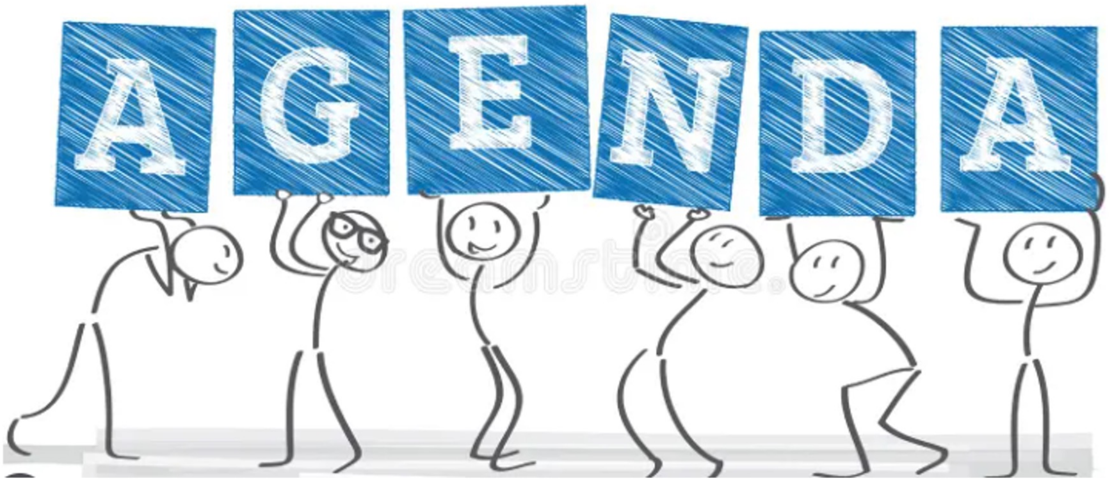
# Operating Systems

## Lecture 2.3

UNIX File System Architecture (Part-II)

Instructor: Muhammad Arif Butt, PhD

# Lecture Agenda




- I/O Redirection in C Programs
- Playing with file offset (`lseek`)
- Misc File Related System Calls
- Accessing File Attributes
  - Determining File Type (`stat`)
  - Determining File Permissions (`stat`)
  - Determining File Owner (`getpwuid`)
  - Determining File Group (`getgrgid`)
  - Determining File Time Stamps (`ctime`)
- Directory Management Calls

# I/O Redirection in C Programs

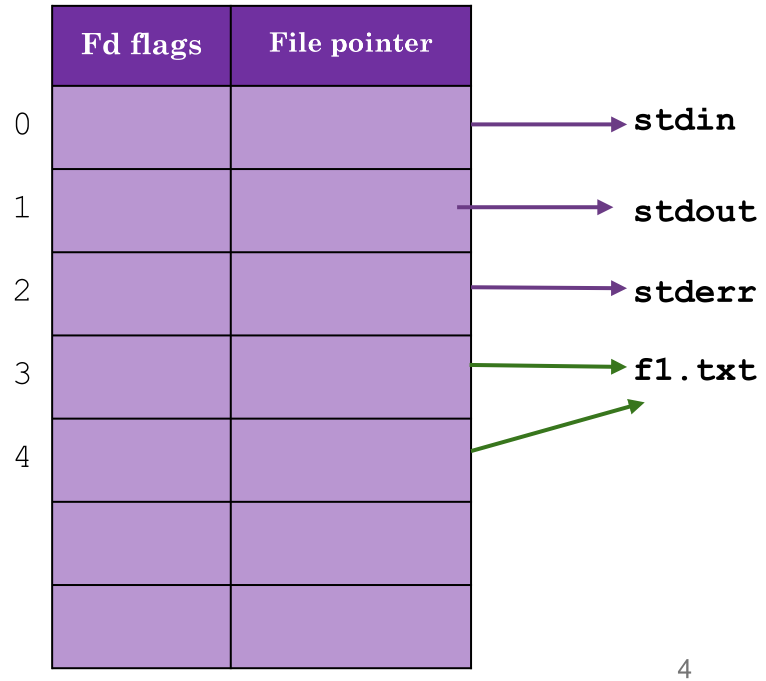
# dup() System Call

```
int dup(int oldfd);
```

