School of Computer Science and Engineering

Program- BTech-3rd Semester	Type- Sp. Core-I
Course Code- CSET213	Course Name-Linux and Shell Programming
Year- 2025	Semester- Odd
Date- 12/11/2025	Batch- Cyber Security

Lab Assignment 13

Exp No	Name	CO1	CO2	CO3	CO4
13	Unix File Structuring, inodes and Related System Calls	-	-	✓	-

Objectives: 1) To learn about file structure, inodes and use system calls for file operations. 2) To learn common system calls to make input-output operations on files, as well as operations to handle files and directories in the Linux.

Outcomes: After executing this assignment, the students will be able to use system calls in C to do file operations, and file structures.

Hands-on Learning Concepts on Unix File Structuring, inodes and Related System Calls (60min)

Unix File Structure

- Most UNIX file systems divide available disk space into two main types of regions: *inode* regions and *data* regions.
- UNIX file systems assign one *inode* to each file in the file system; a file's inode holds critical meta-data about the file such as its stat attributes and pointers to its data blocks.
- The data regions are divided into much larger (typically 8KB or more) *data blocks*, within which the file system stores file data and directory meta-data.
- Directory entries contain file names and pointers to inodes; a file is said to be *hard-linked* if multiple directory entries in the file system refer to that file's inode.
- Both files and directories logically consist of a series of data blocks, which may be scattered throughout the disk much like the pages of an environment's virtual address space can be scattered throughout physical memory.
- The file system environment hides the details of block layout, presenting interfaces for reading and writing sequences of bytes at arbitrary offsets within files.
- The file system environment handles all modifications to directories internally as a part of performing actions such as file creation and deletion.
- Our file system does allow user environments to *read* directory meta-data directly (e.g., with read), which means that user environments can perform directory scanning operations themselves (e.g., to implement the ls program) rather than having to rely on additional special calls to the file system.
- The disadvantage of this approach to directory scanning, and the reason most modern UNIX variants discourage it, is that it makes application programs dependent on the format of directory meta-data, making it difficult to change the file system's internal layout without changing or at least recompiling application programs as well.
- The file structure corresponds to a file open by a process and exists only in memory, being associated with an inode.
- It is the closest VFS entity to user-space; the structure fields contain familiar information of a user-space file (access mode, file position, etc.) and the operations with it are performed by known system calls (read, write, etc.).
- The file operations are described by the structure **struct file_operations**.

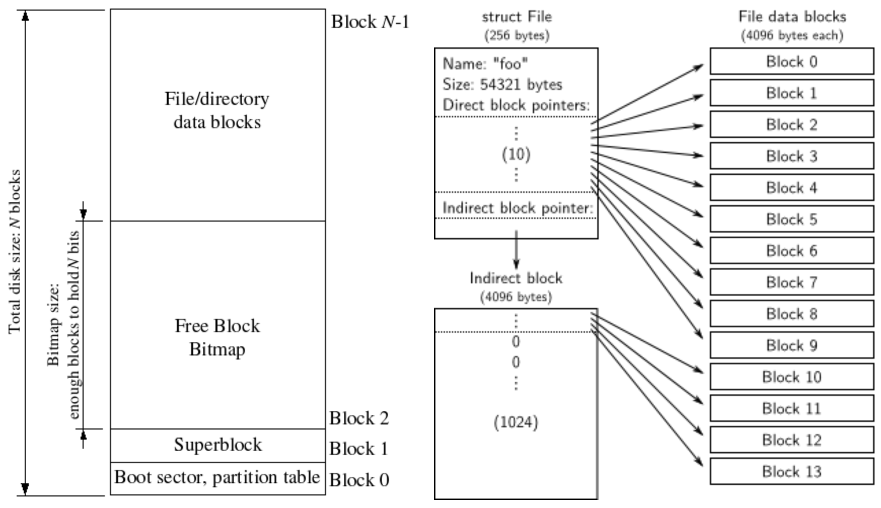
Sectors and Blocks in Unix File Structure

- Most disks cannot perform reads and writes at byte granularity and instead perform reads and writes in units of *sectors*.
- In most OS, sectors are 512 bytes each.
- File systems actually allocate and use disk storage in units of *blocks*.

