOPSIS:

```
#include <unistd.h>

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


(1) If successful, read returns the number of bytes
    actually read.
(2) If unsuccessful, read returns -1 and sets errno.
```

- The **`ssize_t`** data type is a signed integer data type used for the number of bytes read, or -1 if an error occurs.
- The **`size_t`** is an unsigned integer data type for the number of bytes to read.

Note: `Read`

- A **`read`** operation for a regular file may return fewer bytes than requested if, for example, it reached end-of-file before completely satisfying the request.
- A **`read`** operation for a regular file returns 0 to indicate end-of-file.
- When reading from a terminal, read returns 0 when the user enters an end-of-file character(CTRL+D).

`write` System Call

- The **`write`** function attempts to output **`nbyte`** bytes from the user buffer **`buf`** to the file represented by file descriptor **`fildes`**.
- SYNOPSIS:

```
#include <unistd.h>

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


(1) If successful, write returns the number of bytes
    actually written.
(2) If unsuccessful, write returns -1 and sets errno.
```

File Descriptor

- Files are designated with in C program either by **file pointers** or by **file descriptor**.
- A file **descriptor** is an integer value that represents a file or device that is open.
- It is an index into the process file descriptor table.
- The file descriptor table is in the process user area and provides access to the system information for the associated file or device.
- File pointers and file descriptors provide logical designations called *handles* for performing device-independent input and output.

File Descriptor

- The symbolic names for the **file pointers** that represent standard input, standard output and standard error are **`stdin`**, **`stdout`** and **`stderr`**, respectively. These symbolic names are defined in **`stdio.h`**.

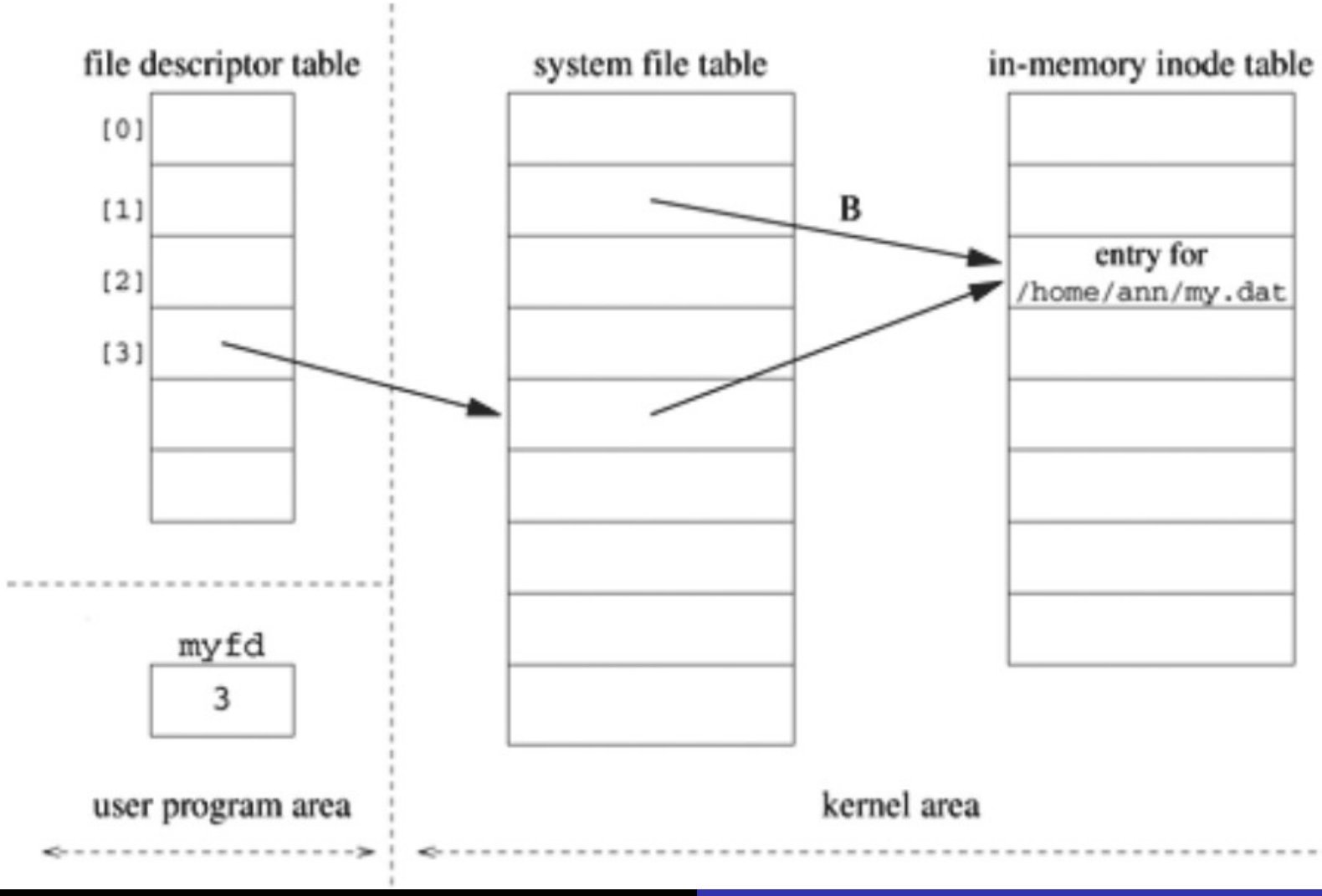
- The symbolic names for the **file descriptors** that represent standard input, standard output and standard error are **`STDIN_FILENO`**, **`STDOUT_FILENO`** and **`STDERR_FILENO`**, respectively. These symbolic names are defined in **`unistd.h`**.