- The `dup()` call takes `oldfd`, an open file descriptor, and returns a new descriptor that refers to the same open file description.
- The old and the new descriptor both point to the same entry in the SWFT. After a successful return from these function , old and new fd's can be used interchangeably.
- The new descriptor is guaranteed to be the lowest unused file descriptor
- If we run the following LOCs, the `open()` call will return 3, the `dup()` call will return the lowest unused descriptor which will be zero. Finally descriptor zero points to the opened file instead of `stdin`.

```
fd = open(...);
close(0);
newfd = dup(fd);
```

- To make the above code simpler, and to ensure we always get the file descriptor we want, we can use `dup2()` .



# dup2() System Call

```
int dup2(int oldfd, int newfd);
```

- The `dup2()` system call makes a duplicate of the file descriptor given in `oldfd` using the descriptor number supplied in `newfd`.
- If the file descriptor specified in `newfd` is already open, `dup2()` closes it first.
- We can simplify the preceding calls to `close(0)` and `dup(fd)` on previous slide to the following:

```
dup2(fd, 0);
```

- A successful `dup2()` call returns the number of the duplicate descriptor i.e., the value passed in `newfd`
- If `oldfd` is a valid file descriptor, and `oldfd` and `newfd` have the same value, then `dup2()` does nothing. In this case, `newfd` is not closed, and `dup2()` returns the `newfd`

# Input Redirection

## Method 1: `close-open`

```
close(0);
fd = open("/etc/passwd", O_RDONLY);
```

## Method 2: `open-close-dup-close`

```
fd = open("/etc/passwd", O_RDONLY);
close(0);
newfd = dup(fd);
close(fd);
```

## Method 3: `open-dup2-close`

```
fd = open("/etc/passwd", O_RDONLY);
newfd = dup2(fd, 0);
close(fd);
```

Use Method 3 in all scenarios. It's the standard, safe, and atomic way to perform input redirection. Methods 1 and 2 should be avoided in production code due to race conditions, though Method 1 might be acceptable in simple single-threaded educational examples where error handling isn't critical.

# Output Redirection

### Method 1: `close-open`

```
close(1);
fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

### Method 2: `open-close-dup-close`

```
fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
close(1);
newfd = dup(fd);
close(fd);
```

### Method 3: `open-dup2-close`

