HANDS ON LIST 1

Question 1

Create the following types of a files using (i) shell command (ii) system call
a. soft link (symlink system call)
b. hard link (link system call)
c. FIFO (mkfifo Library Function or mknod system call)

ANSWER:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
    if (strcmp(argv[1], "softlink") == 0 && symlink(argv[2], argv[3]) != -1) // 0→s
uccess, -1→failure
    {
        printf("Files have been soft linked using system call symlink.\n");
    }
    else if (strcmp(argv[1], "hardlink") == 0 && link(argv[2], argv[3]) != -1)
    {
        printf("Files have been hard linked using system call link.\n");
    }
    else if (strcmp(argv[1], "pipeline_mkfifo") == 0 && mkfifo(argv[2], 0644) !=
-1)
    {
        printf("Pipeline Created Successfully using mkfifo syscall.\n");
    }
    else if (strcmp(argv[1], "pipeline_mknod") == 0 && mknod(argv[2], S_IFIFO |
```

```
0644, 0) != -1) // For device files  use mknod().
    {
        printf("Pipeline Created Successfully using mknod syscall.\n");
    }
    return 0;
}
```

OUTPUT:

```
*Shell Commands:
1 → ln -s souce_file soft_link (for softlink).
2 → ln source_file hard_link (for hardlink).
3 → mkfifo myfifo (for named pipe).
4 → mknod myfifo p (for named pipe).

adityadave@Adityas-MacBook-Air-3 Que1 % ./1 softlink 1.c soft_lnk
Files have been soft linked using system call symlink.
adityadave@Adityas-MacBook-Air-3 Que1 % ./1 hardlink 1.c hard_lnk
Files have been hard linked using system call link.
adityadave@Adityas-MacBook-Air-3 Que1 % ./1 pipeline_mkfifo pipe_mkfifo
Pipeline Created Successfully using mkfifo syscall.s
adityadave@Adityas-MacBook-Air-3 Que1 % ./1 pipeline_mknod pipe_mknod
Pipeline Created Successfully using mknod syscall.

// check file type using ls -l.
```

Concept:

(a) Soft Link (Symbolic Link)

- A **soft link** is like a shortcut.
- It contains the pathname of the target file, not the file data itself.

- If the original file is deleted, the soft link becomes *dangling* (broken).
- Can span across different filesystems.
- A soft link is a **separate inode** of type **symbolic link**.
- Instead of pointing directly to data blocks, this inode contains a **string (the pathname of the target file)**.
- When you open a symlink, the kernel reads the stored pathname and tries to open the target.
- If the target is deleted, the symlink points to nothing → becomes **dangling**.

(b) Hard Link

- A **hard link** points directly to the same inode of the file.
- Both the original file and hard link share the same inode number.
- File data remains until all hard links are deleted.
- Cannot span different filesystems.
- Cannot generally be created for directories.
- Every file in Linux is represented by an **inode** (metadata + pointer to data blocks).
- The **directory entry** just maps a **filename → inode number**.
- When you create a **hard link**, you are simply adding **another filename → same inode number** entry.
- The inode's **link count** is incremented.
- Both the original name and the hard link are **indistinguishable** — they both point to the same inode.

(c) FIFO (Named Pipe)

- A **FIFO** is a special file that acts as a pipe for inter-process communication.
- Unlike normal pipes (`|`), FIFOs exist as files in the filesystem.
- Processes can open and read/write like a file.

- `mkfifo` is essentially a **specialized layer on top of `mknod`. So mkfifo is a layer above mknod** because it simplifies usage and avoids needing to specify `S_IFIFO` manually.

(d) mknod

- A **general system call** to create special files (character devices, block devices, FIFOs).

Viva Questions with Answers

Q1. What happens at the inode level when you create a hard link?

👉 Answer: A new directory entry is created pointing to the same inode as the original file. Both entries share the same inode number and data blocks. The link count in the inode is incremented.

Q2. If you delete the original file after creating a soft link, what happens?

👉 Answer: The soft link becomes a dangling link. It still exists but points to a non-existent pathname. Accessing it results in an error (`No such file or directory`).

Q3. Can you create a hard link to a directory? Why or why not?

👉 Answer: Normally no, because it can create cycles in the filesystem and confuse directory traversal (`.` and `..`). Only the superuser can sometimes force it.

Q4. Why do we prefer `mkfifo()` instead of `mknod()` for creating FIFOs today?

👉 Answer: `mkfifo()` is dedicated for FIFO creation and is safer and more portable. `mknod()` is a general system call used for devices, and FIFO creation with `mknod()` is legacy.

Q5. Why is it usually better to give `0666` permissions to FIFOs instead of `0644`?

👉 Answer: Because FIFOs are used for communication between processes. With `0666`, both reader and writer processes (regardless of user/group) can access it. With `0644`, only the owner can write, which limits communication.

what each field in your `ls -l` output means.

Take this line as an example:

```
-rw-r--r--@ 1 adityadave  staff   2166 Sep  6 21:05 1.c
```

1. File type & permissions (`rw-r--r--@`)

- First character → file type:
  - = regular file
  - `d` = directory
  - `l` = symbolic link
  - `p` = named pipe (FIFO)
  - `c` = character device
  - `b` = block device
- Next 9 characters → permissions split into **3 groups of 3**:
  - `rw-` → owner (read, write)
  - `r--` → group (read only)
  - `r--` → others (read only)
- `@` → extended attributes present.

---

2. Link count (`1` , `2` , etc.)

- Number of **hard links** to the file (i.e., directory entries pointing to this inode).
- Example: your `1.c` has `2` links because you created a hard link `h_link` .

---

3. Owner (`adityadave`)

- Username of the file's owner.

---

4. Group (`staff`)

- Group name that owns the file.

---

5. File size (`2166`)

- For regular files: size in **bytes**.
- For special files (devices, FIFOs): may be `0`.

---

6. Timestamp (`Sep 6 21:05`)

- By default: **last modification time** (mtime).
- Can be changed with `ls -lc` (ctime) or `ls -lu` (atime).

---

7. Filename (`1.c` , `mypipe` , etc.)

- The name of the file itself.
- For symbolic links (`l`), it shows `> target`.

Example breakdown of your output:

- `rwxr-xr-x@ 1 adityadave staff 33632 Sep 3 13:56 1`

  → Regular file, executable, size 33632 bytes.

- `drwxr-xr-x@ 8 adityadave staff 256 Aug 19 14:54 FIFO`

  → Directory with 8 links, name `FIFO`.

- `prw-r--r--@ 1 adityadave staff 0 Sep 24 07:27 mypipe`

  → Named pipe (FIFO), size 0.

- `lrwxr-xr-x@ 1 adityadave staff 3 Sep 2 23:01 soft_lnk → 1.c`

  → Symbolic link pointing to `1.c`.

Question 2

Write a simple program to execute in an infinite loop at the background. Go to
/proc directory and
identify all the process related information in the corresponding proc directory.

ANSWER:

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

int main()
{
    do
    {

    } while (1);
    return 0;
}
```

OUTPUT:

```
instead of proc use "top -pid <PID>"
instead of proc use "lsof -p <PID>"

PID   COMMAND      %CPU  TIME     #TH  #WQ  #POR MEM  PURG CMPRS PG
RP PPID STATE   BOOSTS    %CPU_ME
3742  2           100.4 00:23.58 1/1  0    11   753K 0B   496K  3742 2579 running
*0[1]     0.00000

adityadave@Adityas-MacBook-Air-3 Que2 % lsof -p 6924
COMMAND  PID       USER   FD   TYPE DEVICE SIZE/OFF               NODE NAM
E
2       6924 adityadave  cwd    DIR   1,14      128           49553897 /Users/aditya
dave/IIITB/SEM1/SS/HandsOn1/Que2
2       6924 adityadave  txt    REG   1,14    16832           52143324 /Users/aditya
dave/IIITB/SEM1/SS/HandsOn1/Que2/2
2       6924 adityadave  txt    REG   1,14  2289328 1152921500312524246 /usr/li
b/dyld
2       6924 adityadave    0u   CHR   16,3   0t6699               735 /dev/ttys003
```

```
2     6924 adityadave   1u   CHR   16,3   0t6699            735 /dev/ttys003
2     6924 adityadave   2u   CHR   16,3   0t6699            735 /dev/ttys003
```

Concept:

How `/proc` works

- In Linux, `/proc` is a **virtual filesystem** (not stored on disk, created by kernel).
- Every running process has a directory under `/proc/<PID>/`.
- Inside `/proc/<PID>/`, you can see:
  - `status` → process info (name, state, UID, memory, etc.)
  - `cmdline` → command used to start the process
  - `fd/` → file descriptors opened by process
  - `stat` → CPU usage, scheduling info
  - `environ` → environment variables

Viva Questions with Answers

Q1. What is `/proc` in Linux?