- The **numeric values** that represent standard input, standard output and standard error are 0, 1, and 2 respectively.

File Descriptor

A schematic of the file descriptor table after a program executes the following.

```
myfd = open("/home/ann/my.dat", O_RDONLY);
```



Examples of read() and write() system calls

The following program will read maximum 10 characters from standard input (e.g. keyboard) and write to standard output (e.g. display screen)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <fcntl.h>
int main()
{
        char buf[100];
        ssize_t bytesread;
        bytesread=read(STDIN_FILENO, buf, 10);
        write(STDOUT_FILENO, buf, bytesread);
return 0;
}
```

Examples of read() and write() system calls cont…

Modify the previous program to read from a regular file (e.g. file1,txt) and write to standard output (e.g. display screen)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <fcntl.h>
int main()
{
      int fd1;
      char buf[10];
      ssize_t bytesread;
      fd1=open("file1.txt",O_RDONLY);
      bytesread=read(fd1, buf, 10);
      write(STDOUT_FILENO, buf, bytesread);
      close(fd1);
return 0;
}
```

Examples of read() and write() system calls Cont…

Write a program in C (copyfile.c) to copy the content of a regular file (file1.txt) to another regular file (file2.txt). Display the number of bytes read as well as number of bytes written.

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#define BLKSIZE 100
int main()
{
        int fd1,fd2;
        char buf[BLKSIZE];
        ssize_t bytesread,byteswrite;
        fd1=open("file1.txt",O_RDWR|O_CREAT, S_IRWXU);
        if(fd1==-1)
        {
        perror("Unable to open the file1.txt\n");
        return 0;
        }
        //printf("File Descriptor1=%d\n",fd1);
        bytesread=read(fd1,buf,BLKSIZE);
        close(fd1);
        fd2=open("file2.txt",O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
        if(fd2==-1)
        {
        perror("Unable to open the file file2.txt\n");
        return 0;
        }
        //printf("File Descriptor2=%d\n",fd2);
        byteswrite=write(fd2,buf,bytesread);
        printf("bytesread=%d\nbyteswrite=%d\n",(int)bytesread,(int)byteswrite);
return 0;
}
```

Examples of read() and write() system calls Cont…

Create a C file copyfile1.c by modifying the previous program and make an executable file (copy). Do the necessary steps to run it by name only like other unix commands:
Like $ copy file1 file2 where file1 and file2 are source and destination files respectively.

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#define BLKSIZE 100
int main(int argc,char *argv[])
{
        int fd1,fd2;
        char buf[BLKSIZE];
        ssize_t bytesread,byteswrite;
        if(argc<3) {
        printf("Please provide source and destination files\n");
        return 0;
        }
        fd1=open(argv[1],O_RDWR|O_CREAT, S_IRWXU);
        if(fd1==-1) {
        perror("Unable to open the source file\n");
        return 0;
        }
        bytesread=read(fd1,buf,BLKSIZE);
        close(fd1);
        fd2=open(argv[2],O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
        if(fd2==-1)
        {
        perror("Unable to open the file destination file\n");
        return 0;
        }
        byteswrite=write(fd2,buf,bytesread);
        if( byteswrite<=0)
        printf("0 bytes copied\n");
return 0;
}
```