```
fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
newfd = dup2(fd, 1);
close(fd);
```

**Note: For error redirection, simply replace the file descriptor (1 with 2) in all above code snippets**

# Demonstration


**GitHub Code Repository Link:** https://github.com/arifpucit/OS-Codes

# Playing with File offset

# `lseek()` System Call

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

- For each open file, the kernel records a file offset, also called current file offset (cfo), which is there in the SWFT. This is the location in the file at which the next `read()` or `write()` will commence. The file offset is expressed as an ordinal byte position relative to the start of the file. The first byte of the file is at offset 0.
- The file offset is set to point to the start of the file when the file is opened (unless the `O_APPEND` option is specified) and is automatically adjusted by each subsequent call to `read()` or `write()` so that it points to the next byte of the file after the byte(s) just read or written. Thus, successive `read()` and `write()` calls progress sequentially through a file.
- The `lseek()` system call adjusts the file offset of the open file referred to by the file descriptor `fd`, according to the values specified in `offset` and `whence`. On success, returns the resulting offset location and -1 on failure.

# `lseek()` System Call

The whence directive can take following values:

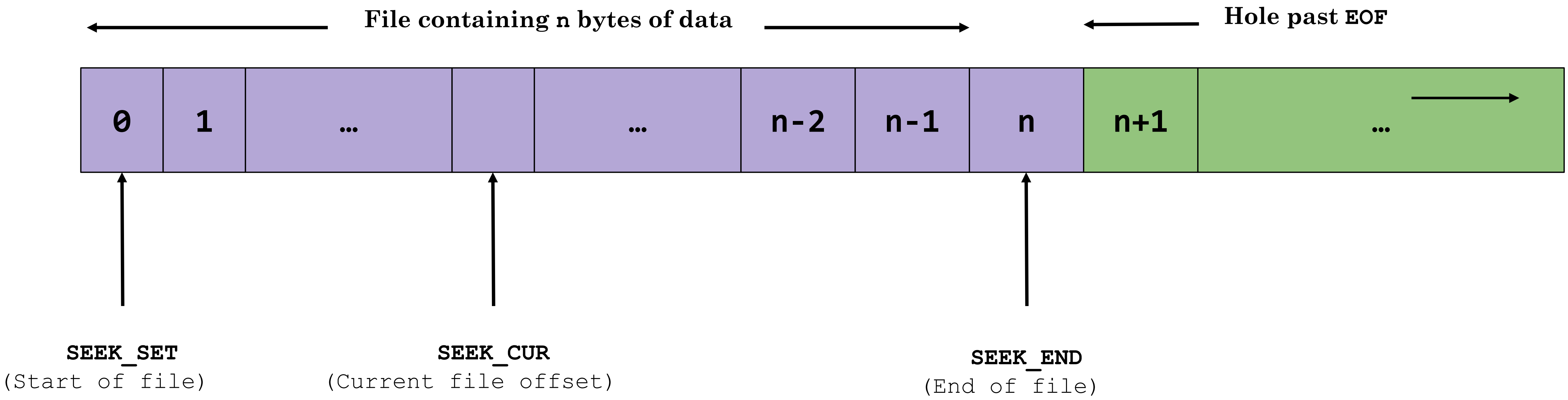
| WHENCE | | Description |
|---|---|---|
| SEEK_SET | 0 | The CFO is set offset bytes from the beginning of the file |
| SEEK_CUR | 1 | The CFO is set offset bytes from current value of CFO |
| SEEK_END | 2 | The CFO is set offset bytes from the end of the file |
| SEEK_HOLE | 3 | The CFO is set to start of the next hole greater than or equal to offset |
| SEEK_DATA | 4 | The CFO is set to start of the next non-hole (i.e., data region) greater than or equal to offset |

**Example**

```
off_t position;
position = lseek(fd, 0, SEEK_CUR);    //return current file offset
position = lseek(fd, 0, SEEK_END);    //next byte after the end of the file
position = lseek(fd, -10, SEEK_CUR);  //ten bytes prior to current location
position = lseek(fd, -1, SEEK_END);   //last byte of file
position = lseek(fd, 100, SEEK_END);  //101 bytes past last byte of file
```

# lseek() System Call

## Interpreting whence argument of lseek() system call



# Demonstration


**GitHub Code Repository Link:** https://github.com/arifpucit/OS-Codes

# Misc File Related System Calls

# Truncating a File

```
int truncate(const char* path, off_t length);

int ftruncate(int fd, off_t length);
```

- Truncating a file means chopping off contents from the tail of the file. The use of the `O_TRUNC` flag in `open()` call is special case of truncation where the file size is reduced to zero and the cfo and EOF are set to location 0
- The `truncate()` and `ftruncate()` system calls truncate an existing file to length bytes.
- If length is smaller than the existing length of the file, the contents of the file beyond length bytes are not accessible anymore.
- If length is greater than the current file size, the file size is increased to length and the space between the previous `EOF` and new `EOF` is filled with zeros and becomes a hole.
- The difference between the two system calls lies in how the file is specified. With `truncate()`, the file, which must be accessible and writable, is specified as a pathname string. If pathname is a symbolic link, it is dereferenced. The `ftruncate()` system call takes a descriptor for a file that has been opened for writing. It doesn’t change the file offset for the file

# rename() Function