- Be wary of the distinction between the two terms: *sector size* is a property of the disk hardware, whereas *block size* is an aspect of the OS using the disk.
- A file system's block size must be a multiple of the sector size of the underlying disk.
- The UNIX xv6 file system uses a block size of 512 bytes, the same as the sector size of the underlying disk.
- Most modern file systems use a larger block size, however, because storage space has gotten much cheaper and it is more efficient to manage storage at larger granularities.
- Our file system will use a block size of 4096 bytes, conveniently matching the processor's page size.

Super Block

- File systems typically reserve certain disk blocks at "easy-to-find" locations on the disk (such as the very start or the very end) to hold meta-data describing properties of the file system as a whole, such as the block size, disk size, any meta-data required to find the root directory, the time the file system was last mounted, the time the file system was last checked for errors, and so on. These special blocks are called *superblocks*.
- Our file system will have exactly one superblock, which will always be at block 1 on the disk.
- Its layout is defined by `struct Super` in `inc/fs.h`.
- Block 0 is typically reserved to hold boot loaders and partition tables, so file systems generally do not use the very first disk block.
- Many "real" file systems maintain multiple superblocks, replicated throughout several widely spaced regions of the disk, so that if one of them is corrupted or the disk develops a media error in that region, the other superblocks can still be found and used to access the file system.



File Meta-data

- The layout of the meta-data describing a file in our file system is described by `struct File` in `inc/fs.h`.
- This meta-data includes the file's name, size, type (regular file or directory), and pointers to the blocks comprising the file.
- As mentioned above, we do not have inodes, so this meta-data is stored in a directory entry on disk.
- Unlike in most "real" file systems, for simplicity we will use this one `File` structure to represent file meta-data as it appears *both on disk and in memory*.
- The `f_direct` array in `struct File` contains space to store the block numbers of the first 10 (`NDIRECT`) blocks of the file, which we call the file's *direct* blocks.
- For small files up to 10*4096 = 40KB in size, this means that the block numbers of all of the file's blocks will fit directly within the `File` structure itself.

- For larger files, however, we need a place to hold the rest of the file's block numbers.
- For any file greater than 40KB in size, therefore, we allocate an additional disk block, called the file's *indirect block*, to hold up to 4096/4 = 1024 additional block numbers.
- Our file system therefore allows files to be up to 1034 blocks, or just over four megabytes, in size.
- To support larger files, "real" file systems typically support *double*- and *triple-indirect blocks* as well.

Directories versus Regular Files

- A `File` structure in our file system can represent either a *regular* file or a directory; these two types of "files" are distinguished by the `type` field in the `File` structure.
- The file system manages regular files and directory-files in exactly the same way, except that it does not interpret the contents of the data blocks associated with regular files at all, whereas the file system interprets the contents of a directory-file as a series of `File` structures describing the files and subdirectories within the directory.
- The superblock in our file system contains a `File` structure (the `root` field in `struct Super`) that holds the meta-data for the file system's root directory.
- The contents of this directory-file is a sequence of `File` structures describing the files and directories located within the root directory of the file system.
- Any subdirectories in the root directory may in turn contain more `File` structures representing sub-subdirectories, and so on.