Examples of read() and write() system calls Cont…

Step 1: make executable file (copy) of the program copyfile.c

$ gcc copyfile1.c -o copy

Step 2: Include the absolute path of the file “copy” into the values of environment variable PATH

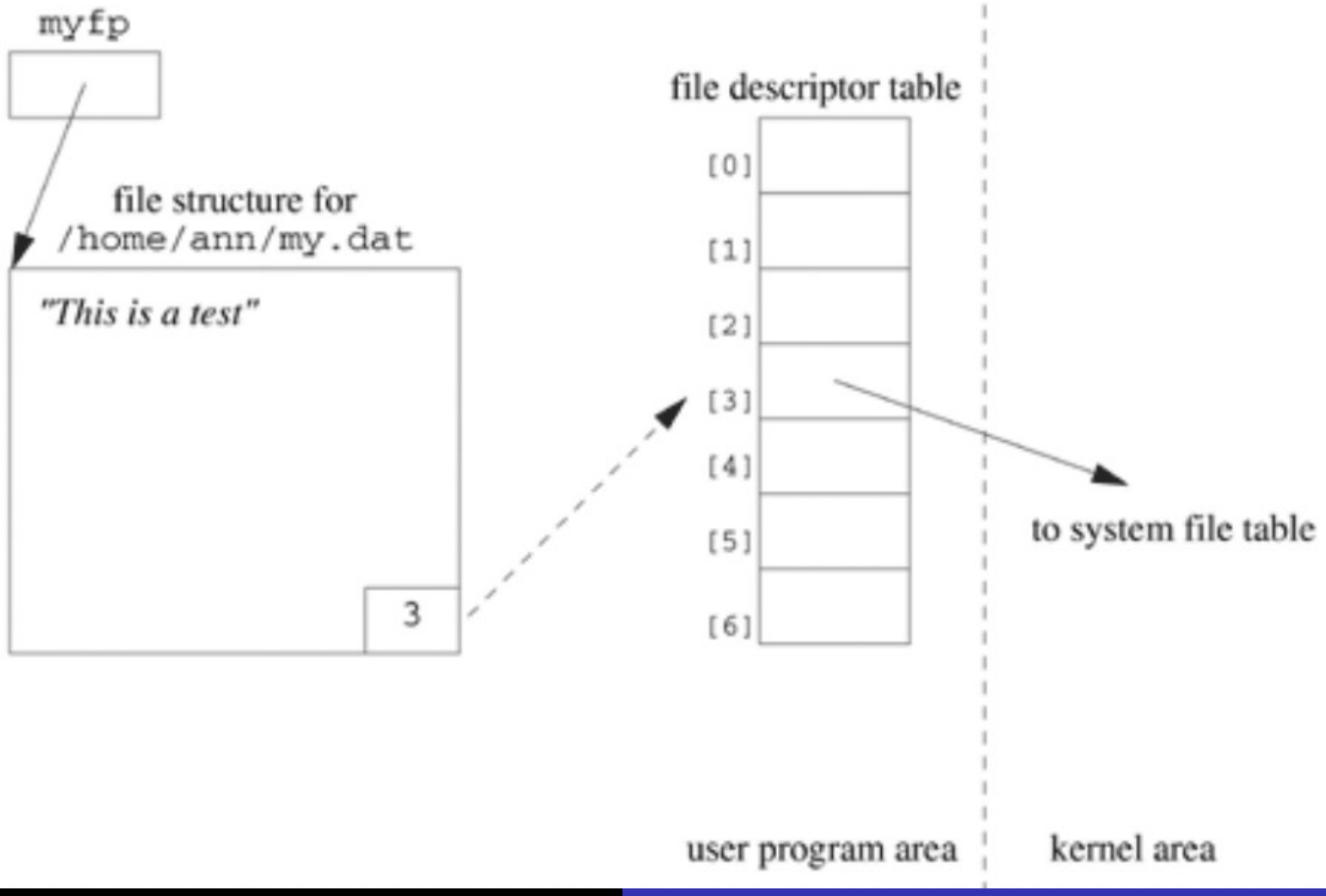
export PATH=$PATH:/Users/jogs/ch4/Lec3

Step 3: run the executable file ‘copy’

$ copy file1.txt file2.txt

File Pointer

```
FILE *myfp;
if ((myfp = fopen("/home/ann/my.dat", "w")) == NULL)
perror("Failed to open /home/ann/my.dat");
else
fprintf(myfp, "This is a test");
```



File descriptor's value from file pointer

```
#include<stdio.h>
int main()
{
    FILE *myfp;
    int fd;
    myfp=fopen("Trial.txt","w");
    if(myfp==NULL)
      {
        perror("Opening Error”);
        return 1;
       }
/* To get the file descriptor value */
    fd=fileno(myfp); /* fileno() is a library function */
    printf("File descriptor=%d\n",fd);
return 0;
}
```

`oflag` Argument Setting

- The POSIX values for the **access mode flags** in `oflag` argument:
  1. `O_RDONLY` : read-only access
  2. `O_WRONLY` : write-only access
  3. `O_RDWR` : read-write-only access