```
int rename(const char*oldpath, const char* newpath);
```

- A programmer can rename a file or a directory with the `rename()` library function
- A sample code snippet that renames a file named `file1` to `file2` in the present working directory is shown below:

```
if(rename("file1","file2") == -1)
    perror("rename(1)");
```

# `remove()` and `unlink()`

```
int remove(const char *pathname);

int unlink(const char* pathname);
```

- Remove is a library call that deletes a name from file system. It calls `unlink()` for files and `rmdir()` for directories
- However, if any process has this file open currently, the file won't be actually erased until the last process holding it open closes it. Until then it will be removed from the directory (i.e., ls won't show it), but not from disk.
- **When a file is deleted, the OS Kernel performs following tasks:**

  i. Frees the inode number associated with that file.

  ii. Frees all the data blocks associated with that file and add them to the list of free blocks.

  iii. Delete the entry from the directory containing that file.

**Note:** The metadata of the file is still there in the inode block and the data of the file in its data blocks. You just need to know how to access those blocks :)

# chown, fchown and lchown Function Instr

```
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *linkname, uid_t owner,gid_t group);
```

- `chown()` changes the owner and group of the file specified by path
- `fchown()` is identical to `chown()` except that file is specified by file descriptor "fd"
- `lchown()` is similar to `chown()`, except if path is a symbolic link, then the link itself is changed , not the file it refers to.
- If owner or group is specified as -1, then that ID is not changed.
- Only a process with super user privileges can use these functions to change any file user ID and group ID.

# chmod() and fchmod() System Call

```
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

- These two functions allow us to change the file access permissions for an existing file.
- The `chmod()` function operates on the specified file, whereas the `fchmod()` function operates on a file that has already been opened using its file descriptor.
- The `mode` is the same as discussed in the flags argument of `open()`
- Following code snippet will give the owner read and write permissions to the file and deny access to all other users.

```
if (chmod(filename, 0644) == -1)
   perror("chmod failed");
```

```
mode_t new_mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
if (chmod(filename, new_mode) == -1)
   perror("chmod failed");
```

# umask() Function

```
mode_t umask(mode_t mask);
```

- The `umask()` is essential for controlling default file permissions in Unix/Linux systems. The file mode creation mask is used whenever the process creates a new file or a new directory.
- The `umask()` function sets the file mode creation "mask" for the process and returns the previous value.
- Remember the mask value of a process is the same as that of its creating shell, i.e. its parent. (mask value is inherited after fork)

```
umask(0077);
int fd = open("myfile.txt", O_CREAT | O_RDWR, 0633);
```

**After the above code executes, the resulting permissions on `myfile.txt` are `rw- --- ---` (`mode &~umask`)**

# access() System Call

```
int access(const char *pathname, int mode);
```

- The `access()` system call determines whether the calling process has access permission to a file or not and it can also check for file existence as well.
- The mode argument is a bit mask consisting of one or more of the permission constants.
- If a process has all the specified permissions the return value is 0, otherwise the return value is -1 & sets errno to `EACCES`
- The `open()` system call performs its access tests based on the EUID and EGID, while the `access()` system call bases its tests on the real UID & GID

| Mode | Description |
|---|---|
| `R_OK` | Test for read permission |
| `W_OK` | Test for write permission |
| `X_OK` | Test for execute permission |
| `F_OK` | Test for existence of file |

# Symlink and link Function

```
int symlink(const char* oldpath, const char* newpath);
int link(const char* oldpath, const char* newpath);
```

- The `link()` and `symlink()` functions are used to create a hard link and a soft link to a file respectively.
- Following sample code snippets show the usage of these library functions:

```
if(symlink("/tmp/file1", "/home/arif/slinktofile1") == -1){
perror("symlink"); exit(1);}
```

```
if(link("/tmp/file1", "/home/arif/hlinktofile1") == -1) {
perror("link"); exit(1); }
```