👉 Answer: `/proc` is a virtual filesystem that exposes kernel and process information. Each process has a directory `/proc/<PID>` containing details like memory usage, status, file descriptors, and command line arguments.

Q2. If I run your program with `./2 &`, what happens?

👉 Answer: The program executes in the background, the shell prints its PID, and it continues running independently. I can still use the terminal while the process is alive.

Q3. How do you find the PID of your process?

👉 Answer: Either from the shell output after running with `&`, or by using commands like `ps`, `top`, or `pidof <program>`.

Q4. What is the difference between a busy loop and a loop with `sleep(1);`?

👉 Answer: A busy loop consumes 100% CPU continuously, while adding `sleep(1);` yields the CPU, making the process less resource-hungry.

Q5. Since you're on macOS and `/proc` is not available, how did you check process information?

👉 Answer: I used `top -pid <PID>` to monitor CPU usage, and `lsof -p <PID>` to check open files and resources. On Linux, I'd use `/proc/<PID>/`.

`lsof -p <PID>` Output

```
COMMAND  PID      USER   FD   TYPE DEVICE SIZE/OFF             NODE NAME
2       6924 adityadave  cwd    DIR   1,14      128          49553897 /Users/adityadave/IIITB/SEM1/SS/HandsOn1/Que2
2       6924 adityadave  txt    REG   1,14    16832          52143324 /Users/adityadave/IIITB/SEM1/SS/HandsOn1/Que2/2
2       6924 adityadave  txt    REG   1,14  2289328 1152921500312524246 /usr/lib/dyld
2       6924 adityadave    0u   CHR   16,3   0t6699               735 /dev/ttys003
2       6924 adityadave    1u   CHR   16,3   0t6699               735 /dev/ttys003
2       6924 adityadave    2u   CHR   16,3   0t6699               735 /dev/ttys003
```

Column meanings:

- **COMMAND** → Name of the process (`2`).
- **PID** → Process ID (`6924`).
- **USER** → Owner of process (`adityadave`).
- **FD** → File descriptor type:
    - `cwd` → Current working directory.
    - `txt` → Program text (executable code or shared library).
    - `0u` → Standard input (file descriptor 0).

    - `1u` → Standard output (fd 1).
    - `2u` → Standard error (fd 2).
    - `u` → File is open for **read/write**.
- **TYPE** → File type:
    - `DIR` → Directory.
    - `REG` → Regular file.
    - `CHR` → Character device (like terminal).
- **DEVICE** → Device numbers (`major,minor` → e.g., `1,14`).
- **SIZE/OFF** → File size or current file offset.
- **NODE** → File's inode number.
- **NAME** → Path to the file or device.

QUESTION 3

Write a program to create a file and print the file descriptor value. Use creat ( ) system call

ANSWER:

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

int main(int argc, char *argv[])
{
    int filedesc;
    filedesc = creat(argv[1], 0644);
    printf("File descriptor for generated file %s is : %d\n", argv[1], filedesc);
    close(filedesc);
```

```
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que3 % ./3 new_file.txt
File descriptor for generated file new_file.txt is : 3
```

CONCEPT:

What is `creat()` ?

- `creat()` is a system call used to **create a new file** or **truncate an existing file**.
- Signature:

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

- Parameters:
  - `pathname` : Name of the file to create.
  - `mode` : File permissions (e.g., `0644` → owner can read/write, group/others can read).
- Returns:
  - **Non-negative integer** = file descriptor (FD) of the new file.
  - 2 = error.

👉 Internally, `creat()` is equivalent to:

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

So, `creat()` is just a **specialized version of `open()`** for write-only + create/truncate.

File Descriptor (FD)

- A file descriptor is a **small integer** used by the kernel to identify an open file for a process.
- By default:
  - `0` = STDIN
  - `1` = STDOUT
  - `2` = STDERR
- The next file you open gets `3`, then `4`, and so on.

File Permissions (mode like 0644)

- These are **octal values** (base 8).
- They represent **owner / group / others** access.

Each digit = sum of:

- `4` = read (r)
- `2` = write (w)
- `1` = execute (x)

Example:

- `0644` → `rw-r--r--`
  - Owner: read + write (`6`)
  - Group: read-only (`4`)
  - Others: read-only (`4`)

Viva Questions with Answers

Q1. What is the difference between `creat()` and `open()`?

👉 Answer: `creat()` is equivalent to `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)`. It always opens the file in write-only mode and truncates it if it exists. `open()` is more flexible because we can specify flags like `O_RDONLY`, `O_RDWR`, `O_APPEND`, etc.

Q2. What file descriptor value will be returned by `creat()` if no other files are open?

👉 Answer: `3`. Because `0`, `1`, and `2` are reserved for `stdin`, `stdout`, and `stderr`.

Q3. What happens if the file already exists and you call `creat()` on it?

👉 Answer: The existing file is **truncated to zero length** (its contents are lost).

Q4. What is the use of the `mode` argument in `creat()`?

👉 Answer: It specifies the **file permissions** (read/write/execute for owner, group, others). For example, `0644` means owner can read/write, others can only read.

Q5. Why do we need to call `close(filedesc)`?

👉 Answer: To release the file descriptor. Every open file consumes an entry in the kernel's file table. If not closed, it may cause resource leaks.

Question 4

Write a program to open an existing file with read write mode. Try O_EXCL flag also.

ANSWER:

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

int main(int argc, char *argv[])
{
    int filedesc;
    filedesc = open(argv[1], O_RDWR, 0644);
    printf("File descriptor for given opened file is: %d\n", filedesc);
    close(filedesc);

    filedesc = open(argv[1], O_RDWR | O_CREAT | O_EXCL, 0644);
```

```
    printf("File descriptor for given opened file with O_EXCL is : %d\n", filedes
c);
    close(filedesc);

    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que4 % ./4 file2.txt
File descriptor for given opened file is: -1
File descriptor for given opened file with O_EXCL is : 3
adityadave@Adityas-MacBook-Air-3 Que4 % ./4 file2.txt
File descriptor for given opened file is: 3
File descriptor for given opened file with O_EXCL is : -1
```

CONCEPT:

open() system call

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

- **flags** → How the file should be opened:
  - `O_RDONLY`, `O_WRONLY`, `O_RDWR` → read/write access
  - `O_CREAT` → create file if it doesn't exist
  - `O_EXCL` → fail if the file already exists (only works with `O_CREAT`)
- **mode** → File permissions (like `0644`), used only when creating new files.

O_EXCL Behavior

- `O_CREAT | O_EXCL` together → create a new file, **fail if it already exists**.

- Return value:
  - **File descriptor (>=0)** if success
  - **-1** if error (file already exists or permission denied)

Viva Questions with Answers

Q1. What is the purpose of `O_EXCL`?

👉 Answer: It ensures that a new file is created. If the file already exists, `open()` fails and returns `-1`. It prevents accidental overwriting.

Q2. Why does `O_EXCL` only work with `O_CREAT`?

👉 Answer: Because without `O_CREAT`, the system won't create a new file. `O_EXCL` is meaningful only when you're asking to create a new file exclusively.

Q3. In your program, why was the first `open(argv[1], O_RDWR, 0644)` returning `-1` in the first run?

👉 Answer: Because the file didn't exist yet, and I didn't specify `O_CREAT`. The system couldn't open it.

Q4. If you run your program twice, why does the second run succeed in the first `open` but fail in the second `open` with `O_EXCL`?

👉 Answer:

- First `open` succeeds because the file exists now.
- Second `open` with `O_CREAT | O_EXCL` fails because the file already exists, and `O_EXCL` forbids opening an existing file.

Q5. Why do we pass `0644` in the first `open` even though the file is not created?

👉 Answer: In the first `open` call, the `mode` argument is **ignored** because no `O_CREAT` is specified. The mode is only used when a new file is created.

Question 5:

Write a program to create five new files with infinite loop. Execute the program in the background
and check the file descriptor table at /proc/pid/fd.

