Program- B. Tech
Semester- 3rd
Course Code- CSET213
Year- 2025
Date- 14/11/2025

Type- Sp. Core-I
Course Name-Linux and Shell Programming
Semester- Odd
Batch- Cyber Security

Lab Assignment 14

Exp No	Name	CO1	CO2	CO3	CO4
14	File handling commands and API	✓	✓		

Objective: Understand file handling APIs in GNU/Linux and explore several applications to demonstrate the proper use of the file handling APIs. To explore the character and string access mechanisms. Investigate both sequential and non-sequential (random access) methods. Review alternate APIs and methods for file access.

Outcomes: After learning and hands-on, the students will be able to create their own Linux command that can be used by other users.

Hands-on Learning on file handling with GNU/Linux (60 minutes)

- **Introduction**
  - Is accomplished through the standard C Library.
  - We can create and manipulate ASCII text or binary files with the same API.
  - We can append to files or seek within them.
  - Useful calls and functions
    - fopen to open or create a file.
    - fwrite and fread functions to write to or read from a file.
    - fseek to position at a given position in an existing file.
    - feof call to test whether we are at the end of a file while reading and
    - some other low-level calls such as open, write and read.
- **File Handling API – Creating a File Handle**
  - The first step is to make visible the file I/O APIs.
  - Done by including the stdio.h header file:

```
#include <stdio.h>
```

  - The next step is to declare a handle to be used in file I/O operations.
  - Often called a file pointer and is a transparent structure that should not be accessed by the developer.
  -

```
FILE *my_fp;
```

- **File Handling API – Opening a File**
  - Opening a file can also be the mechanism to create a file.

- The fopen function is simple and provides the API:

```
FILE * fopen (const char *filename, const char *mode);
```

  - First argument is filename and then the mode we wish to use.
  - The result is a FILE pointer, which could be NULL, indicating an operation failure.
  - The key to the call is the mode.
  - The mode is a string that fopen call uses to determine how to open (or create) the file.
- If we wanted to create a new file, we could simply use:
  -

```
my_fp = fopen ("myfile.txt", "w");
```

  - The result would be the creation of the file for write operations.
  - To open a file to reading we could use:
  -

```
my_fp = fopen ("myfile.txt", "r");
```

  - The read mode assumes that the file exists, and if not, a NULL is returned.

Mode	Description
r	Open an existing file for read
w	Open a file for write (create new if exists)
a	Open a file for append (create if file doesn't exist)
rw	Open for read and write (create if it doesn't exist)

- **Example: Create a New file**
  - In both cases, it is assumed that our file myfile.txt will either exist or be created in the current working directory.
  - The results of all file I/O operations should be checked for success.
  - For the fopen call, we simply test the response for NULL.
  - What happens upon error is ultimately application dependent.

test.c

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>

#define MYFILE "missing.txt"

int main()
{
FILE *fin;
/* Try to open the file for read */
fin =  fopen (MYFILE, "r");
/* Check for failure to open */
if ( fin == (FILE *) NULL)
{
printf ( "%s:  %s\n", MYFILE, strerror( errno ) );
exit( EXIT_FAILURE);
}
/* All was well, close the file */
(void) fclose(fin);
return 0;
```

- After trying to open the file highlighted in red color, we check to see if our new file handle is NULL (zero) highlighted in blue color.
- In case of error, we display an error message that consists of the filename and then the error message that resulted.
- We capture the error number with errno. errno is a special variable set by system calls.
- Executing our application:

```
# gcc test.c -o test
# ./test
missing.txt: No such file or directory
```

- **Reading and Writing Data**
  - We could read or write on a character basis or on a string basis (ASCII text only).
  - We could also use a more general API that permits reading and writing records, supporting ASCII and binary representations.
  - The standard I/O library presents a buffered interface.
  - This has two very important properties.
    - System reads and writes are in blocks, written to the FILE buffer. The buffer is written to the media automatically when it's full.
    - Second, fflush is necessary, or nonbuffered I/O must be set if the data is being sent to an interactive device such as the console terminal.
- **Character Interfaces**
  - The prototypes for character oriented functions:

```
int fputc (int c, FILE *stream);
```

```
int getc (FILE *stream);
```

  - **The fputc example:**

charout.c

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
      int i;
      FILE *fout;
      const char string[] = { "This \r\n is a test
\r\nfile.\r\n\0"};
      fout = fopen("inpfile.txt", "w");
      if (fout == (FILE *)NULL)
            exit(EXIT_FAILURE);
      i=0;
      while (string[i] != '\0')
      {
            (void) fputc( (int)string[i], fout  );
            ++i;
      }
      (void) fclose(fout );
      return 0;
}
```

    - Our loop walks through the entire string until a NULL is detected which is highlighted in blue color, at which point we exit and close the file highlighted in red color.

