CS 332/532 Systems Programming

Lecture 9

-UNIX Files-

Professor : Mahmut Unan – UAB CS

Agenda

- UNIX File System
- Unix vs Windows
- File Types
- File Descriptors
- File Permissions
- Open/Close/read/write a file

UNIX Files System

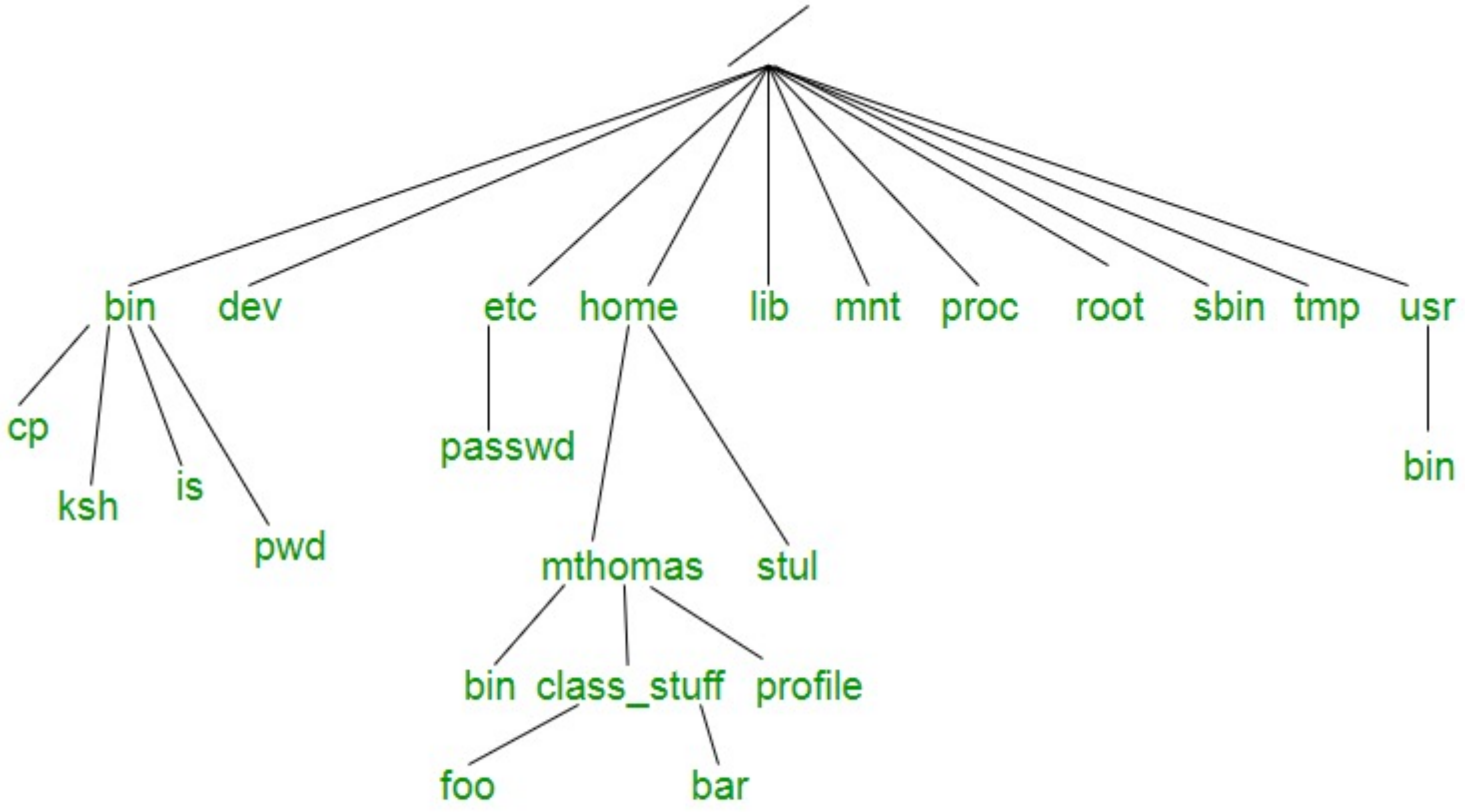
- All data organized into files
- All files organized into directories
- Directories
  - Tree-like structures
  - Multi-level hierarchy (directory tree)
  - Top level - root
    - /
    - All other directories are the children of the root
- File = sequence of bytes