System calls when working with files

System call OPEN

Opening or creating a file can be done using the system call open. The syntax is:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path,
        int flags,... /* mode_t mod */);
```

- ✓ This function returns the file descriptor or in case of an error -1.
- ✓ The number of arguments that this function can have been two or three.
- ✓ The third argument is used only when creating a new file.
- ✓ When we want to open an existing file only two arguments are used.
- ✓ The function returns the smallest available file descriptor.
- ✓ This can be used in the following system calls: *read*, *write*, *lseek* and *close*.
- ✓ The effective UID or the effective GID of the process that executes the call has to have read/write rights, based on the value of the argument *flags*.
- ✓ The file pointer is places on the first byte in the file. The argument *flags* is formed by a bitwise OR operation made on the constants defined in the *fcntl.h* header.

O_RDONLY:	Opens the file for reading.
O_WRONLY:	Opens the file for writing.
O_RDWR:	The file is opened for reading and writing.
O_APPEND:	It writes successively to the end of the file.
O_CREAT:	The file is created in case it didn\92t already exist.
O_EXCL:	If the file exists and O_CREAT is positioned, calling *open* will fail.
O_NONBLOCK:	In the case of pipes and special files, this causes the open system call and any other future I/O operations to never block.
O_TRUNC:	If the file exists all of its content will be deleted.
O_SYNC:	It forces to write on the disk with function *write*. Though it slows down all the system, it can be useful in critical situations.

- ✓ The third argument, *mod*, is a bitwise OR made between a combination of two from the following list:

  `S_IRUSR, S_IWUSR, S_IXUSR:` Owner: *read*, *write*, *execute*.
  `S_IRGRP, S_IWGRP, S_IXGRP:` Group: *read*, *write*, *execute*.
  `S_IROTH, S_IWOTH, S_IXOTH:` Others: *read*, *write*, *execute*.
- ✓ The above define the access rights for a file and they are defined in the *sys/stat.h* header.

System call CREAT

- ✓ A new file can be created by:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *path, mode_t mod);
```

- ✓ The function returns the file descriptor or in case of an error it returns the value -1. This call is equivalent with:

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mod);
```

- ✓ The argument *path* specifies the name of the file, while *mod* defines the access rights.
- ✓ If the created file doesn't exist:
  - o a new i-node is allocated and a link is made to this file from the directory it was created in.
  - o The owner of the process that executes the call - given by the effective UID and the effective GUID - must have writing permission in the directory.
  - o The open file will have the access rights that were specified in the second argument (see *umask*, too).
  - o The call returns the smallest file descriptor available.
  - o The file is opened for writing and its initial size is 0.
  - o The access time and the modification time are updated in the i-node.
- ✓ If the file exists (permission to search the directory is needed),
  - o it loses its contents, and it will be opened for writing.
  - o The ownership and the access permissions won't be modified.
  - o The second argument is ignored.

System call READ

- ✓ When we want to read a certain number of bytes starting from the current position in a file, we use the *read* call. The syntax is:

```
#include <unistd.h>
ssize_t read(int fd, void* buf, size_t noct);
```

- ✓ The function returns the number of bytes read, 0 for end of file (EOF) and -1 in case an error occurred.
- ✓ It reads *noct* bytes from the open file referred by the *fd* descriptor and it puts it into a buffer *buf*.
- ✓ The pointer (current position) is incremented automatically after a reading that certain amount of bytes.
- ✓ The process that executes a read operation waits until the system puts the data from the disk into the buffer.

System call WRITE

- ✓ For writing a certain number of bytes into a file starting from the current position we use the *write* call, as:

```
#include <unistd.h>
ssize_t write(int fd, const void* buf, size_t noct);
```

- ✓ The function returns the number of bytes written and the value -1 in case of an error.
- ✓ It writes *noct* bytes from the buffer *buf* into the file that has as its descriptor *fd*.
- ✓ **It is interesting to note that the actual writing onto the disk is delayed.**

- ✓ This is done at the initiative of the root, without informing the user when it is done.
- ✓ If the process that did the call or another process reads the data that haven't been written on the disk yet, the system reads all this data out from the cache buffers.
- ✓ The delayed writing is faster, but it has three disadvantages:
    - a disk error or a system error may cause losing all the data
    - a process that had the initiative of a write operation cannot be informed in case a writing error occurred
    - the physical order of the write operations cannot be controlled.
- ✓ To eliminate these disadvantages, in some cases the O_SYNC is used.
- ✓ But as this slows down the system and considering the reliability of today's systems it is better to use the mechanism which includes using cache buffers.

System call CLOSE

- ✓ For closing a file and thus eliminating the assigned descriptor we use the system call *close*.