ANSWER:

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

int main()
{
    int filedesc[5];
    const char *files[] = {"file1.txt", "file2.txt", "file3.txt", "file4.txt", "file5.txt"};
    for (int i = 0; i < 5; i++)
    {
        filedesc[i] = creat(files[i], 0644);
        printf("Created %s with the file descriptor value of : %d\n", files[i], filede
sc[i]);
    }
    while (1)
        ;
    close(filedesc[0]);
    close(filedesc[1]);
    close(filedesc[2]);
    close(filedesc[3]);
    close(filedesc[4]);
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que5 % ./5
Created file1.txt with the file descriptor value of : 3
Created file2.txt with the file descriptor value of : 4
Created file3.txt with the file descriptor value of : 5
Created file4.txt with the file descriptor value of : 6
Created file5.txt with the file descriptor value of : 7

adityadave@Adityas-MacBook-Air-3 Que5 % lsof -p 3010
COMMAND  PID       USER   FD   TYPE DEVICE SIZE/OFF                NODE NAM
E
5       3010 adityadave  cwd    DIR   1,15      288            49576076 /Users/aditya
dave/IIITB/SEM1/SS/HandsOn1/Que5
5       3010 adityadave  txt    REG   1,15    33504            53698706 /Users/aditya
dave/IIITB/SEM1/SS/HandsOn1/Que5/5
5       3010 adityadave  txt    REG   1,15  2289328 1152921500312524573 /usr/li
b/dyld
5       3010 adityadave    0u   CHR   16,1    0t788                 721 /dev/ttys001
5       3010 adityadave    1u   CHR   16,1    0t788                 721 /dev/ttys001
5       3010 adityadave    2u   CHR   16,1    0t788                 721 /dev/ttys001
5       3010 adityadave    3w   REG   1,15        0            49577943 /Users/adityad
ave/IIITB/SEM1/SS/HandsOn1/Que5/file1.txt
5       3010 adityadave    4w   REG   1,15        0            49577944 /Users/adityad
ave/IIITB/SEM1/SS/HandsOn1/Que5/file2.txt
5       3010 adityadave    5w   REG   1,15        0            49577945 /Users/adityad
ave/IIITB/SEM1/SS/HandsOn1/Que5/file3.txt
5       3010 adityadave    6w   REG   1,15        0            49577946 /Users/adityad
ave/IIITB/SEM1/SS/HandsOn1/Que5/file4.txt
5       3010 adityadave    7w   REG   1,15        0            49577947 /Users/adityad
ave/IIITB/SEM1/SS/HandsOn1/Que5/file5.txt
```

CONCEPT:

File Descriptor Table

- Every process has a **file descriptor table** maintained by the kernel.
- By default:
  - `0` = `stdin`
  - `1` = `stdout`
  - `2` = `stderr`
- When you open/create new files, they are assigned the **next available integer** (starting from 3).
- You can check open file descriptors of a process in Linux at:

  ```
  /proc/<PID>/fd/
  ```

  This contains symbolic links to actual files opened by that process.

Viva Questions with Answers

Q1. Why do file descriptors start from 3?

👉 Answer: `0`, `1`, and `2` are reserved for `stdin`, `stdout`, and `stderr`. So the first available FD is `3`.

Q2. What will `/proc/<PID>/fd/` show for this process?

👉 Answer: It will contain symbolic links like:

```
0 → /dev/pts/0   (stdin)
1 → /dev/pts/0   (stdout)
2 → /dev/pts/0   (stderr)
3 → file1.txt
4 → file2.txt
5 → file3.txt
6 → file4.txt
7 → file5.txt
```

(Only if we keep them open).

Q3. What happens when you `close(filedesc[i])`?

👉 Answer: The file descriptor is released back to the kernel. If you open another file later, it may reuse that same FD number.

Q4. Why is the infinite loop required here?

👉 Answer: To keep the process running in the background, so we have time to inspect its `/proc/<PID>/fd` directory or check with `lsof`.

Q5. What happens if you run the program again without deleting the old files?

👉 Answer: `creat()` will truncate the existing files to size 0, and reassign file descriptors again.

Question 6:

Write a program to take input from STDIN and display on STDOUT. Use only read/write system calls.

ANSWER:

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

int main()
{
   char buff[99];
   int count = read(0, buff, 99);
   write(1, buff, count);
   return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que6 % ./6
Hello, I am testing question 6
Hello, I am testing question 6
```

CONCEPT:

read() system call

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

- Reads up to `count` bytes from file descriptor `fd` into `buf`.
- Returns number of bytes actually read.
- For STDIN, `fd = 0`.

write() system call

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

- Writes `count` bytes from buffer `buf` to file descriptor `fd`.
- Returns number of bytes written.
- For STDOUT, `fd = 1`.

Key Point

Here we bypass standard C library (`printf`, `scanf`) and directly use **low-level system calls** (`read`, `write`), which are closer to the kernel.

Viva Questions with Answers

Q1. What is the difference between `read/write` and `scanf/printf`?

👉 Answer: `read/write` are **system calls**, directly invoking the kernel with file descriptors. They work on any file descriptor (files, sockets, pipes).

`scanf/printf` are **library functions** built on top of system calls, with formatting support, buffering, etc.

Q2. Why do we pass `0` in `read()` and `1` in `write()`?

👉 Answer: Because file descriptor `0` is **STDIN**, `1` is **STDOUT**, and `2` is **STDERR`. These are always reserved by the kernel.

Q3. If I type more than 99 characters, what happens?

👉 Answer: Only the first 99 characters are read into the buffer. The rest remain in the input stream and can be read by another call to `read()`.

Q4. Does `read()` automatically add a null character (`\0`) at the end of the buffer?

👉 Answer: No. `read()` just fills raw bytes. Null-termination must be added manually if we want to treat the buffer as a string.

Q5. What will `read()` return when the user presses Ctrl+D (EOF)?

👉 Answer: It will return `0`, meaning **end of file** (no more input).

Question 7:

Write a program to copy file1 into file2 ($cp file1 file2).

ANSWER:

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

int main(int argc, char *argv[])
{
    char buff[1];
    int fd1 = open(argv[1], O_RDONLY);
    int fd2 = open(argv[2], O_WRONLY);
    while (read(fd1, buff, 1) > 0)
    {
```

```
        write(fd2, buff, 1);
    }
    printf("Successfully Copied  Content of file %s to file %s\n", argv[1], argv
[2]);
    close(fd1);
    close(fd2);
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que7 % ./7 main_file.txt copy_file.txt
Successfully Copied  Content of file main_file.txt to file copy_file.txt
```

CONCEPT:

Replicate the behavior of the shell command:

```
cp file1 file2
```

System Calls Involved

1. **open()**
   - `int open(const char *pathname, int flags, mode_t mode);`
   - Used to open existing file1 (with `O_RDONLY`).
   - Used to open/create file2 (with `O_WRONLY | O_CREAT | O_TRUNC`).
2. **read()**
   - Reads from `fd1` into a buffer.
3. **write()**
   - Writes buffer contents into `fd2`.

4. **close()**
    - Closes file descriptors when done.

👉 Typical logic:

- Open source file (read-only).
- Open destination file (write-only, create if needed, truncate if exists).
- Loop with `read()` until EOF, writing contents to destination.
- Close files.

Viva Questions with Answers

Q1. What happens if `copy_file.txt` does not exist in your program?

👉 Answer: Since I used only `O_WRONLY`, `open()` will fail. The correct approach is to use `O_WRONLY | O_CREAT` so the file gets created if missing.

Q2. Why do we use `O_TRUNC` when opening the destination file?

👉 Answer: So that if the destination file already exists, its old content is erased before writing new data. Otherwise, old content might remain if the new copy is smaller.

Q3. Why did you use buffer size = 1? What if you increase it?

👉 Answer: With size = 1, the program reads/writes one byte at a time (very inefficient). If we increase buffer size (say 1024), we perform fewer system calls, which improves performance.

Q4. How does the kernel know when to stop reading from `fd1`?

👉 Answer: When `read()` returns 0, it means **EOF (end of file)** has been reached.

Q5. Can this same program be used to copy binary files (e.g., images)?

👉 Answer: Yes. Since `read()` and `write()` handle raw bytes, the program works for text and binary files alike.

Question 8:

Write a program to open a file in read only mode, read line by line and display each line as it is read. Close the file when end of file is reached.

ANSWER:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int fd = open(argv[1], O_RDONLY);
    char buff[256];
    char line[1024];
    int n, i, line_index = 0;

    while ((n = read(fd, buff, sizeof(buff))) > 0)
    {
        for (i = 0; i < n; i++)
        {
            line[line_index++] = buff[i];

            if (buff[i] == '\n' || line_index == sizeof(line) - 1)
            {
                write(1, line, line_index);
                line_index = 0;
            }
        }
    }

    if (line_index > 0)
        write(1, line, line_index);

    close(fd);
```

```
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que8 % ./8 read_file.txt
Reading Line 1
Reading Line 2
Reading Line 3
Reading Line 4
Ending.
```

CONCEPT:

System Calls

- `open(path, O_RDONLY)` → open file for reading.
- `read(fd, buf, size)` → read characters from the file.
- `write(1, buf, size)` → write them to STDOUT.
- Loop until `read()` returns 0 (EOF).
- Close file with `close(fd)`.