- We used fputc to use our character(as an int, based on fputc prototype) as well as specifying our output stream(fout).

- **The fgetc Example:**

charin.c

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
      int c;
      FILE *fin;
      fin = fopen("inpfile.txt", "r");
      if (fin == (FILE *)NULL)
            exit(EXIT_FAILURE);
      do
      {
            c = fgetc(fin );
            if ( c != EOF )
            {
                  printf("%c", (char) c);
            }
      }while (c != EOF);

      (void) fclose(fin );
      return 0;
}
```

- **Executing our applications:**

```
gcc charout -o charout.c
gcc charin -o charin.c

./charout
./charin
```

- **String Interfaces**
  - There are four library functions which are used for string interfaces.
    - The fputs and fgets are simple string interfaces, and
    - The fprintf and fscanf are more complex and provide additional capabilities.
  - The fputs and fgets provide the means to write and read variable length strings to files.
  - Prototypes for the fputs and fgets are:

```
int fputs (int c, FILE *stream);
char * fgets ( char *s, int size, FILE *stream);
```

  - **Writing variable length strings to a file**

strout.c

```
#include<stdio.h>
#include<stdlib.h>
#define LEN 80

int main()
{
      char line[LEN +1];
      FILE *fout;
      FILE *fin;
      fout = fopen("testfile.txt", "w");
      if (fout == (FILE *)NULL)
            exit(EXIT_FAILURE);
      fin = fdopen(0, "r");
      /* The NULL appears when you clash the descriptor,
      which is achieved through pressing ctrl+D at the
      keyboard*/
      while ((fgets(line, LEN, fin)) != NULL )
      {
            (void)fputs(line, fout);
      }

      (void) fclose(fin);
      (void) fclose(fout);

      return 0;
}
```

- We opened output file testfile.txt highlighted in blue color.
- We checked the error status highlighted in red color.
- We used fdopen to associate an existing file descriptor with a stream highlighted in green color.
- Whatever we now type in (standard-in) will be routed to this file stream.
- We enter a loop highlighted in yellow color that attempts to read from the fin stream and write this out to the output stream (fout).
- The NULL will appear when we close the descriptor (Ctrl+D).

- **Reading with fgets**

strin.c

```
#include<stdio.h>
#include<stdlib.h>
#define LEN 80
int main()
{
      char line[LEN +1];
      FILE *fin;
      fin = fopen("testfile.txt", "r");
      if (fin == (FILE *)NULL)
            exit(EXIT_FAILURE);
      while ((fgets(line, LEN, fin)) != NULL )
      {
            printf("%s", line);
      }
      (void) fclose(fin);
      return 0;
}
```

- We opened our input file and created a new input stream handle called fin.
- We used this at line, highlighted in yellow color, to read variable length strings from the file.

structout.c

