



EE6253 - Operating Systems & Network Programming

O.G.Y.N. Gamlath
Department of Electrical and Information Engineering
Faculty of Engineering
University of Ruhuna

Lecture 06 – File System



Intended Learning Outcomes

- Introduction to file system
- File system types
- File system components
- File system operations
- File system management
- File system issues
- Low level file I/O
- High level file I/O



Introduction to File System

- **Definition:** A file system is a method used by OS to organize and store data on storage devices.
- **Purpose:** File systems provide a structured way to access, manage, and manipulate files stored on a storage medium.



File System Types

- **Disk-Based File Systems:**
 - Eg: NTFS (New Technology File System), FAT (File Allocation Table), ext4 (Fourth Extended File System).
 - Designed for traditional hard disk drives (HDDs) and solid-state drives (SSDs).
- **Network File Systems (NFS):**
 - Facilitates file sharing and storage over a network.
 - Eg: NFS, SMB (Server Message Block).
- **Distributed File Systems:**
 - Span multiple machines and coordinate file access and storage across a network.
 - Eg: Hadoop Distributed File System (HDFS), Google File System (GFS).



File System Components

- **File:** A file is a collection of data or information stored under a specific name on a storage device.
- **Directory:** Also known as folders, are containers used to organize files hierarchically within a file system.
- **Metadata:** Metadata refers to the data about the file or directory itself rather than its actual content. This includes information such as file name, size, type, permissions, creation date, and modification date.
- **File Attributes:** File attributes include properties such as read, write, and execute permissions, which determine who can access or modify the file. Other attributes may include ownership information and timestamps.
- **File System Operations:** File systems provide various operations to manage files and directories efficiently. These operations include creating, opening, reading from, writing to, renaming, moving, and deleting files and directories.



File System Operations

- `create()`
- `delete()`
- `open()`
- `close()`
- `read()`
- `write()`
- `append()`
- `truncate()`
- `rename()`
- `move()`
- `copy()`



File System Management

- **Storage Allocation:** File systems manage the allocation of storage space to files and directories. This involves strategies for allocating space efficiently.
- **Disk Partitioning:** File systems operate within partitions, which are logical divisions of a physical storage device. Disk partitioning involves dividing a storage device into multiple partitions, each formatted with its file system.
- **File System Maintenance:** Regular maintenance tasks such as disk defragmentation, error checking, and backup are essential for ensuring the health and reliability of file systems.



File System Issues

- **Data Corruption:** Data corruption can occur due to hardware failures, software bugs, or improper shutdowns, leading to loss or alteration of stored data.
- **File System Errors:** File system errors such as file system corruption or inconsistency can occur due to software bugs, hardware failures, or improper system shutdowns.
- **File System Security:** File systems must implement robust security mechanisms to protect sensitive data from unauthorized access, modification, or deletion.
- **File System Scalability:** File systems should be designed to scale efficiently as the storage requirements of the system grow over time. Scalability issues can arise due to limitations in file system architecture.

Low Level File I/O

- System Calls: Kind of software interruption which is going to interrupt the current execution of the process.
- We use the system calls for interact with OS to do some file I/O operations.
Eg: Low level system calls and High level system calls
- System Calls are defined in System Headers. You need to include header files in your program.
- Need to include the header files **<unistd.h>, <fcntl.h>, and <sys/stat.h>**
- UNIX I/O system calls return -1 on failure.
- File descriptors **0,1,2 are assigned to standard input (Keyboard), standard output (Display/Screen of your computer), standard error (can be used to identify errors occurred)** respectively.
- File descriptors can be used to identify the files.



open() System Call

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

- This system call are included in **<unistd.h>** header file.
- First argument (pathname) is a path to a file (location where the file is stored on your secondary storage).
- Path modes: Absolute Reference (exact path) & Relative Reference (path relative to the present directory)
- Second argument is flags which can be one of the followings. It contains of what purpose this file is open and related privileges for open file.
- These flags are defined in **<fcntl.h>** header file.