Viva Questions with Answers

Q1. How does your program detect the end of the file?

👉 Answer: When `read()` returns 0, it means EOF has been reached, so the loop exits.

Q2. You read one byte at a time. Is this efficient?

👉 Answer: No, it's inefficient because it makes a system call for every character. A larger buffer (like 512 or 1024 bytes) would reduce system calls and be faster.

Q3. How can you modify your program to actually print line by line?

👉 Answer: By reading into a buffer and checking for the newline character (`\n`). Once a newline is detected, we can print that as a line before continuing.

Q4. What will happen if the file has no newline characters at all?

👉 Answer: The program will still read and display all characters, but it won't break into separate lines. The output will be continuous.

Q5. What's the difference between `read()` and `fgets()`?

👉 Answer: `read()` is a low-level system call that works with file descriptors and raw bytes. `fgets()` is a C library function that works with `FILE*` streams and reads formatted data (like line by line with buffering).

Purpose of `fgets`

`fgets` reads **up to** `n-1` **characters** from a file stream into a buffer **until it hits a newline (**`\n`**) or EOF**. It also **adds a null terminator** `\0` at the end of the string.

```
char *fgets(char *str, int n, FILE *stream);
```

- `str`: buffer to store data
- `n`: max number of characters to read (including `\0`)
- `stream`: the file pointer

Question 9:

Write a program to print the following information about a given file.
a. inode
b. number of hard links
c. uid
d. gid
e. size
f. block size
g. number of blocks
h. time of last access

i. time of last modification
j. time of last change

ANSWER:

```
#include <stdio.h>
#include <sys/stat.h>
#include <time.h>

int main(int argc, char *argv[])
{
    struct stat file_stat_q9;
    stat(argv[1], &file_stat_q9);

    printf("Inode of file %s is: %ld\n", argv[1], file_stat_q9.st_ino);
    printf("Number of hard link file %s have is : %ld\n", argv[1], file_stat_q9.st_n
link);
    printf("User Id of file %s is : %d\n", argv[1], file_stat_q9.st_uid);
    printf("Group Id of file %s is : %d\n", argv[1], file_stat_q9.st_gid);
    printf("Size of the file %s is : %ld bytes\n", argv[1], file_stat_q9.st_size);
    printf("Block size of the file %s is : %lld\n", argv[1], file_stat_q9.st_blksize);
    printf("No of Blocks of the file %s is : %lld\n", argv[1], file_stat_q9.st_block
s);
    printf("\n");
    printf("Last access happeneded on file %s : %s\n", argv[1], ctime(&file_stat
_q9.st_atime));
    printf("Last modification done on file %s : %s\n", argv[1], ctime(&file_stat_q
9.st_mtime));
    printf("Last change happeneded on file %s : %s\n", argv[1], ctime(&file_stat
_q9.st_ctime));
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que9 % ./9 file.txt
Inode of file file.txt is: 49632556 (ls -i 9.c)
Number of hard link file file.txt have is : 3
User Id of file file.txt is : 501
Group Id of file file.txt is : 20
Size of the file file.txt is : 5 bytes
Block size of the file file.txt is : 4096
No of Blocks of the file file.txt is : 8

Last access happeneded on file file.txt : Mon Aug 18 22:59:13 2025
Last modification done on file file.txt : Sat Aug 16 18:50:18 2025

Last change happeneded on file file.txt : Mon Aug 18 22:59:13 2025
```

CONCEPT:

When you call `stat(pathname, &buf);`, the kernel fills `buf` with metadata about the file. Key fields:

1. **`st_ino` (Inode number)**
   - Unique identifier for a file in the filesystem.
   - All hard links to the same file share the same inode number.
2. **`st_nlink` (Number of hard links)**
   - Tells how many directory entries point to this inode.
   - File data is deleted only when this count drops to 0.
3. **`st_uid` (User ID of owner)**
   - The ID of the user who owns the file.
   - Can be mapped to username with `getpwuid()`.
4. **`st_gid` (Group ID of owner)**
   - The group that owns the file.

- Can be mapped to group name with `getgrgid()`.

5. `st_size` **(File size in bytes)**
   - The total size of the file content.
   - For regular files → exact size in bytes.
   - For directories → may store size of directory entries.
6. `st_blksize` **(Block size)**
   - The preferred block size for efficient I/O operations.
   - Not the actual file size — more about how the OS reads/writes data.
7. `st_blocks` **(Number of allocated blocks)**
   - Number of disk blocks (usually 512 bytes each) allocated to the file.
   - Can be larger than `st_size/512` due to block rounding.
   - Sparse files may use fewer physical blocks than `st_size` suggests.
8. `st_atime` **(Last access time)**
   - Time when the file was last read.
   - Example: running `cat file.txt` updates `atime`.
9. `st_mtime` **(Last modification time)**
   - Time when file contents were last modified.
   - Example: editing file updates `mtime`.
10. `st_ctime` **(Last status change time)**
    - Time when file's metadata changed (permissions, owner, link count).
    - ❌ Not creation time (common confusion in viva).

Viva Questions with Answers

Q1. What is an inode?

👉 Answer: An inode is a data structure in the filesystem that stores metadata about a file — such as size, permissions, owner, timestamps, and data block locations. The filename itself is not stored in the inode but in the directory entry.

Q2. What is the difference between `st_mtime` and `st_ctime`?

👉 Answer:

- `st_mtime` → last modification time (file content changed).
- `st_ctime` → last status change (metadata changed, e.g., permissions, ownership, link count). It does **not** mean file creation time.

Q3. Why can the number of blocks (`st_blocks`) be larger than file size?

👉 Answer: Because disk allocation happens in blocks. Even a 1-byte file will consume at least 1 block. Sparse files may also show allocated blocks differently.

Q4. If a file has 3 hard links, what does it mean?

👉 Answer: It means there are 3 directory entries pointing to the same inode. The file data is deleted only when all links are removed (link count = 0).

Q5. How do you find the file's owner name from `st_uid`?

👉 Answer: Use `getpwuid(file_stat.st_uid)` from `<pwd.h>` to get the username. Similarly, `getgrgid(file_stat.st_gid)` for group name.

Question 10:

Write a program to open a file with read write mode, write 10 bytes, move the file pointer by 10
bytes (use lseek) and write again 10 bytes.
a. check the return value of lseek
b. open the file with od and check the empty spaces in between the data.

ANSWER:

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

```
int main()
{
    int fd = open("lseek.txt", O_RDWR);

    char first_10_bytes[10] = "ADITYADAVE";
    char next_10_bytes[10] = "DAVEADITYA";

    off_t pos0 = lseek(fd, 0, SEEK_CUR);
    printf("Lseek position : %lld\n", pos0);

    write(fd, first_10_bytes, 10);

    off_t pos1 = lseek(fd, 10, SEEK_CUR);
    // off_t pos = lseek(fd, 10, SEEK_SET); for starting

    printf("Lseek position : %lld\n", pos1);

    write(fd, next_10_bytes, 10);

    off_t pos2 = lseek(fd, 0, SEEK_CUR);
    printf("Lseek position : %lld\n", pos2);

    printf("File content of lseek.txt: \n");
    int fd2 = open("lseek.txt", O_RDONLY);
    char buff[1];
    while (read(fd2, buff, 1) > 0)
    {
        write(1, buff, 1);
    }
    printf("\n");
    close(fd2);
    close(fd);
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que10 % ./10
Lseek position : 0
Lseek position : 20
Lseek position : 30
File content of lseek.txt:
ADITYADAVE          DAVEADITYA
adityadave@Adityas-MacBook-Air-3 Que10 % od -c lseek.txt

0000000    A   D   I   T   Y   A   D   A   V   E  \0  \0  \0  \0  \0  \0
0000020   \0  \0  \0  \0   D   A   V   E   A   D   I   T   Y   A
0000036
```

CONCEPT:

lseek() system call

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

- **fd** → file descriptor
- **offset** → number of bytes to move
- **whence** → reference point:
  - `SEEK_SET` → from beginning of file
  - `SEEK_CUR` → from current position
  - `SEEK_END` → from end of file

👉 `lseek()` moves the file offset (where next read/write will happen).

👉 Return value = new offset in the file.

File holes (sparse files)

- If you `lseek()` **beyond the end of file** and then write, the gap is filled with **holes (zero bytes)**.
- These do not consume disk blocks (filesystem optimization).
- You can verify holes using `od -c` or `od -x` (hexdump).

Viva Questions with Answers

Q1. What does `lseek(fd, 10, SEEK_CUR)` do?