Windows vs Unix

- Windows files are stored in folders on different data drives
  - C: D: E: ….
- Unix files are ordered in a tree structure
  - root and the children
- Peripherals such as hard drives, CD-Rom, printers, scanners
  - Windows consider them as devices
  - Unix consider them as files
- Naming convention
  - Windows → UAB and uab are the same, can't be under the same folder
  - Unix → they are different; UAB, uab, Uab, uAb…

Unix Directory Structure

- A file system consists of files, relationships to other files, as well as the attributes of each file
- root contains other files and directories
- each file or directory
  - uniquely identified by;
    - name
    - the directory that contains the file/directory
    - unique identifier (inode)
  - includes;
    - file type, size, owner, protection/privacy. time stamp
- each file is self contained



picture: https://www.geeksforgeeks.org/unix-file-system/

Listing the names of all files

```
#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can’t open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Figure 1.3 List all the files in a directory

File Types

- Directory Files
- Ordinary Files
  - text
  - data
  - program instruction….
  - …
- Special Files
  - devices
  - shortcuts
  - …

Unix File Descriptors

- It is a non-negative integer number that uniquely identifies an open file in Unix.
- It describes a data resource, and how that resource may be accessed.
- The first three file descriptors;
  - STDIN (standard input). 0
  - STDOUT (standard output). 1
  - STDERR (standard error). 2

File Permissions

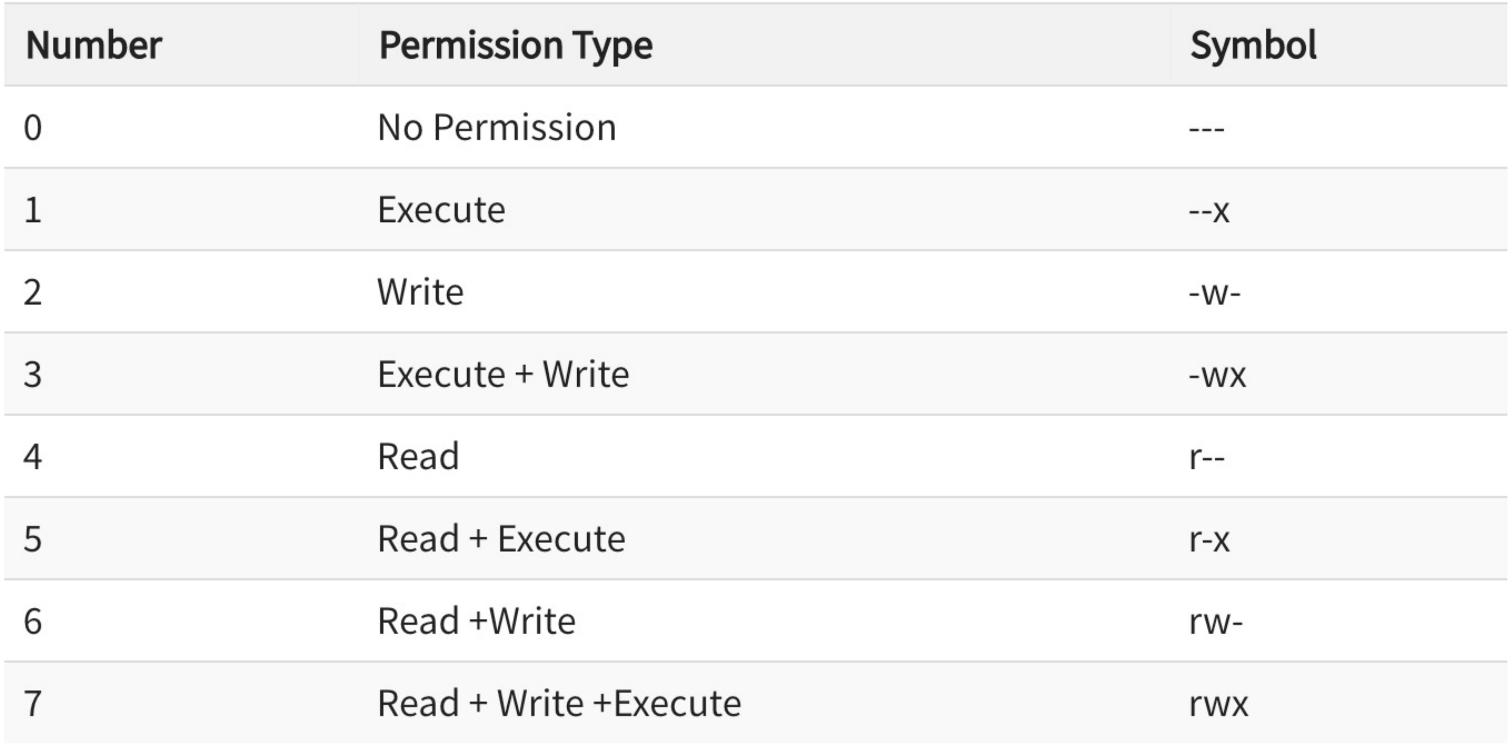
- use `ls -l` command to display the permissions
  - r w x - → read write execute no permission
- Owners permission -> First three characters
- Group permission -> next three characters
- World (other) Permission -> last three characters
- For example;

  -rwxrw-r--

  drwxr-xr--

Change Permission

- Use the `chmod` command to set permissions (read, write, execute) on a file/directory for the owner, group and the world



Number	Permission Type	Symbol
0	No Permission	---
1	Execute	--x
2	Write	-w-
3	Execute + Write	-wx
4	Read	r--
5	Read + Execute	r-x
6	Read +Write	rw-
7	Read + Write +Execute	rwx

https://www.guru99.com/file-permissions.html

UNIX file I/O Functionalities

- Processes needs system calls to handle file operation
- Opening a file
  - open or openat functions