```
#include<stdio.h>
#include<stdlib.h>
#include "mytype.h"
#define FILENAME "myfile.txt"
#define MAX_OBJECTS 3
typedef struct {
      int id;
      float x_coord;
      float y_coord;
      char_name[MAX_LINE+1];
} MY_TYPE_T;
MY_TYPE_T objects [MAX_OBJECTS] = {
      {0, 1.5, 8.4, "First-object"},
      {1, 9.2, 7.4, "Second-object"},
      {2, 4.1, 5.6, "Final-object"}
};
int main ()
{
      int i;
      FILE *fout;
      fout = fopen(FILNAME, "w");
      if (fout == (FILE *)NULL)
            exit(EXIT_FAILURE);
      for ( i=0; i < MAX_OBJECTS; i++  )
      {
            fprintf(fout, "%d %f %f %s\n",
            objects[i].id,
            objects[i].x_coord,
            objects[i].y_coord,
            objects[i].name);
      }
      (void) fclose(fout);
      return 0;
}
```

- When done, we print it via printf at line highlighted in green color.
- What if our data is more structured than simply strings?
  - If our strings are made up of integers, floating-point values, or other types?
  - We can use another method to deal with them more easily.

- **Handling Structured Data**
  - Let's say that we want to store an integer item (an id), two floating point values (2d coordinates), and a string (an object name).
  - Here also we deal with strings. Using API, ability to translate to native data types is provided.
  - We created a test structure (highlighted in red color) to represent our data.
  - We initialized this structure (highlighted in yellow color) with data.
  - Highlighted green color represent opening and then checking fout and perform a for loop to send our data to the file using fprintf API function.
  - We are printing an int (%d), two floating point values (%f), and then finally a string (%s).
  - This converts all data to string format and writes it to the output file.
    - We could have achieved this with a sprint call.
    - The disadvantage is that local space must be declared for the string being emitted.
    - This would not be required with a call to fprintf directly.
  - The protypes for fprintf and sprint

```
int fprintf (FILE *stream, const char *format,….);
int sprintf (char *str, const char *format,….);
```

- **Reading Structured Data in ASCII Format**

structin.c

```
#include<stdio.h>
#include<stdlib.h>
#include "mytype.h"
#define FILENAME "myfile.txt"
int main ()
{

      FILE *fin;
      MY_TYPE_T object;
      fin = fopen(FILNAME, "r");
      if (fin == (FILE *)NULL)
            exit(EXIT_FAILURE);
      while ( !feof( fin ))
      {
            (void)fscanf(fin, "%d %f %f %s\n",
            &object.id,
            &object.x_coord,
            &object.y_coord,
            &object.name);

            printf("%d %f %f %s\n",
            object.id,
            object.x_coord,
            object.y_coord,
            object.name);

      }
      (void) fclose(fin);
      return 0;
}
```

- We could have achieved this with sscanf (to parse our input string).

```
char line [81];
….
fgets (fin, 80, line);
sscanf (line, 80, "%d %f %f %s \n",
      objects[i].id,
      objects[i].x_coord,
      objects[i].y_coord,
      objects[i].name);
```

- The disadvantage is that local space must be declared for the parse to be performed on the input string.
- This would not be required with a call to fscanf directly.
- The fscanf and sscanf function prototypes

```
int fscanf (FILE *stream, const char *format, …);
int sscanf (const char *str, const char *format, …);
```

- When I/O is complete, the file should be closed with fclose (or, flushed with fflush).

- **Reading and Writing Binary Data**

- The fwrite and fread functions provide the ability to deal not only with the I/O of objects, but also with array of objects.
- The prototypes of the fwrite and fread functions are:

```
size_t fread (void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- Using fwrite for structured data

binout.c

```
#include<stdio.h>
#include<stdlib.h>
#include "mytype.h"
#define FILENAME "myfile.bin"
#define MAX_OBJECTS 3
typedef struct {
      int id;
      float x_coord;
      float y_coord;
      char_name[MAX_LINE+1];
} MY_TYPE_T;
MY_TYPE_T objects [MAX_OBJECTS] = {
      {0, 1.5, 8.4, "First-object"},
      {1, 9.2, 7.4, "Second-object"},
      {2, 4.1, 5.6, "Final-object"}
};
int main ()
{
      int i;
      FILE *fout;
      fout = fopen(FILNAME, "w");
      if (fout == (FILE *)NULL)
            exit(EXIT_FAILURE);
      /* write out the entire objects structure */
      (void) fwrite(void *)objects, sizeof(MY_TYPE_T), 3, fout);
      (void) fclose(fout);
      return 0;
}
```

- We specify the object that we are emitting (variable object, passed as a void pointer) and then the size of a row in this structure (the type MY_TYPE_T) using the sizeof operator.
- We then specify the number of elements in our array of types (3) and finally the output stream to which we want this object to be written.
- After executing the binout executable, the file myfile.bin is generated.