👉 Answer: It moves the file offset 10 bytes forward from the current position, without reading or writing anything.

Q2. What happens to the skipped bytes when you `lseek()` beyond current end and then write?

👉 Answer: The skipped region becomes a **file hole** filled with zero bytes logically, but not stored on disk (sparse file).

Q3. How do you verify the presence of holes?

👉 Answer: By using `od -c filename`, which shows `\0` (null characters) for the hole. Printing with `cat` or `printf` will just display spaces or nothing visible.

Q4. What is the difference between `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`?

👉 Answer:

- `SEEK_SET` → set position relative to file start.
- `SEEK_CUR` → move relative to current position.
- `SEEK_END` → move relative to end of file.

Q5. Why does `lseek()` return an offset instead of just success/failure?

👉 Answer: Because it not only repositions the file offset but also tells you the new absolute position, which is often useful.

Question 11:

Write a program to open a file, duplicate the file descriptor and append the file with both the
descriptors and check whether the file is updated properly or not.
a. use dup
b. use dup2
c. use fcntl

ANSWER:

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int filedesc = open(argv[1], O_RDWR);
    printf("original Value of File descriptor is : %d\n", filedesc);

    int dup_filedesc = dup(filedesc);
    printf("New Value of file descriptor using dup() system call is : %d\n", dup_filedesc);

    int dup2_filedesc = dup2(filedesc, 7);
    printf("New Value of file descriptor using dup2() system call is : %d\n", dup2_filedesc);

    int fcntl_filedesc = fcntl(filedesc, F_DUPFD, 0);
    printf("New Value of file descriptor using fcntl() system call is : %d\n", fcntl_filedesc);

    write(filedesc, "Hello from original FD\n", 23);
    write(dup_filedesc, "Hello from dup FD\n", 19);
    write(dup2_filedesc, "Hello from dup2 FD\n", 20);
    write(fcntl_filedesc, "Hello from fcntl FD\n", 21);

    close(filedesc);
```

```
    close(dup_filedesc);
    close(dup2_filedesc);
    close(fcntl_filedesc);

    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que11 % ./11 file.txt
original Value of File descriptor is : 3
New Value of file descriptor using dup() system call is : 4
New Value of file descriptor using dup2() system call is : 7
New Value of file descriptor using fcntl() system call is : 5
```

CONCEPT:

File Descriptors

- When a file is opened, the kernel returns a **file descriptor (FD)** — a small integer index in the process's file table.
- Multiple FDs can point to the same open file description (same offset, same file status flags).

---

dup()

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

- Creates a duplicate FD of `oldfd`.
- Returns the **lowest available FD number**.
- Both FDs share the same file offset.

---

dup2()

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

- Duplicates `oldfd` into `newfd`.
- If `newfd` is already open, it is first closed.
- Useful when you want to force a specific FD (e.g., redirect `stdout` to a file).

---

fcntl() with F_DUPFD

```
int fcntl(int oldfd, F_DUPFD, int minfd);
```

- Duplicates `oldfd` into a new FD number **greater than or equal to minfd**.
- More flexible than `dup`.

👉 All three (`dup`, `dup2`, `fcntl`) create **new FD entries** pointing to the **same open file description**. So they share:

- Same file offset
- Same file flags

Viva Questions with Answers

Q1. What is the difference between `dup` and `dup2`?

👉 Answer: `dup` returns the lowest unused FD, while `dup2` lets us specify the new FD number. If that FD is already open, `dup2` closes it first.

Q2. What advantage does `fcntl(F_DUPFD)` give over `dup`?

👉 Answer: `fcntl(F_DUPFD, minfd)` ensures the new FD is ≥ `minfd`. This gives more control over FD allocation, while `dup` always picks the lowest available FD.

Q3. Do these duplicated FDs share the same file offset?

👉 Answer: Yes. All FDs created by `dup`, `dup2`, or `fcntl` point to the same **open file description**, so they share offset and flags. If one FD moves the offset, the others see it too.

Q4. If you close the original FD, can you still use the duplicated FD?

👉 Answer: Yes. Closing one FD does not affect others. The file remains open until all FDs pointing to it are closed.

Q5. Why might `dup2` be useful in practice?

👉 Answer: It's commonly used for I/O redirection in shells. For example, redirecting `stdout (fd=1)` to a file by calling `dup2(filefd, 1)`.

How Linux Stores File Descriptors Internally

There are **three main layers**:

1. **File Descriptor Table (per process)**
    - Each process has a table: `fd → open file description`.
    - Example:

FD	Points to (open file description)
3	inode 12345, offset 0
4	inode 12345, offset 0

2. **Open File Description (in kernel)**
    - Contains:
        - Pointer to **inode** (metadata about the file)
        - **File offset** (current position for reads/writes)
        - **File status flags** (O_RDONLY, O_WRONLY, etc.)
    - Multiple FDs can **point to the same open file description** (shared offset).
3. **Inode (filesystem)**
    - Contains file metadata: size, permissions, data blocks, etc.
    - **FD → Open File Description → Inode**

Question 12

Write a program to find out the opening mode of a file. Use fcntl.

ANSWER:

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

int main(int argc, char *argv[])
{
    int fd = open(argv[1], O_RDONLY);

    int flag = fcntl(fd, F_GETFL);
    printf("Original flags: %d\n", flag);

    int check = flag & O_ACCMODE;
    if (check == 0)
        printf("Current Mode is Read Only Mode\n");
    else if (check == 1)
        printf("Current Mode is Write Only Mode\n");
    else if (check == 2)
        printf("Current Mode is Read Write Mode\n");
    close(fd);
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que12 % ./12 file.txt
Original flags: 0
Current Mode is Read Only Mode
```

CONCEPT:

fcntl()

`fcntl()` is a system call that performs various operations on file descriptors.

Here we use it to **get file status flags**.

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

- `F_GETFL` → gets the file access mode and status flags (set at `open()`).

File Access Mode Macros

- When you call `open()`, you specify mode (`O_RDONLY`, `O_WRONLY`, `O_RDWR`).
- These are stored in the FD's flag field.
- To extract mode → mask with `O_ACCMODE`.

```
flags = fcntl(fd, F_GETFL);
mode = flags & O_ACCMODE;
```

Values:

- `O_RDONLY` = 0
- `O_WRONLY` = 1
- `O_RDWR` = 2

Viva Questions with Answers

Q1. What does `F_GETFL` do in `fcntl`?

👉 Answer: It returns the file status flags of the file descriptor, including the access mode and options like `O_APPEND`, `O_NONBLOCK`, etc.

Q2. Why do we mask the result with `O_ACCMODE`?

👉 Answer: Because the flag field contains both access mode bits and other options. `O_ACCMODE` is a mask to extract only the access mode.

Q3. What values correspond to `O_RDONLY`, `O_WRONLY`, `O_RDWR`?

👉 Answer:

- `O_RDONLY = 0`
- `O_WRONLY = 1`
- `O_RDWR = 2`

Q4. If you opened the file with `open(argv[1], O_WRONLY | O_APPEND)`, what would `fcntl(F_GETFL)` return?

👉 Answer: It would return a flag field where `O_WRONLY` is set and also the `O_APPEND` bit is set. So after masking with `O_ACCMODE`, the mode would be write-only, but the raw flag value would also contain `O_APPEND`.

Q5. What is the difference between `F_GETFL` and `F_SETFL`?

👉 Answer:

- `F_GETFL` → gets the current flags.
- `F_SETFL` → changes the status flags (like enabling `O_APPEND` or `O_NONBLOCK`).

Question 13:

Write a program to wait for a STDIN for 10 seconds using select. Write a proper print statement to
verify whether the data is available within 10 seconds or not (check in $man 2 select).

ANSWER:

```
#include <stdio.h>
#include <sys/select.h>
```

```
int main()
{
    fd_set fd;
    struct timeval Ten_sec_timeout = {10, 0};

    FD_ZERO(&fd);
    FD_SET(0, &fd);

    if (select(1, &fd, NULL, NULL, &Ten_sec_timeout))
        printf("User entered data within 10 seconds.\n");
    else
        printf("No data was entered on the terminal within 10 seconds.\n");
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que13 % ./13
Hi
User entered data within 10 seconds.
Hi
```

CONCEPT:

select() system call

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct ti
meval *timeout);
```

- Used for **I/O multiplexing** → lets you wait on multiple file descriptors to see if they're ready for read/write/exception.
- `fd_set` is a bitmask that represents a group of file descriptors.

- `0` is the file descriptor for **standard input (stdin)**.
- After this, the `fd` set contains only **stdin**.
- Parameters:
  - `nfds` → the highest FD + 1 (since FD sets are arrays indexed by FD number).

Why `max_fd + 1` in `select()`

Function prototype:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, str
uct timeval *timeout);
```

  - `nfds` → The **highest-numbered file descriptor you want to watch, plus 1**.
  - **Why +1?**
    - File descriptors are **0-indexed** (0,1,2,...).
    - The kernel loops internally from **0 to nfds-1**.
    - If the highest FD you want to monitor is 4, you must pass `nfds = 5`.
  - `readfds` → set of FDs to check for readability.
  - `writefds` → set of FDs to check for writability.
  - `exceptfds` → set of FDs to check for exceptional conditions.
  - `timeout` → how long to wait (in `struct timeval`).