```
fd = open(path,flag,mode)
```

https://man7.org/linux/man-pages/man2/open.2.html

flag

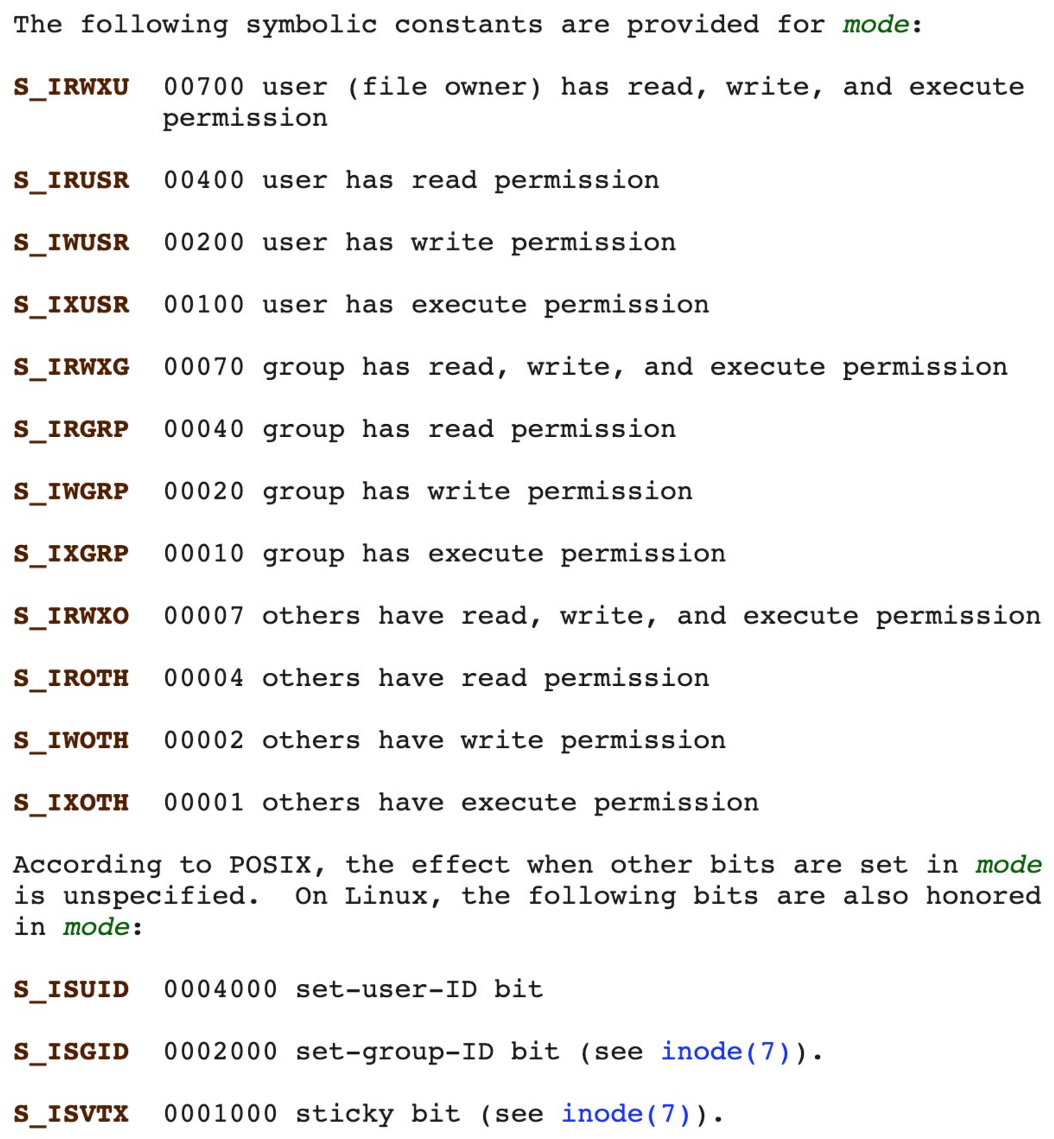
```
O_RDONLY
O_WRONLY
O_RDWR
O_EXEC
O_APPEND
O_CREAT
```

……

……

mode

- The third argument → *mode* specifies the permissions to use in case a new file is created.



The following symbolic constants are provided for *mode*:

S_IRWXU	00700 user (file owner) has read, write, and execute permission
S_IRUSR	00400 user has read permission
S_IWUSR	00200 user has write permission
S_IXUSR	00100 user 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

According to POSIX, the effect when other bits are set in *mode* is unspecified. On Linux, the following bits are also honored in *mode*:

S_ISUID	0004000 set-user-ID bit
S_ISGID	0002000 set-group-ID bit (see inode(7)).
S_ISVTX	0001000 sticky bit (see inode(7)).

https://man7.org/linux/man-pages/man2/open.2.htm
l

close

- system call
- the file descriptor is returned to the pool of available descriptors