- 
- O_RDONLY – Open for read only
 - O_WRONLY – Open for write only
 - O_RDWR – Open for reading and writing

ORed with following if necessary. Use | to add them properly.

- O_CREAT – Create file if the file doesn't exist
- O_TRUNC – Delete the content if the file exist
- O_APPEND – Does not delete the content when the file exists. Writing will be appended to the existing file.
- Third argument (mode) of open is set of file permission for set of users. Typically, 0666 set read and write privileges for all users.
- Otherwise read, write, execute permissions for each user can be specified using,

- 
- S_IRWXU – Set read, write, execute permission for user
 - S_IRWXG – Set read, write, execute permission for user group
 - S_IRWXO - Set read, write, execute permission for other users

Or individual permission for each user can be set as follows.

- S_IWUSR – Set write permission for user
- S_IRGRP – Set read permission for group
- S_IXOTH – Set execute permission for others
- Relevant header file is **<sys/stat.h>**. Open() returns -1 on error.
- If you open a file you must close that file.



close() System Call

- Defined in **<unistd.h>**
- After you finished the present program, the file descriptor will not released.
- You are not using the file descriptor, you must close that as well.

int close(int file_des)

- Only one argument is appeared here (file descriptor).
- This closes the already opened file descriptor. Upon error returns -1, otherwise returns 0.



perror() System Call

- This is defined in the header file <errno.h>
- It prints the last encountered error to the standard error.
- The integer variable errno contains the error number of the last error.

void perror(const char * s)

- We can pass the file name here as the character.



write() System Call

size t write(int file_descriptor, void * buffer, size t size);

- In write() first argument is file descriptor (Which is the file need to write data to).
- Second argument is the memory address of the buffer from which the data is written. Data type is void pointer. Pointers are used to store the address of a variable.
- Third argument is the expected maximum size of the buffer (Number of bytes we expect to write the file).
- write() returns the number of bytes written to the file. Returns -1 on error.
- write() is used to write to a file from a memory buffer.



read() System Call

size_t read(int file_descriptor, void * buffer, size_t size);

- `read()` returns -1 in error. Otherwise returns the number of bytes read.
- `read()` is used to read from a file to a memory buffer.



File offset and lseek()

- This is defined in **<unistd.h>**
- The file offset is set to point to the start of the file when the file is opened and is automatically adjusted by each subsequent call to read() or write() so that it points to the next byte of the after the byte(s) just read or written.
- Successive read() and write() calls progress sequentially through a file.

```
off_t lseek(int file_des, off_t offset, int whence);
```

- The lseek() system call adjusts the file offset of the open file referred to by the file descriptor, according to the value specified in offset and whence.
- lseek() first argument is file descriptor.
- Second argument is the number of offset n bytes.
- Third argument is called whence and is one of the followings.
 - SEEK_SET or 0 – The file offset is set offset bytes from the beginning of the file.
 - SEEK_CUR or 1 – The file offset is adjusted by offset bytes relative to the current file offset.
 - SEEK_END or 2 – The file offset is set to the size of the file plus offset. That means offset is independent with respect to the next byte after the last byte of the file.