👉 Return value:

- `>0` → number of ready descriptors.
- `0` → timeout expired (no activity).
- `1` → error.

This question

- We only care about **STDIN** → FD = 0.

- Timeout = **10 seconds**.
- If user types something within 10 seconds, `select()` reports activity.
- Otherwise, it times out.

Viva Questions with Answers

Q1. What is the purpose of `FD_ZERO` and `FD_SET`?

👉 Answer: `FD_ZERO` initializes the set of file descriptors (clears it). `FD_SET(fd, &set)` adds a given file descriptor to that set.

Q2. Why is `select()` called with `nfds = 1` here?

👉 Answer: Because `nfds` should be the **highest file descriptor in the set + 1**. Here the only FD is `0` (stdin), so `nfds = 0+1 = 1`.

Q3. What happens if the user types input before 10 seconds?

👉 Answer: `select()` immediately returns `>0`, and the program prints that data was entered.

Q4. What happens if the user does nothing for 10 seconds?

👉 Answer: `select()` returns 0, meaning timeout, and the program prints that no data was entered.

Q5. Can `select()` monitor multiple file descriptors at once?

👉 Answer: Yes, it can monitor sets of descriptors for readability, writability, and exceptions simultaneously. That's why it's used in networking (e.g., handling multiple sockets).

Fields of `select()`

`select()` watches **three sets of file descriptors**:

Parameter	Purpose	Example
`fd_set *readfds`	FDs to check if **data is available to read**	stdin, sockets
`fd_set *writefds`	FDs to check if **ready for writing**	sockets, pipes

Parameter	Purpose	Example
`fd_set *exceptfds`	FDs to check for **exceptional conditions**	OOB data on sockets
`nfds`	**Highest FD + 1**	explained above
`timeout`	Max time to wait for readiness	struct timeval

- Return value:
  - 0 → number of ready FDs
  - 0 → timeout
  - 1 → error

---

3. `fd_set` Macros

`fd_set` is a **bitmask** representing FDs.

Macro	Purpose
`FD_ZERO(&fd)`	Initialize the set to empty
`FD_SET(fd, &fdset)`	Add FD to the set
`FD_CLR(fd, &fdset)`	Remove FD from the set
`FD_ISSET(fd, &fdset)`	Check if FD is ready after `select()`

Example:

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(0, &readfds);

select(1, &readfds, NULL, NULL, &timeout);

if (FD_ISSET(0, &readfds))
    printf("stdin is ready to read\n");
```

- `FD_ISSET` is **only valid after `select()` returns**.

---

4. `timeval` Structure

`timeval` specifies **how long `select()` should wait:**

```
struct timeval {
    long tv_sec;   // seconds
    long tv_usec;  // microseconds (1 millionth of a second)
};
```

Example:

```
struct timeval timeout = {10, 0};  // 10 seconds
```

- **tv_sec** → seconds to wait
- **tv_usec** → microseconds to wait (0–999999)
- Passing `NULL` → wait **forever**
- Passing `{0,0}` → **poll** (non-blocking check)

---

5. How `select()` Works Internally (High Level)

1. Kernel scans **all FDs from 0 → nfds-1**.
2. Checks **read/write/exception sets**.
3. If any FD is ready, return immediately.
4. If none are ready, wait up to **timeout**.
5. Updates the `fd_set`s to **only include ready FDs**.

Question 14:

Write a program to find the type of a file.
a. Input should be taken from command line.
b. program should be able to identify any type of a file.

ANSWER:

```
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    struct stat file_type;
    stat(argv[1], &file_type);
    if (S_ISREG(file_type.st_mode))
        printf("File %s is regular file.\n", argv[1]);
    else if (S_ISDIR(file_type.st_mode))
        printf("File %s is directory.\n", argv[1]);
    else if (S_ISCHR(file_type.st_mode))
        printf("File %s is character device.\n", argv[1]);
    else if (S_ISBLK(file_type.st_mode))
        printf("File %s is block file.\n", argv[1]);
    else if (S_ISFIFO(file_type.st_mode))
        printf("File %s is fifo pipeline file.\n", argv[1]);
    else if (S_ISLNK(file_type.st_mode))
        printf("File %s is symbolic link.\n", argv[1]);
    else if (S_ISSOCK(file_type.st_mode))
        printf("File %s is socket file.\n", argv[1]);
    else
        printf("%s is of unknown type.\n", argv[1]);
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que14 % ./14 myfifo
```

```
File myfifo is fifo pipeline file.
```

CONCEPT:

File types in Unix/Linux

Every file in Linux is represented by an **inode**, and the `st_mode` field of `struct stat` contains both **permissions** and **file type information**.

We use **macros** on `st_mode` to test file type:

- `S_ISREG(m)` → Regular file (e.g., `.txt`, `.c`)
- `S_ISDIR(m)` → Directory
- `S_ISCHR(m)` → Character device (e.g., `/dev/tty`)
- `S_ISBLK(m)` → Block device (e.g., `/dev/sda`)
- `S_ISFIFO(m)` → Named pipe (FIFO)
- `S_ISLNK(m)` → Symbolic link
- `S_ISSOCK(m)` → Socket

---

stat() vs lstat()

- `stat(path, &st)` → follows symbolic links. If `argv[1]` is a symlink, you get info about the **target**.
- `lstat(path, &st)` → gets info about the **link itself** (so it will show type as symlink).

Viva Questions with Answers

Q1. How does the program determine the file type?

👉 Answer: By calling `stat()` on the file and checking the `st_mode` field using macros like `S_ISREG`, `S_ISDIR`, etc.

Q2. What is the difference between `stat()` and `lstat()`?

👉 Answer: `stat()` follows symbolic links and gives information about the target file, while `lstat()` gives information about the link itself.

Q3. Give an example of a character device and a block device.

👉 Answer:

- Character device → `/dev/tty` (terminal), `/dev/null`
- Block device → `/dev/sda` (hard disk), `/dev/mmcblk0` (SD card)

Q4. What command in Linux shows file type similar to this program?

👉 Answer: `ls -l` shows the first character:

- regular file
- `d` directory
- `c` character device
- `b` block device
- `p` FIFO
- `l` symlink
- `s` socket

Q5. Can a file be both a block device and a character device?

👉 Answer: No. A device node is either block-oriented (access in chunks, like disks) or character-oriented (stream access, like keyboard).

Question 15:

Write a program to display the environmental variable of the user (use environ).

ANSWER:

```
#include <stdio.h>
extern char **environ;
int main()
```

```
{
    char **ans = environ;
    printf("Environment variables are:\n");
    while (*ans != NULL)
    {
        printf("%s\n", *ans);
        ans++;
    }
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que15 % ./15
Environment variables:
MallocNanoZone=0
COMMAND_MODE=unix2003
__CFBundleIdentifier=com.microsoft.VSCode
PATH=/opt/homebrew/opt/llvm/bin:/opt/homebrew/opt/coreutils/libexec/gnub
in:/opt/homebrew/opt/coreutils/libexec/gnubin:/opt/homebrew/opt/coreutils/li
bexec/gnubin:/opt/homebrew/opt/coreutils/libexec/gnubin:/opt/homebrew/op
t/coreutils/libexec/gnubin:/opt/homebrew/opt/coreutils/libexec/gnubin:/opt/ho
mebrew/opt/coreutils/libexec/gnubin:/opt/homebrew/opt/coreutils/libexec/gn
ubin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/homebrew/bin:/opt/homeb
rew/lib/python3.13/site-packages/pip:/Users/adityadave/Library/Python/3.9/bi
n:/Users/adityadave/.local/bin
SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.hvEW9nfM4q/Listeners
HOME=/Users/adityadave
SHELL=/bin/zsh
// There are more but I removed.
```

CONCEPTS:

In memory:

```
environ ──▶ ["PATH=/usr/bin:/bin",
            "HOME=/home/alice",
            "SHELL=/bin/bash",
            "USER=alice",
            NULL]
```

📖 Code Flow

1. `char **ans = environ;`

   → copy pointer to start of environment array.

2. `while (*ans != NULL)`

   → loop through until we hit NULL terminator.

3. `printf("%s\n", *ans);`

   → print each `"NAME=value"` string.

4. `ans++;`

   → move to the next environment variable.

Environment Variables

- Environment variables are key–value pairs maintained by the shell and passed to processes.
- `extern` means: *"this variable is defined somewhere else, I'm just using it here"*.
- Examples:
  - `PATH=/usr/bin:/bin`
  - `HOME=/home/aditya`
  - `USER=adityadave`

They are inherited by child processes when you run a program.

Accessing Environment Variables

- Every process gets a pointer to its **environment table**, usually exposed as:

```
extern char **environ;
```

- This is an array of strings (`char*`), each of the form `KEY=VALUE`.
- The array is null-terminated (`NULL` at the end).

Viva Questions with Answers

Q1. What is the difference between `argv` and `environ`?

👉 Answer: `argv` holds command-line arguments passed to the program, while `environ` holds environment variables inherited from the parent process.

Q2. How are environment variables usually set?

👉 Answer: They are set in the shell (e.g., `export VAR=value`) and then inherited by child processes.

Q3. Give an example of an important environment variable.

👉 Answer: `PATH` (tells the shell where to look for executables).

Q4. What is another way to access environment variables in C apart from `environ`?

👉 Answer: Using `getenv("VAR_NAME")` to fetch the value of a specific variable.

Q5. If you modify `environ` inside your program, does it affect the parent shell?

👉 Answer: No. Changes apply only to the current process and its children. The parent shell's environment is unaffected.

Question 16:

Write a program to perform mandatory locking.
a. Implement write lock
b. Implement read lock

ANSWER:

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    if (strcmp(argv[1], "read_lock") == 0)
    {
        int fd = open(argv[2], O_RDONLY);
        struct flock fl = {0};
        fl.l_type = F_RDLCK;
        fl.l_whence = SEEK_SET;
        fl.l_start = 0;
        fl.l_len = 0;

        fcntl(fd, F_SETLKW, &fl);

        puts("Read lock is active if you want to release lock just press any key.");
        getchar();

        fl.l_type = F_UNLCK;
        fcntl(fd, F_SETLK, &fl);
        close(fd);
    }
    else if (strcmp(argv[1], "write_lock") == 0)
    {
        int fd = open(argv[2], O_RDWR);
        struct flock file_lock = {0};
        file_lock.l_type = F_WRLCK;
        file_lock.l_whence = SEEK_SET;
        file_lock.l_start = 0;
        file_lock.l_len = 0;
```

```
        fcntl(fd, F_SETLKW, &file_lock);

        puts("Write lock is active if you want to release lock just press any key.");
        getchar();

        file_lock.l_type = F_UNLCK;
        fcntl(fd, F_SETLK, &file_lock);
        close(fd);
    }
    return 0;
}
```

OUTPUT:

```
Terminal 1:
adityadave@Adityas-MacBook-Air-3 Que16 % ./16 write_lock lockdemo.txt
Write lock held. Press Enter to unlock...
Terminal 2:
adityadave@Adityas-MacBook-Air-3 Que16 %  ./16 write_lock lockdemo.txt
```

CONCEPT:

File Locking in Unix/Linux

Two kinds of locks:

1. **Advisory Locking** (default in Linux)
    - Processes must **cooperate** (all must use `fcntl()` / `flock()` to check locks).
    - If one ignores locks and does raw `write()`, it can still modify the file.
2. **Mandatory Locking** (special case)
    - Kernel enforces locking.

- If one process locks a file, others cannot read/write without waiting for the lock to be released.
- Must be explicitly enabled on the file.

Enabling Mandatory Locking

1. File must have **group-ID bit (SGID)** set and **group-execute bit cleared**.

   ```
   chmod g+s,g-x filename
   ```

   (This means: set SGID bit and remove group execute bit).

2. Then `fcntl()` locking becomes mandatory (kernel enforces it).

---

fcntl() with struct flock

```
struct flock {
    short l_type;    // F_RDLCK, F_WRLCK, F_UNLCK
    short l_whence;  // SEEK_SET, SEEK_CUR, SEEK_END
    off_t l_start;   // starting offset
    off_t l_len;     // number of bytes (0 → till EOF)
    pid_t l_pid;     // process ID of the lock holder
};
```

- **F_SETLK** → set lock (non-blocking, fail if cannot acquire).
- **F_SETLKW** → set lock and wait until available.
- **F_GETLK** → check lock.
- **l_type** → kind of lock (`F_RDLCK`, `F_WRLCK`, `F_UNLCK`)
- **l_whence, l_start, l_len** → specify byte range of lock (you can lock only a part of a file!)
- **l_pid** → which process holds the lock (for conflict detection).
- `l_type` → what kind of lock (read/write/unlock).
- `l_whence` → reference point (beginning, current, or end of file).

- `l_start` → where to begin locking.
- `l_len` → how many bytes (0 = until EOF).

This Program

- If run with `"read_lock"`, applies a **shared lock** (`F_RDLCK`). Multiple readers allowed unless a writer holds the lock.
- If run with `"write_lock"`, applies an **exclusive lock** (`F_WRLCK`). Only one writer allowed, blocks others.
- Unlocks when pressing a key.

Viva Questions with Answers

Q1. What is the difference between advisory and mandatory locking?

👉 Advisory requires cooperation of processes. Mandatory is enforced by kernel (others are blocked even if they don't check).

Q2. How do you enable mandatory locking on a file in Linux?

👉 By setting the SGID bit and removing group-execute permission:

```
chmod g+s,g-x filename
```

Q3. What's the difference between `F_SETLK` and `F_SETLKW`?

👉 `F_SETLK` tries to acquire lock immediately (non-blocking). If unavailable, it fails. `F_SETLKW` waits (blocks) until lock is available.

Q4. Can multiple processes hold read locks simultaneously?

👉 Yes. Read locks are **shared**. But if a write lock exists, no read lock can be acquired, and vice versa.

Q5. What happens if a process holding a lock terminates?

👉 All locks held by that process are automatically released by the kernel.

Question 17:

Write a program to simulate online ticket reservation. Implement write lock Write a program to open a file, store a ticket number and exit. Write a separate program, to open the file, implement write lock, read the ticket number, increment the number and print the new ticket number then close the file.

ANSWER:

17.c

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

int main()
{
    struct
    {
        int ticket_no;
    } database;

    struct flock lock;
    int fd = open("database", O_RDWR);

    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;

    printf("This line is executed before entering into critical section\n");
    fcntl(fd, F_SETLKW, &lock);

    printf("We are currently in critical section\n");

    lseek(fd, 0, SEEK_SET);
```

```
    read(fd, &database, sizeof(database));
    printf("Ticket number as of now :%d\n", database.ticket_no);

    database.ticket_no++;

    lseek(fd, 0, SEEK_SET);
    write(fd, &database, sizeof(database));
    fsync(fd);

    printf("New ticket number written: %d\n", database.ticket_no);

    printf("Press Enter to release the lock\n");
    getchar();

    lock.l_type = F_UNLCK;
    fcntl(fd, F_SETLK, &lock);

    printf("We are out of critical section.\n");

    close(fd);
}
```

db.c

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

int main()
{
    int fd;
    struct
    {
        int ticket_no;
```

```
    } database;
    database.ticket_no = 0;
    fd = open("database", O_RDWR | O_CREAT | O_EXCL, 0744);
    write(fd, &database, sizeof(database));
    close(fd);
    fd = open("database", O_RDONLY);
    read(fd, &database, sizeof(database));
    printf("Ticket no: %d\n", database.ticket_no);
    close(fd);
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que17 % ./db
Ticket no: 0
adityadave@Adityas-MacBook-Air-3 Que17 % ./inc
Before entering into critical section
Inside the critical section
Current ticket number: 1
New ticket number written: 2
Press Enter to unlock...

Exited critical section
```

CONCEPT:

Critical Section Problem

- When multiple processes access and modify **shared data** (here: the ticket number in a file), race conditions can occur.
- Example: Two users run the reservation program simultaneously → both read the same ticket number (say `5`) before incrementing, then both write `6` → result is wrong (lost update).

👉 To prevent this, we use **locks** so only one process enters the **critical section** at a time.

Solution using File Locking

1. Maintain a database file (`database`) with an integer ticket number.
2. Use **write lock (exclusive lock)** when a process wants to update it.
3. Inside critical section:
    - Read ticket number.
    - Increment it.
    - Write back updated value.
4. Release lock.

This ensures correctness even if multiple processes run simultaneously.