```
int close(int fd);
```

- `close()` returns zero on success. On error, -1 is returned, and errno is set appropriately.

read()

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

- **read**() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

- On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.

https://man7.org/linux/man-pages/man2/read.2.html

write()

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

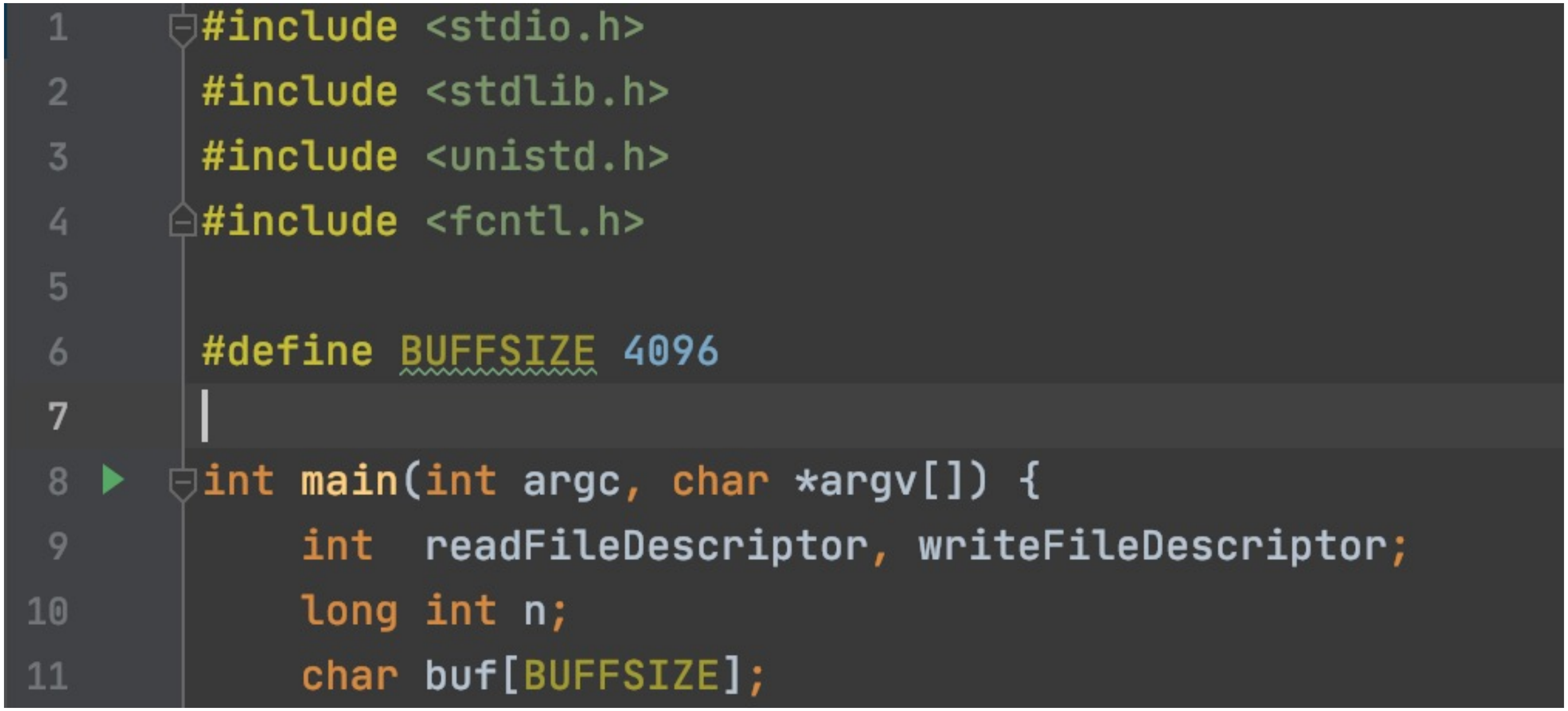
- **write**() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