# Demonstration


**GitHub Code Repository Link:** *https://github.com/arifpucit/OS-Codes*

# Accessing File Attributes

# Accessing File Attributes

```
int stat(const *char pathname, struct stat *buff);
int fstat(int fd, struct stat *buff);

int lstat(const *char linkname, struct stat *buff);
```

- Above functions can be used to access the file attributes stored in its inode. No permissions are required on the file itself, however, execute (search) permission is required on all of the directories in pathname that lead to the file.
- `stat()` stats the file pointed by "path" and fills in "buff"
- `fstat()` is identical to `stat()` except that file to be stated is specified by file descriptor "fd"
- `lstat()` is similar to `stat()`, except if path is a symbolic link, then the link itself is stated , not the file it refers to.
- **On success** returns 0 and on error returns `-1` and set `errno`.
- On success, populate the **`stat`** structure as mentioned on next slide

# File Attributes

```
struct stat{
      dev_t st_dev;                     //ID of device containing file
      ino_t st_ino;                     //inode number
      mode_t st_mode;                   //file type & permission
      nlink_t st_nlink;                 //number of hard links
      uid_t st_uid;                     //user ID of owner
      gid_t st_gid;                     //group ID of owner
      off_t st_size;                    //total size in bytes
      time_t st_atime;                  //time of last access
      time_t st_mtime;                  //time of last data modification
      time_t st_ctime;                  //time of last status change
      struct timespec st_birthtim;      //file creation time
      blksize_t st_blksize;             //block size for I/O
      blkcnt_t st_blocks;               //numb of 512B blocks allocated
      };
```

# Understanding the `st_mode` of `struct stat`

| File Type (4 bits) | Special Permissions (3 bits) | User (3 bits) | Group (3 bits) | Others (3 bits) |
|---|---|---|---|---|

The lower 16 bits of `st_mode` member of `struct stat` is shown above that contains information about file type and permissions

**Example:** If it contains a value of $100640_8 = 81A0_{16} = 1000000110100000_2$

- Bit 2-0 specifies permissions for others (000): **`---`**
- Bit 5-3 specifies permissions for group members ((100): **`r--`**
- Bit 8-6 specifies permissions for owner (110): **`rw-`**
- Bit11-9 specifies special permissions (000): **`---`**
- Bit15-12 specifies file type (1000): Regular File

| File Type | Symbol | Binary (bits 15-12) | Constant |
|---|---|---|---|
| Regular File | - | 1000 | S_IFREG |
| Directory | d | 0100 | S_IFDIR |
| Character Device | c | 0010 | S_IFCHR |
| Block Device | b | 0110 | S_IFBLK |
| FIFO/Named Pipe | p | 0001 | S_IFIFO |
| Symbolic Link | l | 1010 | S_IFLNK |
| Socket | s | 1100 | S_IFSOCK |

# Using Macros to Determine File Type

| File Type (4 bits) | Special Permissions (3 bits) | User (3 bits) | Group (3 bits) | Others (3 bits) |
|---|---|---|---|---|

- The file `/usr/include/linux/stat.h` contains some related macros. So to decipher the file type, instead of creating your own mask, one can use these macros:

```
#define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)
#define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)
```

- A sample code snippet that uses these macros to determine file type is shown below:

```
if (S_ISREG(buf.st_mode))
    printf("Regular File\n");

else if (S_ISDIR(buf.st_mode))
    printf("Directory\n");
```