  Note: specify exactly one of above designating read-only, write-only or read-write access.

- The `oflag` argument is also constructed by taking the bitwise OR (|) of the desired combination of the access mode and the **additional flags**.

- The additional flags:
  1. `O_APPEND`
  2. `O_CREAT`
  3. `O_EXCL`
  4. `O_NOCTTY`
  5. `O_NONBLOCK`
  6. `O_TRUNC`

Additional Flags

`O_APPEND`: The O_APPEND flag causes the file offset to be moved to the end of the file before a write, allowing you to add to an existing file.

`O_CREAT`: The O_CREAT flag causes a file to be created if it doesn't already exist. If O_CREAT flag is included, a third argument to **`open()`** must be passed to designate the permissions.

`O_EXCL`: If you want to avoid writing over an existing file, use the combination O_CREAT | O_EXCL. This combination returns an error if the file already exists.

`O_NOCTTY`: The O_NOCTTY flag prevents an opened device from becoming a controlling terminal.

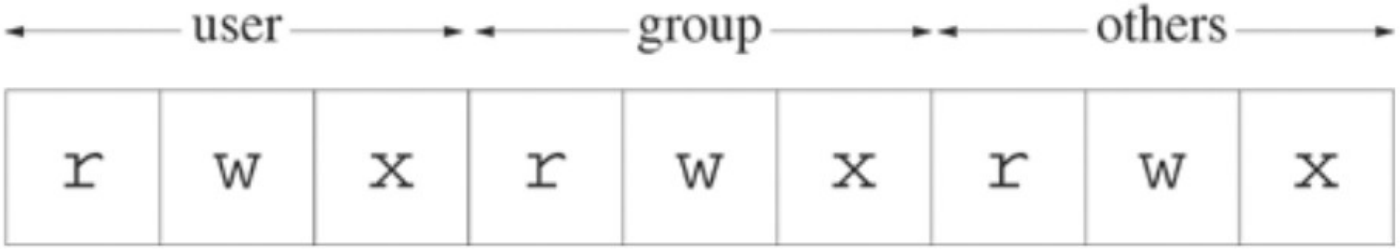
`O_NONBLOCK`: The O_NONBLOCK flag controls whether the **`open()`** returns immediately or blocks until the device is ready.

`O_TRUNC`: O_TRUNC truncates the length of a regular file opened for writing to 0.