- On success, the number of bytes written is returned. On error, -1 is returned, and *errno* is set to indicate the cause of the error.

https://www.man7.org/linux/man-pages/man2/write.2.html

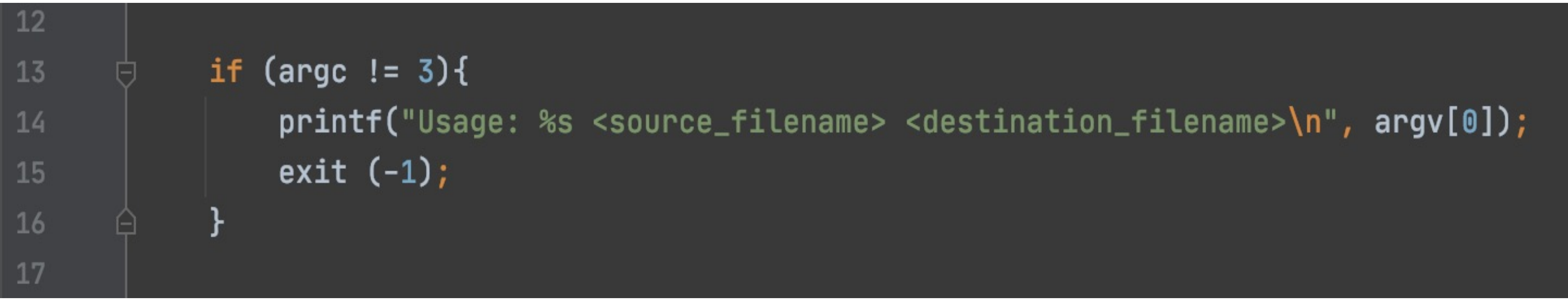
Exercise

- C code to copy one file and copy the contents of that file to a new file



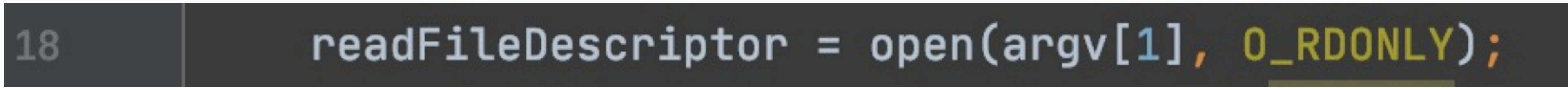
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFSIZE 4096

int main(int argc, char *argv[]) {
    int  readFileDescriptor, writeFileDescriptor;
    long int n;
    char buf[BUFFSIZE];
```