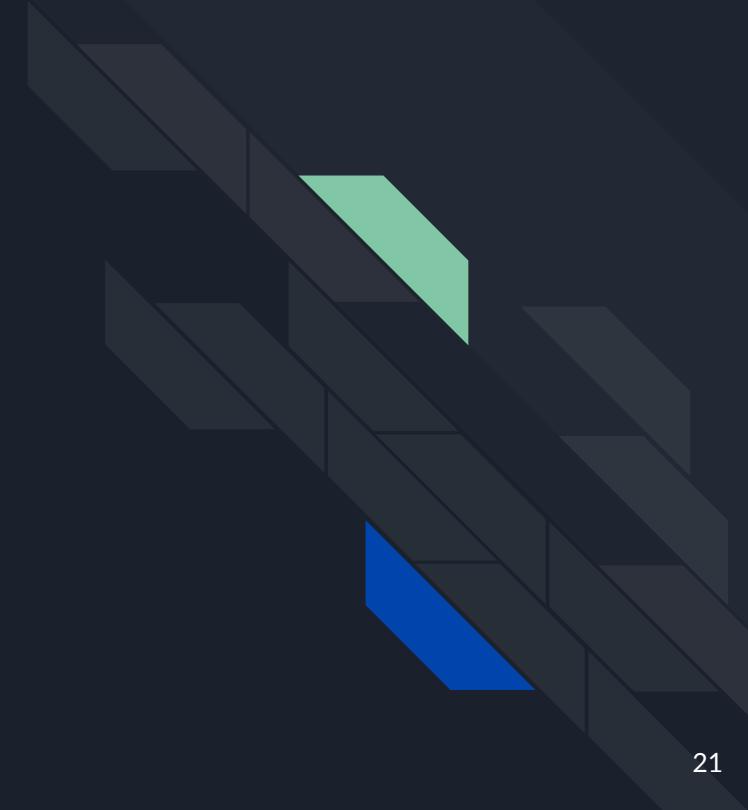


chmod() System Call

void chmod(const char * pathname, mode_t mode);

- Changes the mode of a closed file in pathname as shown by mode.
- Can change access privileges of the file using this. (read, write, execute)
- It should only be a closed file. You can't do the change mode operation for the open file.
- That's why you specify the path name here. If it is an open file you can pass the file descriptor here as well.

High Level File I/O



- 
- High Level I/O system calls return NULL pointer on failures.
 - OS environment is responsible for opening three files and providing pointers to them.

Standard Input, Standard Output, Standard Error

- The corresponding file pointers are called stdin, stdout, and stderr are declared in **<stdio.h>**
- Stdin is connected to the keyboard and stdout and stderr are connected to the screen, but stdin and stdout may be redirected to files.



fopen() System Call

- Defined in **<stdio.h>**
- The return type is a pointer to a structure called FILE.

FILE * fopen(const char * pathname, const char * mode);

- 1st argument (pathname) is a path to a file.
- 2nd argument (mode) is equivalent flags argument of a open() system call in the following manner.

mode in fopen()	flags in open()
r	O_RDONLY
w	O_WRONLY O_CREAT O_TRUNC
a	O_WRONLY O_CREAT O_APPEND
r+	O_RDWR
w+	O_RDWR O_CREAT O_TRUNC
a+	O_RDWR O_CREAT O_APPEND

- Changing user privileges is not done in fopen() unlike open(). In open() the argument mode was used to change the user privileges.
- fopen() returns NULL pointer on error. Otherwise it will return address of a FILE structure.



fclose() System Call

- Defined in `<stdio.h>`

int fclose(FILE * stream)

- This closes the already opened file descriptor FILE pointer stream.
- Upon error; returns -1. otherwise return 0.



perror() System Call

- This is defined in the header file <errno.h>

```
void perror(const char * s)
```

- Prints the last encountered error to the standard error.
- The integer variable errno contains the error number of the last error.



fprintf() System Call

- Defined in `<stdio.h>`

```
int fprintf (FILE * stream, const char * format, variables for format);
```

- **fprintf()** is used to write to a file from a memory buffer or user defined input.
- In **fprintf()** first argument is file stream, second argument is a string called **format** which contains a pointer to the string to be written.
- **fprintf()** returns the number of bytes written. Returns -1 on error.



fscanf() System Call

- Defined in **<stdio.h>**

```
int fscanf (FILE * stream, const char * format, address of variable  
with format);
```

- fscanf() is used to read from a file stream to a memory buffer.
- fscanf() returns -1 in error.
- Returns number of bytes read upon meeting “ ” (space) character.
- Upon coming to the end of the file, fscanf() returns EOF.



chmod() System Call

- Defined in `<sys/stat.h>`

```
void chmod (const char * pathname, mode_t mode);
```

- Changes the mode of a closed file in pathname as shown by mode.



Thank You...