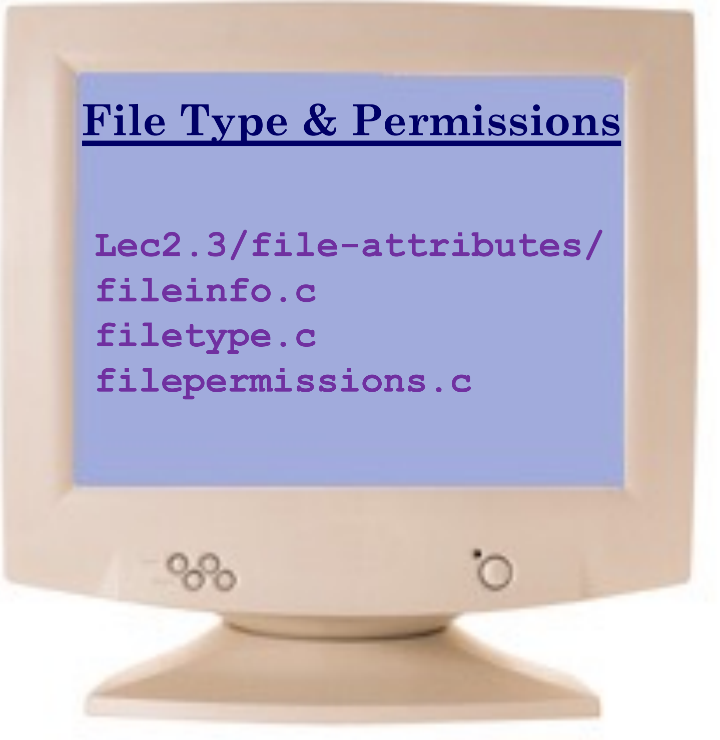
# Determining File Permissions

| File Type (4 bits) | Special Permissions (3 bits) | User (3 bits) | Group (3 bits) | Others (3 bits) |
|---|---|---|---|---|

- The way we have determined the file type, similarly we can identify the different permission sets a file has.
- We can create a mask to determine the permissions a file has by setting the specific permission bit ON. Then we can perform a bitwise & operation of the `st_mode` value with the specific `mask`, and check if the specific bit for that permission is set or not. If it is set that means the permission is ON otherwise it is OFF.
- Following code snippet determine whether the file owner has read, write and execute permissions on the file:

```
if((buf.st_mode & 0000400) == 0000400)
            printf("Owner has read permission\n");
if((buf.st_mode & 0000200) == 0000200)
            printf("Owner has write permission\n");
if((buf.st_mode & 0000100) == 0000100)
            printf("Owner has execute permission\n");
```

# Demonstration




**GitHub Code Repository Link:** https://github.com/arifpucit/OS-Codes

# Accessing Information about File Owner

```
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
```

- The `getpwuid()` function is passed a user ID (UID) and it searches that user account information from the system's password database (`/etc/passwd`).
- Returns a pointer to a `struct passwd` on success and NULL on failure

```
struct passwd {
    char    *pw_name;        /* username */
    char    *pw_passwd;      /* user password (usually 'x') */
    uid_t    pw_uid;         /* user ID */
    gid_t    pw_gid;         /* group ID */
    char    *pw_gecos;       /* user information/comment */
    char    *pw_dir;         /* home directory */
    char    *pw_shell;       /* login shell */
};
```

# Accessing Information about File Group

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
```

- The `getprgid()` function is passed a group ID (GID) and it searches that group information from the system's group database (`/etc/group`).
- Returns a pointer to a `struct group` on success and NULL on failure