- Check if the correct numbers of the argument are given. There should be three arguments: name of the program, input file name, and output file name. If the number of arguments is not three then the program should print an error message and terminate. Also, input and output file names should not be the same.

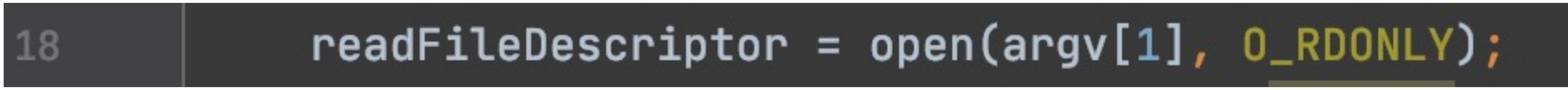


```c

    if (argc != 3){
        printf("Usage: %s <source_filename> <destination_filename>\n", argv[0]);
        exit (-1);
    }

```

- Use the *open* function in *read only mode* to read the input file.
- The open function takes the name of the file as the first argument and the open flag as the second argument.
- The open flag specifies if the file should be opened in read only mode (O_RDONLY), write only mode (O_WRONLY), or read-write mode (O_RDWR).
- There is an optional third argument that specifies the file permissions called *mode*, we will not use the optional third argument here. The third argument specifies the file permissions of the new file created for writing. Note that the UNIX file uses {*read, write, execute*} (*rwx)* permissions for the user, group, and everyone



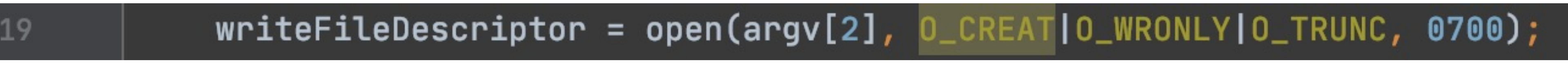
```
        readFileDescriptor = open(argv[1], O_RDONLY);
```



```
readFileDescriptor = open(argv[1], O_RDONLY);
```