```
#include <unistd.h>
int close(int fd);
```

- ✓ The function returns 0 in case of success and -1 in case of an error.
- ✓ At the termination of a process an open file is closed anyway.

System call LSEEK

- ✓ To position a pointer (that points to the current position) in an absolute or relative way can be done by calling the *lseek* function.
- ✓ Read and write operations are done relative to the current position in the file. The syntax for *lseek* is:

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int ref);
```

- ✓ The function returns the displacement of the new current position from the beginning of the file or -1 in case of an error.
- ✓ There is not done any I/O operation and the function doesn't send any commands to the disk controller.
- ✓ Its *ref* is set to SEEK_SET the positioning is done relative to the beginning of the file (the first byte in the file is at position 0).
    - If *ref* is SEEK_CUR the positioning is done relative to the current position.
    - If *ref* is SEEK_END then the positioning is done relative to the end of the file.
- ✓ The system calls *open*, *creat*, *write* and *read* execute an *lseek* by default.
- ✓ If a file was opened using the symbolic constant O_APPEND then an *lseek* call is made to the end of the file before a write operation.

System call LINK

- ✓ To link an existing file to another directory (or to the same directory) link can be used.
- ✓ To make such a link in fact means to set a new name or a path to an existing file.
- ✓ The *link* system call creates a hard link.
- ✓ Creating symbolic links can be done using *symlink* system call. The syntax of link is:

```
#include <unistd.h>
int link(const char* oldpath, const char* newpath);
int symlink(const char* oldpath, const char* newpath);
```

- ✓ The function returns 0 in case of success and -1 in case of an error.
- ✓ The argument *oldpath* has to be a path to an existing file.
- ✓ Only the root has the right to set a link to a directory.

System call UNLINK

- ✓ To delete a link (a path) in a directory we can use the *unlink* system call. Its syntax is:

```
#include <unistd.h>
int unlink(const char* path);
```

- ✓ The function returns 0 in case of success and -1 otherwise.
- ✓ The function decrements the hard link counter in the i-node and deletes the appropriate directory entry for the file whose link was deleted.
- ✓ If the number of links of a file becomes 0 then the space occupied by the file and its i-node will be freed.
- ✓ Only the root can delete a directory.

System calls STAT, LSTAT and FSTAT

- ✓ To obtain more details about a file the following system call can be used: *stat*, *lstat* or *fstat*.

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char* path, struct stat* buf);
int lstat(const char* path, struct stat* buf);
int fstat(int df, struct stat* buf);
```

- ✓ These three functions return 0 in case of success and -1 in case of an error.
- ✓ The first two gets as input parameter a name of a file and completes the structure of the buffer with additional information read from its i-node.
- ✓ The *fstat* function is similar, but it works for files that were already opened and for which the file descriptor is known.
- ✓ The difference between *stat* and *lstat* is that in case of a symbolic link, function *stat* returns information about the linked (referred) file, while *lstat* returns information about the symbolic link file.
- ✓ The *struct stat* structure is described in the *sys/stat.h* header and has the following fields:

```
struct stat {
     mode_t st_mode;    /* file type & rights */
     ino_t st_ino;      /* i-node */
     dev_t st_dev;      /* număr de dispozitiv (SF) */
     nlink_t st_nlink;  /* nr of links */
     uid_t st_uid;      /* owner ID */
     gid_t st_gid;      /* group ID */
     off_t st_size;     /* ordinary file size */
     time_t st_atime;  /* last time it was accessed */
     time_t st_mtime;  /* last time it was modified */
     time_t st_ctime;  /* last time settings were changed */
     dev_t st_rdev;    /* nr. dispozitiv */
                        /* pt. fişiere speciale /
     long st_blksize;  /* optimal size of the I/O block */
     long st_blocks;    /* nr of 512 byte blocks allocated */
};
```

- ✓ The Linux command that the most frequently uses this function is *ls*.
- ✓ Type declarations for the members of this structure can be found in the *sys/stat.h* header.
- ✓ The type and access rights for the file are encrypted in the *st_mode* field and can be determined using the following macros:

Table 1. Macros for obtaining the type of a file

Macro	Meaning
`S_ISREG(st_mode)`	Regular file.
`S_ISDIR(st_mode)`	Directory file.
`S_ISCHR(st_mode)`	Special device of type character.
`S_ISBLK(st_mode)`	Special device of type block.
`S_ISFIFO(st_mode)`	Pipe file or FIFO.
`S_ISLNK(st_mode)`	Symbolic link.

- ✓ Decrypting the information contained in the *st_mode* field can be done by testing the result of a bitwise AND made between the *st_mode* field and one of the constants (bit mask): S_IFIFO, S_IFCHR, S_IFBLK, S_IFDIR, S_IFREG, S_IFLNK, S_ISUID (*suid* bit set), S_ISGID (*sgid* bit set), S_ISVTX (*sticky* bit set), S_IRUSR (read right for the owner), S_IWUSR (write right for the owner), S_IWUSR (execution right for the owner), etc.

System call ACCESS

- ✓ When opening a file with system call *open* the root verifies the access rights in function of the UID and the effective GID.
- ✓ There are some cases though when a process verifies these rights based upon the real UID and GID.
- ✓ A situation when this can be useful is when a process is executed with other access right using the *suid* or *sgid* bit.
- ✓ Even though a process may have root rights during execution, sometimes it is necessary to test whether the real user can or cannot access the file.
- ✓ For this we can use *access* which allows verifying the access rights of a file based on the real UID or GID.
- ✓ The syntax for this system call is:

```
#include <unistd.h>
int access(const char* path, int mod);
```

- ✓ The function returns 0 if the access right exists and -1 otherwise.
- ✓ The argument mod is a bitwise AND between R_OK (permission to read), W_OK (permission to write), X_OK (execution right), F_OK (the file exists).

System call UMASK

- ✓ To enhance the security in case of operations regarding the creation of files, the Linux offers a default mask to reset some access rights.
- ✓ Encrypting this mask is made in a similar way to the encrypting of the access rights in the i-node of a file.
- ✓ When creating a file those bits that are set to 1 in the mask invalidate the corresponding bits in the argument that specify the access rights.
- ✓ The mask does not affect the system call *chmod*, so the processes can explicitly set their access rights independently form the *umask* mask. The syntax for the call is:

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

- ✓ The function returns the value of the previous mask. The effect of the call is shown below:

```
main() /* test umask */
{
int fd;
umask(022);
if ((fd=creat("temp", 0666))==-1)
     perror("creat");
system("ls -l temp");
}
```

- ✓ The result will be of the following form: `-rw-r--r-- temp`
- ✓ Note that the write permission for the group and other users beside the owner was automatically reset.

System call CHMOD

- ✓ To modify the access rights for an existing file we use:

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

```
        int chmod(const char* path, mode_t mod);
```

- ✓ The function returns 0 in case of a success and -1 otherwise.
- ✓ The *chmod* call modifies the access rights of the file specified by the *path* depending on the access rights specified by the *mod* argument.
- ✓ To be able to modify the access rights the effective UID of the process has to be identical to the owner of the file or the process must have root rights.
- ✓ The *mod* argument can be specified by one of the symbolic constants defined in the *sys/stat.h* header.
- ✓ Their effect can be obtained by making a bitwise OR operation on them:

Table 2. Bit masks for testing the access rights of a file

Mode	**Description**
`S_ISUID`	Sets the suid bit.
`S_ISGID`	Sets the sgid bit.
`S_ISVTX`	Sets the sticky bit.
`S_IRWXU`	Read, write, execute rights for the owner obtained from: `S_IRUSR | S_IWUSR | S_IXUSR`
`S_IRWXG`	Read, write, execute rights for the group obtained from: `S_IRGRP | S_IWGRP | S_IXGRP`
`S_IRWXO`	Read, write, execute rights for others obtained from: `S_IROTH | S_IWOTH | S_IXOTH`

System call CHOWN

- ✓ This system call is used to modify the owner (UID) and the group (GID) that a certain file belongs to. The syntax of the function is:

```
        #include <sys/types.h>
        #include <unistd.h>
        int chown(const char* path, uid_t owner, gid_t grp);
```

- ✓ The function returns 0 in case of success and -1 in case of an error.
- ✓ Calling this function will change the owner and the group of the file specified by the argument *path* to the values specified by the argument's *owner* and *grp*.
- ✓ None of the users can change the owner of any file (even of his/her own files), except the root user, but they can change the GID for their own files to that of any group they belong to.

System call UTIME

- ✓ There are 3 members of the structure *stat* that refer to time. They are presented in table 3:

Table 3. Timing information associated with a file

Field	**Description**	**Operation**
`st_atime`	Last time the data in the file was accessed	Read
`st_mtime`	Last time the data in the file was modified	Write
`st_ctime`	Changing the settings for the i-node	chmod, chown

- ✓ The difference between the time the file was last modified and the change in the setting of the i-node is that the first one refers to the time when the contents of the file were modified while the second one refers to the time when the information in the i-node was last modified.
- ✓ This is since the information in the i-node is kept separately from the contents of the file.
- ✓ System calls that change the i-node are those ones which modify the access rights of a file, change the UID, change the number of links, etc.
- ✓ The system does not keep the time when the i-node was last accessed. Therefore, neither of the system calls *access* or *stat* do not change these times.
- ✓ The access time and last modification time of any kind of files can be changed by calling one of the system call presented below:

```
        #include <sys/time.h>
        int utimes(const char* path, const struct timeval* times);
        int lutimes(const char* path, const struct
timeval* times);
        int futimes(int fd, const struct timeval* times);
```

- ✓ The functions return 0 in case of success and -1 otherwise.
- ✓ Only the owner of a file or the root can change the times associated with a file.
- ✓ The parameter *times* represents the address (pointer) of a list of two *timeval* structures, corresponding to the access and modification time.
- ✓ The fields of the *timeval* structure are:

```
        struct timeval {
        long tv_sec; /* seconds passed since 01.01.1970 */
        suseconds_t tv_usec;     /* microseconds */
        }
```

- ✓ To obtain the current time in the form it is required by the *timeval* structure, we can use the *gettimeofday* function.
- ✓ For different conversions between the normal format of a data and hour and the format specific to the *timeval* structure the function *ctime* can be used or any other functions belonging to the same family (for more details see the Advanced Network Programming By Stevens).

Functions for working with directories

- ✓ A directory can be read as a file by anyone whoever has reading permissions for it.
- ✓ Writing a directory as a file can only be done by the kernel.
- ✓ The structure of the directory appears to the user as a succession of structures named directory entries.
- ✓ A directory entry contains, among other information, the name of the file and the i-node of this.
- ✓ For reading the directory entries one after the other we can use the following functions:

```
        #include <sys/types.h>
        #include <dirent.h>
        DIR* opendir(const char* pathname);
        struct dirent* readdir(DIR* dp);
        void rewinddir(DIR* dp);
        int closedir(DIR* dp);
```

- ✓ The *opendir* function opens a directory. It returns a valid pointer if the opening was successful and NULL otherwise.
- ✓ The *readdir* function, at every call, reads another directory entry from the current directory.
- ✓ The first *readdir* will read the first directory entry; the second call will read the next entry and so on.
- ✓ In case of a successful reading the function will return a valid pointer to a structure of type *dirent* and NULL otherwise (in case it reached the end of the directory, for example).
- ✓ The *rewinddir* function repositions the file pointer to the first directory entry (the beginning of the directory).
- ✓ The *closedir* function closes a previously opened directory. In case of an error it returns -1.
- ✓ The structure *dirent* is defined in the *dirent.h* file. It contains at least two elements:

```
        struct dirent {
             ino_t d_fileno;                 // i-node nr.
             char d_name[MAXNAMLEN + 1];    // file name
             }
```

Examples (30 Minutes)

Example 1: Write a program that creates a file with a 4K bytes free space. Such files are called files with holes.