```
struct group {
    char    *gr_name;     /* group name */
    char    *gr_passwd;   /* group password (usually unused) */
    gid_t    gr_gid;      /* group ID */
    char   **gr_mem;      /* null-terminated array of strings containing usernames */
};
```

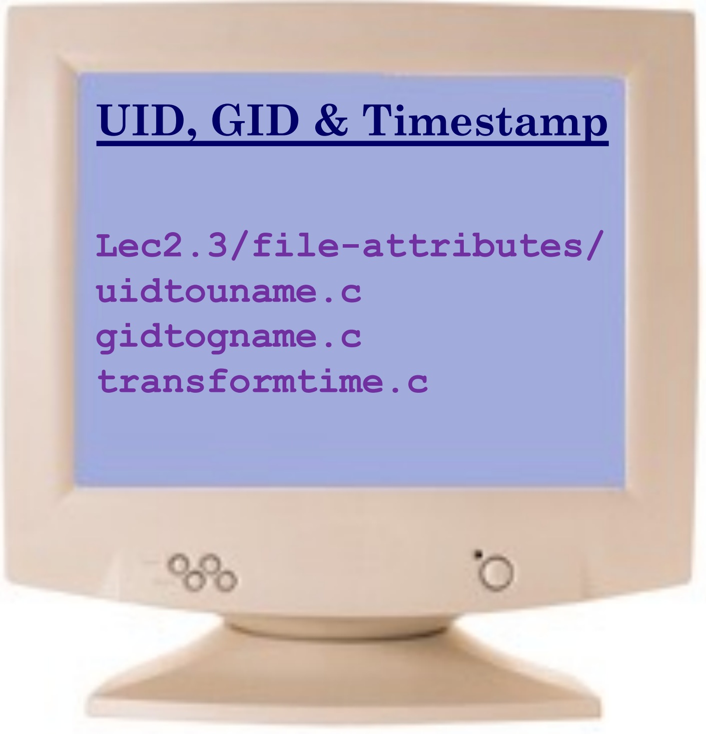
# Accessing File Time Stamps

```
#include <time.h>
time_t time(time_t *tloc);
char *ctime(const time_t *timep);
```

- The `time()` function returns the current time as the number of seconds, since UNIX epoch (January 1, 1970, 00:00:00 UTC). The optional parameter is a pointer which gets populated with the result, however, it is normally set to NULL and we use the return value instead.
- The `ctime()` function is converts a time value represented as seconds passed since UNIX epoch into a human readable string format.
- On success, returns pointer to a static string having 26 characters including newline and null terminator:

"Wed Jul 21 12:34:56 2025\n"

# Demonstration




**GitHub Code Repository Link:** https://github.com/arifpucit/OS-Codes

# Directory Management Calls

# Creating and Deleting directories

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

int rmdir(const char *pathname);
```

- The `mkdir()` function creates a new, empty directory with two entries . & .., and new directory will be owned by the effective UserID of the process. Permissions on the created directory can be calculated by: **`mode & ~umask`**. An empty directory has a link count of 2. As you add subdirectories to this directory, the link count will increase by 1 for each subdirectory (because each subdirectory's .. entry creates an additional hard link back to the parent directory).

```
umask(0022);
mkdir("mydir", 0755);
```

```
umask(0077);
mkdir("mydir", 0755);
```

```
umask(0022);
mkdir("mydir", 0777);
```

- The `rmdir()` function deletes an empty directory by removing the . and .. entries and the link count reaches zero.
- When link count reaches 0, and no other processes have it open, kernel frees:
  - The directory's inode.
  - The data blocks containing the directory entries.
  - Any associated metadata.

**To delete a directory with contents, you must manually traverse the directory tree and delete each file/subdirectory individually, then finally delete the empty parent directory using `rmdir()`.**

# Opening directories

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char* dirpath);
```

- The `opendir()` function opens the directory specified by `dirpath` and returns a pointer to a structure of type `DIR` that can be used to refer to the directory in later calls.
- Upon return from `opendir()`, the directory stream (`DIR`) is positioned at the first entry in the directory list.
- Directories can be read by anyone who has access permission to read the directory. But only the kernel can write to a directory, so the write permission bits and execute permission bits for a directory determine if we can create new files in the directory and remove files from the directory

```
int closedir(DIR *dirp);
```

- Closes the directory stream associated with "`dirp`". The directory stream descriptor "`dirp`" is not available after this call.

# Changing directory

```
char *getcwd(char *buf, size_t size);
int chdir(const char *pathname);
```

- Every process has a current working directory (cwd), where the search for all relative "pathnames" starts. You can access it using the `getcwd()` function. The `buf` is a pointer to buffer where the current working directory path will be stored and `size` is the size of the buffer in bytes. It returns pointer to `buf` on success and returns NULL in case of error and sets `errno`
- We can change the current working directory of the calling process by calling the `chdir()` function. The pathname is a pointer to string containing the directory path to change. It returns 0 on success and returns -1 in case of error and sets `errno`.