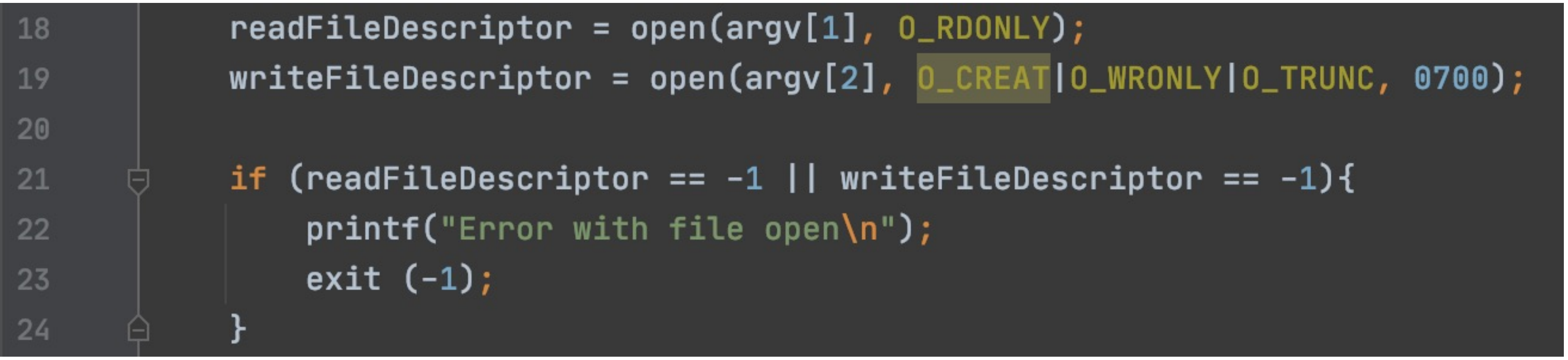
- The function returns a file descriptor which is typically a non-negative integer.
- If there is an error opening the file, the function returns -1.
- Note that most programs have access to three standard file descriptors: standard input (stdin) - 0, standard output (stdout) - 1, and standard error (stderr) - 2.
- Instead of using the values 0, 1, and 2 we can also use the POSIX name STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO, respectively.

- Use the open function to create a new write for writing the output such that if the file does not exist it will create a new file and if a file with the given name exists it will overwrite the existing file.
- This is accomplished by ORing the different open flags: O_CREAT, O_WRONLY, and O_TRUNC. O_CREAT specifies to open a new empty file if the file does not exist and requires the third file permission argument to be provided.
- O_WRONLY specifies that the file should be open for writing only and the O_TRUNC flag specifies that if the file already exists, then it must be truncated to zero length (i.e., destroy any previous data).



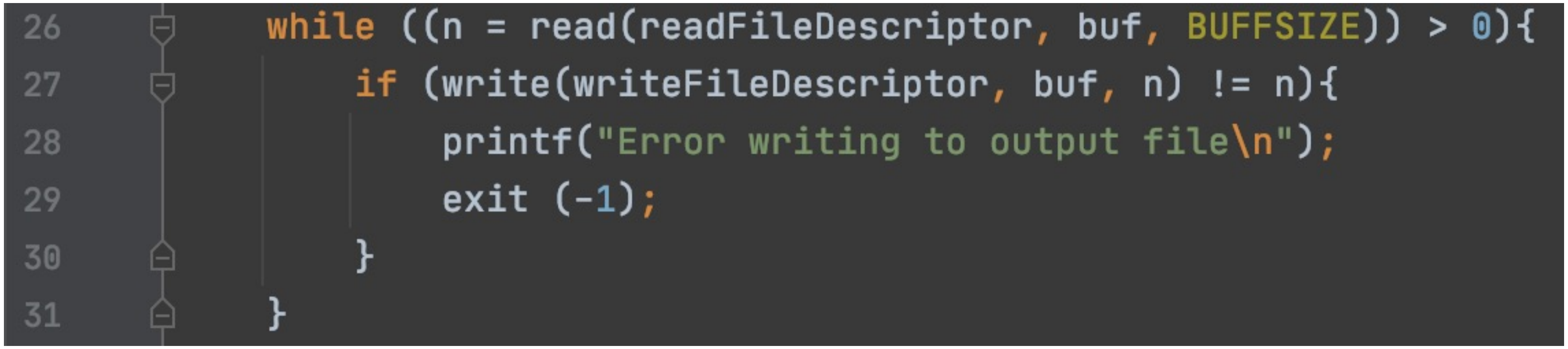
```
    writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);
```