```
    #include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
char buf1[]="LAB ";
char buf2[]="OS Linux";

int main( void)
{
     int fd;
     if ((fd=creat("file.gol", 0666)) < 0) {
          perror("Creation error");
           exit (1);
     }

     if (write(fd, buf1, sizeof(buf1)) < 0)
            perror("Writing error");
            exit(2);
     }

     if (lseek(fd, 4096, SEEK_SET) < 0)
            perror("Positioning error");
            exit(3);
     }

     if (write(fd, buf2, sizeof(buf2)) < 0)
            perror("Writing error");
            exit(2);
     }
}
```

Trace the execution of the program with the help of the following commands:

```
ls -l
stat file.gol
od -c file.gol
```

Example 2: Write a program that copies the contents of an existing file into another file. The names of the two files should be read as an input from the command line. You may presume that any of the commands *read* or *write* may cause errors.

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

#define BUFSIZE 512

int main (int argc, char** argv)
{
     int from, to, nr, nw, n;
     char buf[BUFSIZE];

     if ((from=open(argv[1], O_RDONLY)) < 0) {
            perror(Error opening source file);
            exit(1);
      }
```

```
            if ((to=creat(argv[2], 0666)) < 0) {
                    perror("Error creating destination file");
                    exit(2);
            }

           while((nr=read(from, buf, sizeof( buf))) != 0) {
              if (nr < 0) {
                 perror("Error reading source file");
                exit(3);
              }
              nw=0;
              do {
                if ((n=write(to, &buf[nw], nr-nw)) < 0) {
                  perror("Error writing destination file");
                  exit(4);
                }
                nw += n;
              } while (nw < nr);
            }
            close(from); close(to);
      }
```

Example 3: Write a program that displays the contents of a directory, specifying the type for each of its files. The name for the directory should be an input parameter.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
void listDir(char *dirName)
{
   DIR* dir;
   struct dirent *dirEntry;
   struct stat inode;
   char name[1000];
   dir = opendir(dirName);
   if (dir == 0) {
      perror ("Eroare deschidere fisier");
      exit(1);
   }
   while ((dirEntry=readdir(dir)) != 0) {
      sprintf(name,"%s/%s",dirName,dirEntry->d_name);
      lstat (name, &inode);

       // test the type of file
      if (S_ISDIR(inode.st_mode))
          printf("dir ");
      else if (S_ISREG(inode.st_mode))
            printf ("fis ");
           else
              if (S_ISLNK(inode.st_mode))
                printf ("lnk ");
            else;
      printf(" %s\n", dirEntry->d_name);
```

```
        }
    }

    int main(int argc, char **argv)
    {
        if (argc != 2) {
          printf ("UTILIZARE: %s nume_dir\n", argv[0]);
          exit(0);
        }
        printf(\94Continutul directorului este:\n\94);
        listDir(argv[1]);
    }
```

Scripting Problems for Assessment (30 Minutes)

Task: Write a program in C to open a file test.txt in append mode using open system call and perform the read write operations on text.txt using read/write system calls. Close this file using close system call.

Submission Instructions:

1. Submission requires the screen shots of all the incurred steps to execute a shell script or a video showing the whole process.
2. All these files are in single zip folder.
3. Use the naming convention: Prog_CourseCode_RollNo_LabNo.docx (Example: BTech3rdSem_CSET213_ E21CSEU002_Lab1.1)
4. Submission is through LMS only