```
char cwd[PATH_MAX];
getcwd(cwd, sizeof(cwd))
printf("Current working directory: %s\n", cwd);
printf("\nChanging directory to /tmp...\n");
chdir("/tmp")
```

# Reading Directories

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- The `readdir()` function reads successive entries from a directory stream. Each call to `readdir()` reads the next directory entry from the directory stream referred to by **`dirp`** and returns a pointer to a statically allocated structure of type **`dirent`**, containing the following information about the entry (it may vary from OS to OS):

```
struct dirent {
    ino_t d_ino;       /* File i-node number */
    char d_name[];     /* Null-terminated name of file */
};
```

- This structure is overwritten on each call to `readdir()`
- The filenames returned by `readdir()` are not in sorted order, but rather in the order in which they happen to occur in the directory, this depends on the order in which the file system adds files to the directory and how it fills gaps in the directory list after files are removed. (The command `ls -f` lists files in the same unsorted order that they would be retrieved by `readdir()`)

# Reading Directory

On end-of-directory or error, `readdir()` returns `NULL`, in the latter case setting `errno` to indicate the error. To distinguish these two cases, we can write the following:

```
errno = 0;
struct dirent *entry = readdir(dp);
if (entry == NULL && errno != 0) {
    /* Handle error */
} else if (entry == NULL) {
    /* We reached end-of-directory */
}
```

If the contents of a directory change while a program is scanning it with `readdir()`, the program might not see the changes. SUSv3 explicitly notes that it is unspecified whether `readdir()` will return a filename that has been added to or removed from the directory since the last call to `opendir()`. All filenames that have been neither added nor removed since the last such call are guaranteed to be returned

# Directory Stream Functions

```
#include <dirent.h>
off_t telldir(DIR *dirp);
```

- The `telldir()` function returns the current location associated with the directory stream `dirp`. On error, -1 is returned, and `errno` is set appropriately.

```
#include <dirent.h>
void seekdir(DIR *dirp, off_t offset);
```

- The `seekdir()` function sets the location in the directory stream from which the next `readdir()` call will start. The `seekdir()` should be used with an offset returned by `telldir()`. The `seekdir()` function returns no value

# Demonstration


**GitHub Code Repository Link:** *https://github.com/arifpucit/OS-Codes*

# To Do:

Consider the following shell command:

```
$ grep kakamanna < /etc/passwd > out.txt
```

- The grep command gets string kakamanna as command line argument
- The input of the grep command is attached to `/etc/passwd`
- The output of the grep command is attached to `out.txt`

**How can we write a C program that can do this?**

```
$ ./a.out kakamanna < /etc/passwd > out.txt
```

- Receives `argv[1]` as a search string
- Open `argv[2]` file in read mode and `argv[3]` file in write mode
- Use `dup2()` to duplicate descriptor 0 with fd of input file
- Use `dup2()` to duplicate descriptor 1 with fd of output file
- Use `close()` to close the descriptors achieved in step 2
- Finally `exec` your program with grep program by passing it the only command line argument, i.e., the search string

# To Do

- Watch SP video on File related system calls:
https://www.youtube.com/watch?v=DZQkyoXgkMs&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=13

- Watch SP video on **`ls`** Utility:
https://www.youtube.com/watch?v=24WNjxn4asY&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=14


**Coming to office hours does NOT mean that you are academically weak!**