- Inspecting the contents of the binary file

- Inspecting the contents of the generated binary file

```
$ ./binout
$ more myfile.bin
$ od -x myfile.bin
```

- Reading file using fread
  - Let's read the file in a nonsequential way (aka random access)
  - This requires the use of fseek and rewind.
  -

```
void rewind (FILE *stream);
int fseek (FILE *stream, long offset, int whence);
```

  - The rewind function simply resets the file read pointer back to the start of the file
  - The fseek function allows us to the new position given an index.
  - The whence argument defines whether the position is relative to the start (SEEK_SET) or the current (SEEK_CUR) position of the file.

- The lseek function operates like fseek, but instead on a file descriptor.

```
int lseek (FILE *stream, long offset, int whence);
```

- File relative positions

Name	Description
SEEK_SET	**Moves the file position to the position defined by offset.**
SEEK_CUR	**Moves the file position the number of bytes defined by offset from the current file position.**
SEEK_END	**Moves the file position to the number of bytes defined by offset from the end of the file.**

- **Using fread and fseek/rewind to read structured data**

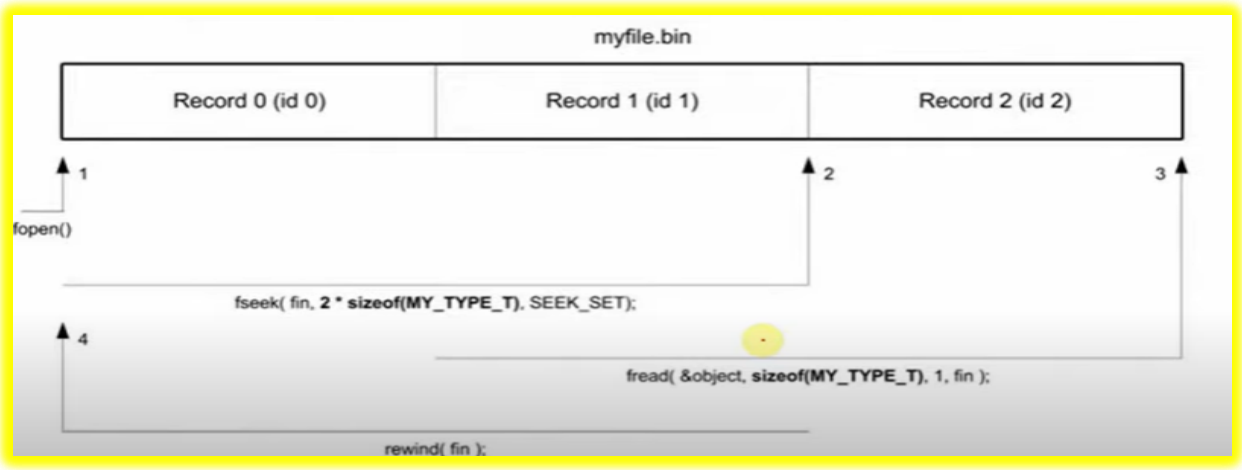
nonseq.c

```
#include<stdio.h>
#include<stdlib.h>
#include "mytype.h"
#define FILENAME "myfile.bin"
typedef struct {
        int id;
        float x_coord;
        float y_coord;
        char_name[MAX_LINE+1];
} MY_TYPE_T;
int main ()
{
        FILE *fin;
        long int offset;
        MY_TYPE_T object;
        fin = fopen(FILNAME, "r");
        if (fin == (FILE *)NULL)
                exit(EXIT_FAILURE);
        /* print the last entry */
        offset = 2L * (long) sizeof(My_TYPE_T);
        (void) fseek(fin, offset, SEEK_SET);
        (void) fread(&object, sizeof(MY_TYPE_T), 1, fin);
        printf("%d %f %f %s\n",
        object.id,
        object.x_coord,
        object.y_coord,
        object.name);
        /* print the second to last entry */
        offset = 1L * (long) sizeof(My_TYPE_T);
        rewind (fin);
        (void) fseek(fin, offset, SEEK_SET);
        (void) fread(&object, sizeof(MY_TYPE_T), 1, fin);
        printf("%d %f %f %s\n",
        object.id,
        object.x_coord,
        object.y_coord,
        object.name);
        /* print the first entry */
        rewind (fin);
        (void) fread(&object, sizeof(MY_TYPE_T), 1, fin);
        printf("%d %f %f %s\n",
        object.id,
        object.x_coord,
        object.y_coord,
        object.name);
        (void) fclose(fin);
        return 0; }
```