- Check the file descriptor to see if there is a problem or not



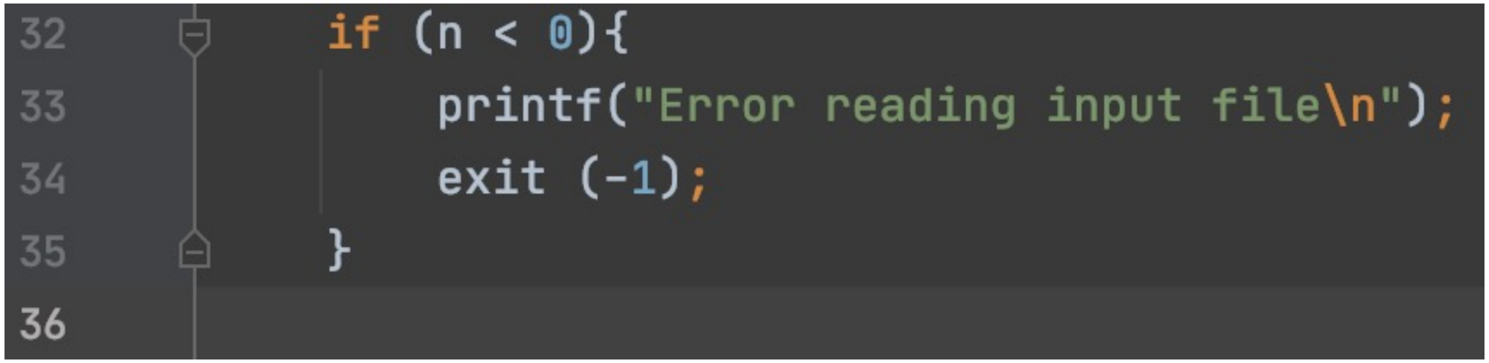
```c
    readFileDescriptor = open(argv[1], O_RDONLY);
    writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);

    if (readFileDescriptor == -1 || writeFileDescriptor == -1){
        printf("Error with file open\n");
        exit (-1);
    }
```

- Now read the file by reading fixed chunks of data using the *read* function by providing the file descriptor, input buffer address, and the maximum size of the buffer provided .
- A successful read returns the number of bytes read, 0 if the end-of-file is reached, or -1 if there is an error.
- After you read the data in to the buffer write the buffer to the new file using the *write* function.
- The *write* function takes the file descriptor, buffer to write, and the number of bytes to write. If the write is successful, it will return the number of bytes actually written.



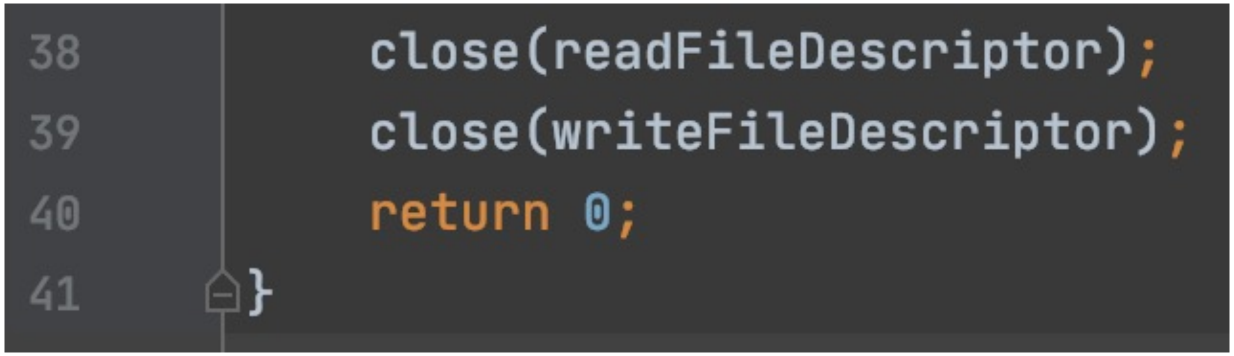
```
while ((n = read(readFileDescriptor, buf, BUFFSIZE)) > 0){
    if (write(writeFileDescriptor, buf, n) != n){
        printf("Error writing to output file\n");
        exit (-1);
    }
}
```

- check the error condition



```
    if (n < 0){
        printf("Error reading input file\n");
        exit (-1);
    }

```

- After completing the copy process use the *close* function to close both file descriptors. The close function takes the file descriptor as the argument and returns 0 on success and -1 in case of an error.



```
    close(readFileDescriptor);
    close(writeFileDescriptor);
    return 0;
}
```