Third argument to `open()`

- Each file has three classes associated with it: a user (or owner), a group and everybody else (others).
- The possible permissions or privileges are read(r), write(w) and execute(x). These privileges are specified separately for the user, the group and others.
- When you open a file with the O_CREAT flag, you must specify the permissions as the third argument to open in a mask of type mode_t.
- POSIX defines symbolic names for masks corresponding to the permission bits. These names are defined in **sys/ stat.h**

POSIX symbolic names for file permissions



Symbol	meaning
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute by owner
S_IRWXU	read, write, execute by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXG	read, write, execute by group
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute by others
S_IRWXO	read, write, execute by others
S_ISUID	set user ID on execution
S_ISGID	set group ID on execution

Filters

- A filter reads from standard input, performs a transformation, and outputs the result to standard output.
- Filters write their error messages to standard error.
- **Examples:** `head`, `tail`, `more`, `sort`, `grep`, `sed`, and `awk` etc.
- **`cat`** as a filter: The cat command takes a list of filenames as command-line arguments, reads each of the files in succession, and echoes the contents of each file to standard output. However, if no input file is specified, cat takes its input from standard input and writes its results to standard output. In this case, cat behaves like a filter.

Redirection

- a file descriptor is an index into the file descriptor table of that process.
- Each entry in the file descriptor table points to an entry in the system file table, which is created when the file is opened.
- A program can modify the file descriptor table entry so that it points to a different entry in the system file table. This action is known as **redirection**.
- Most shells interpret the greater than character (>) on the command line as redirection of standard output and the less than character (<) as redirection of standard input.
- **Example:** `$ cat > myfile.txt`, redirects standard output to `myfile.txt` with >.

dup() & dup2(): Duplicate a File Descriptor

```
#include <unistd.h>

int dup(int fildes);
```

Return:

(1) dup() creates a copy of the file descriptor fildes.
(2) dup() uses the lowest-numbered unused descriptor for the new descriptor returned
(3) -1 on error

```
#include <unistd.h>

 fd=open("read.c",O_RDONLY);
 nfd=dup(fd);          /* duplicates the file descriptor fd */
 printf("Duplicate fd=%d\n",nfd);
```

nfd is the lowest-numbered unused file descriptor. It is the duplicate of **fd**.

Example: dup()

Duplicate the standard output file descriptor to write onto a file descriptor **fd**

```
/*duplicating STDOUT_FILENO to a file descriptor fd */
int main()
{
 int fd;  fd=open("duptest.txt",O_WRONLY|O_CREAT|
 O_TRUNC,S_IRUSR|
     S_IWUSR|S_IRGRP);
 printf("File descriptor: fd=%d\n",fd);
 dup(STDOUT_FILENO);/* save descriptor STDOUT_FILENO */
 close(1);                /* closing 1, creates an empty slot */
 dup(fd);
 close(fd);               /* duplicate fd to standard output */
 write(STDOUT_FILENO,"USP\n",4);
 write(STDOUT_FILENO,"DOS\n",4);
 return 0;
}
```

Run the code, then open the file: **$ cat duptest.txt**. Data is now written onto the file instead of monitor

dup2 () : Duplicate a File Descriptor

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

Return:

(1) On success, dup2 returns the file descriptor value that was duplicated.
(2) -1 on error

Example: dup2 ()

Duplicate the standard output file descriptor to write onto a file descriptor **fd**

```
/*duplicating STDOUT_FILENO to a file descriptor fd */

#define CREATE_FLAGS (O_WRONLY | O_CREAT | O_APPEND)
#define CREATE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(void) {
    int fd;
    fd = open("dup2test.txt", CREATE_FLAGS, CREATE_MODE);
    if (dup2(fd, STDOUT_FILENO) == -1) {
        perror("Failed to redirect standard output");
        return 1;
    }
    close(fd);
    write(STDOUT_FILENO, "OK", 2);
    return 0;
}
```

Run the code, then open the file: **$ cat dup2test.txt**. Data is now written onto the file instead of monitor

dup() vs dup2()

```
dup2(fd1,fd2);
```

Is equivalent to

```
close(fd2);
dup(fd1);
```

Thank You