- We Open the file using fopen, which automatically sets the read index to the start of the file.
- Since we want to read the last record first, we seek into the file using fseek highlighted in green color.
- The index that we specify is twice the size of the record size (My_TYPE_T).
- This puts us at the first byte of the third record, which is then read with the fread function highlighted in yellow color.
- Our read index is now at the end of the file, so we reset our read position to the top of the file using the rewind function highlighted in red color.
- We repeat this process, setting the file read position to the second element highlighted in blue color and then read again with fread.
- The final step is reading the first element in the file.
- This requires no fseek because after the rewind highlighted in purple color, we are at the top of

the file.
- We can then fread the first record highlighted in yellow color.

- **Nonsequential reads in a binary file**



- **The ftell function**

- Returns the current position as a long type and can be used to pass as the offset to fseek.
- The ftell prototype is:

```
long ftell (FILE *stream);
```

- An alternate API to ftell and fseek are fgetpos and fsetpos which provides the same functionality, but in a different form.
- An opaque type is used to represent the position (returned by fgetpos, passed into fsetpos).
- The prototypes for these functions are:

```
int fgetpos (FILE *stream, fpos_t *pos);
int fsetpos (FILE *stream, fpos_t *pos);
```

- **Example**

```
fpos_t file_pos;
……
/* Get desired position */
fgetpos (fin, &file_pos);
……
rewind(fin);
/* Return to desired
position */
fsetpos (fin, &file_pos);
```

- It is recommended to use fgetpos and fsetpos functions over the ftell and fseek methods.

- **Base API**

- The open, read and write functions can also be used for file I/O
  (Referred to as the base API because they are the platform for which the standard I/O library is built.
- The open function allows us to open or create a new file.
- Two variations are provided, with their APIs listed here:

```
int open (const char *pathname, int flags);
int open (const char *pathname, int flags, mode_t mode);
```

- The flags argument is one of O_RDONLY, O_WRONLY, or O_RDWR.
  (Note: for more flag and mode options visit https://man7.org/linux/man-pages/man2/open.2.html)

- **Open, read and write Example calls**

- To open a new file in the temp directory:

```
int fd;
fd = open ("/tmp/newfile.txt", O_CREAT, O_WRONLY);
```

- To instead open an existing file for read:

```
int fd;
fd = open ("/tmp/newfile.txt", O_RDONLY);
```

- Reading and writing to these files is done with:

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

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

- These are used simply with a buffer and a size to represent the number of bytes to read or write:

```
unsigned char buffer [MAX_BUF+1];
int fd, ret;
……..
ret = read (fd, (void *) buffer, MAX_BUF);
……..
Ret = write(fd, (void *) buffer, MAX_BUF);
```

- The set of functions to read and write data can also be used for pipes and sockets.
- A file descriptor can be attached to a stream by using the fdopen system call

```
FILE *fdopen (int filedes, const char *mode);
```

```
FILE *fp;
int fd;
fd = open ("/tmp/newfile.txt", O_RDWR);
fp = fdopen (fd, "rw");
```

- If we have opened a device using the open, we can associate a stream with it using fdopen:

Now we can use read/write with the fd descriptor or fscanf/fprintf with the fp descriptor

- One other useful API is the pread/pwrite API

  - These functions require an offset into the file to read or write, but they do not affect the file pointer.

```
ssize_t pread (int filedes, void *buf, size_t nbyte, off_t offset);
ssize_t pwrite (int filedes, void *buf, size_t nbyte, off_t offset);
```

Scripting Problems for Assessment (60 Minutes)

1. Develop your own cp command with name yournameCp which can do same tasks as Linux cp command. (30 minutes)

   Requirement:

   i. The source and destination files must be taken from the command line arguments like Linux cp command.

   ii. The developed command should run from any location.

2. Develop your own rename command with name yournamemv which can do same tasks as Linux mv command. (30 minutes)

   Requirement:

   i. Old and new name must be taken from the command line arguments like Linux mv command.

   ii. The developed command should run from any location.

Submission Instructions:

1. Submission requires the screen shots of all the incurred steps to execute the program.
2. All these files are in single document preferably in pdf format.
3. Use the naming convention: Prog_CourseCode_RollNo_LabNo.docx.
4. Submission is through LMS only