Two Programs

- `db.c` → Initializes the database file with ticket number = 0.
- `inc.c` (main one) → Simulates a user booking:
    - Acquires write lock.
    - Reads + increments ticket number.
    - Writes back.
    - Releases lock.

Viva Questions with Answers

Q1. Why do we need file locking here?

👉 To prevent race conditions where multiple processes access and modify the ticket number simultaneously, causing lost updates.

Q2. Why do we use a write lock instead of a read lock?

👉 Because we are modifying the file (read + update). Write locks are **exclusive**, preventing other processes from even reading during the update.

Q3. What does `fcntl(fd, F_SETLKW, &lock)` do?

👉 It tries to set the lock. If the lock is unavailable, it **waits (blocks)** until it can acquire it.

Q4. What would happen if we used `F_SETLK` instead?

👉 `F_SETLK` is non-blocking. If lock unavailable, it fails immediately. This could cause failed reservations instead of queued waiting.

Q5. Why do we use `fsync(fd)` after writing?

👉 To force the OS to flush changes from buffer to disk, ensuring the updated ticket number is not lost if the program or system crashes.

Q6. How does Linux know when to release the lock?

👉 Locks are automatically released if:

- The process explicitly unlocks (`F_UNLCK`).
- Or the process terminates.

Question 18:

Write a program to perform Record locking.
a. Implement write lock
b. Implement read lock
Create three records in a file. Whenever you access a particular record, first lock it then modify/access to avoid race condition.

ANSWER:

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

```
struct Course
{
    int course;
    int number_of_students;
};

int main()
{
    int fd, input;
    struct Course db;
    struct flock lock;

    fd = open("record.txt", O_RDWR);

    printf("Enter course number (1-3) to increment student count: ");
    scanf("%d", &input);

    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = (input - 1) * sizeof(db);
    lock.l_len = sizeof(db);
    lock.l_pid = getpid();

    lseek(fd, (input - 1) * sizeof(db), SEEK_SET);
    read(fd, &db, sizeof(db));

    printf("Before entering critical section...\n");

    fcntl(fd, F_SETLKW, &lock);

    printf("Current student count for course %d: %d\n", db.course, db.number_
of_students);

    db.number_of_students++;
```

```
    lseek(fd, (input - 1) * sizeof(db), SEEK_SET);
    write(fd, &db, sizeof(db));

    printf("To confirm booking, press Enter...\n");
    getchar();
    getchar();

    lock.l_type = F_UNLCK;
    fcntl(fd, F_SETLK, &lock);

    printf("Ticket booked for course %d, total students: %d\n", db.course, db.n
umber_of_students);

    close(fd);
    return 0;
}
```

db.c:

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

struct Course
{
    int course;
    int number_of_students;
};

int main()
{
    int fd;
    struct Course db[3];

    for (int i = 0; i < 3; i++)
```

```
    {
        db[i].course = i + 1;
        db[i].number_of_students = 0;
    }

    fd = open("record.txt", O_RDWR | O_CREAT | O_TRUNC, 0744);
    write(fd, db, sizeof(db));
    close(fd);
    printf("Database initialized with 3 courses.\n");
    return 0;
}
```

OUTPUT:

```
./18
Enter course number (1-3) to increment student count: 2
Before entering critical section...
Current student count for course 2: 3
To confirm booking, press Enter...

Ticket booked for course 2, total students: 4
```

CONCEPT:

File vs Record Locking

- In Q17 (ticket reservation), you locked the **whole file**.
- In Q18, we divide the file into **records** (here: 3 course entries).
- Each record can be independently locked → allows **concurrent access** if processes are working on different records.

👉 Example:

- Process A locks Course 1 record.

- Process B can still lock Course 2 record at the same time.
- This improves concurrency while still preventing race conditions.

How record locking works

- Use `fcntl()` with a `struct flock`.
- Specify:
  - `l_whence = SEEK_SET`
  - `l_start = (record_index * sizeof(record))`
  - `l_len = sizeof(record)`
- This means only that byte range is locked.

So → only one record is protected, not the entire file.

Use Cases

- Banking (lock only one account record).
- Ticket booking (lock only one course/seat).
- Inventory management (lock only one product stock entry).

Viva Questions with Answers

Q1. How is record locking different from whole-file locking?

👉 File locking blocks access to the whole file. Record locking locks only a specific range of bytes (a record), allowing concurrent access to other parts of the file.

Q2. How do you specify which record to lock?

👉 By setting `l_start = (record_number - 1) * sizeof(record)` and `l_len = sizeof(record)` in the `struct flock`.

Q3. What happens if two processes try to lock the same record?

👉 If one holds a write lock, the second process blocks (if using `F_SETLKW`) until the lock is released.

Q4. Can multiple processes read the same record simultaneously?

👉 Yes, if they use **read locks (`F_RDLCK`)**, multiple readers are allowed. But if a write lock exists, readers are blocked.

Q5. What are real-life applications of record locking?

👉 Banking systems (per-account locks), reservation systems (per-seat locks), inventory (per-item locks).

Question 19:

Write a program to find out time taken to execute getpid system call. Use time stamp counter.

ANSWER:

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

int main(int argc, char *argv[])
{
   struct timespec start_time, end_time;
   timespec_get(&start_time, TIME_UTC);
   getpid();
   timespec_get(&end_time, TIME_UTC);
   double time_taken = ((end_time.tv_sec - start_time.tv_sec) * 10 ^ 9) + (end_time.tv_nsec - start_time.tv_nsec);
   printf("Time taken by getpid() system call is: %1f nano seconds\n", time_taken);
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que19 % ./19
Time taken by getpid() system call is: 9.000000 nano seconds
```

CONCEPT:

Why measure system call time?

- System calls involve a **context switch** from user mode → kernel mode → back to user mode.
- Measuring execution time shows us **system overhead**.

Time Measurement Options

1. **`time()` / `clock()`** → low resolution (seconds / microseconds).
2. **`gettimeofday()`** → microsecond precision.
3. **`clock_gettime()` / `timespec_get()`** → nanosecond precision.
4. **CPU Timestamp Counter (`rdtsc`)** → hardware cycle-accurate, even finer.

Here you used `timespec_get()` with nanosecond precision.

Viva Questions with Answers

Q1. Why do system calls take more time than normal function calls?

👉 Because they involve a **context switch** from user mode to kernel mode and back.

Q2. Why did we use `timespec_get()` instead of `time()`?

👉 `time()` only gives seconds resolution. `timespec_get()` provides **nanosecond precision**, which is necessary to measure very fast calls like `getpid()`.

Q3. Why is `getpid()` chosen for measurement?

👉 It is a **simple, fast system call** that doesn't do heavy work. Ideal for testing system call overhead.

Q4. How would you measure with CPU Timestamp Counter (`rdtsc`)?

👉 Use inline assembly to read the CPU cycle counter before and after the system call, then subtract. This gives cycle-accurate timings.

Question 20:

Find out the priority of your running program. Modify the priority with nice command.

ANSWER:

```
#i#include <stdio.h>
#include <unistd.h>
#include <sys/resource.h>
#include <sys/time.h>

int main(int argc, char *argv[])
{
   int current_process_id = getpid();
   int priority = getpriority(PRIO_PROCESS, current_process_id);
   printf("Priority of Current process with pid : %d is %d\n", current_process_i
d, priority);
   setpriority(PRIO_PROCESS, current_process_id, priority + 5);
   printf("New Priority of Current process with pid : %d is %d\n", current_proc
ess_id, getpriority(PRIO_PROCESS, current_process_id));
   return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que20 % ./20
Priority of Current process with pid : 7135 is 0
adityadave@Adityas-MacBook-Air-3 Que20 % sudo nice -n 5 ./20
Priority of Current process with pid : 7135 is 5
adityadave@Adityas-MacBook-Air-3 Que20 % ./20
Priority of Current process with pid : 5465 is 0
New Priority of Current process with pid : 5465 is 5
```

CONCEPT:

Process Priority & Scheduling

- In Linux/Unix, the **scheduler** decides which process gets CPU time.
- **Priority (nice value)** influences scheduling.

👉 **Nice Value Range:**

- `-20` → highest priority (more CPU time).
- `0` → default.
- `+19` → lowest priority (less CPU time).

So, lower value = higher priority.

Commands

- `nice -n value ./program` → start a process with a given nice value.
- `renice -n value -p PID` → change priority of an already running process.

System Calls

- `getpriority(int which, id_t who)`
  - `which` can be `PRIO_PROCESS`, `PRIO_PGRP`, `PRIO_USER`.
  - `who` specifies PID, PGID, or UID.

- `setpriority(int which, id_t who, int prio)` → change priority (only root can set negative nice values).

Viva Questions with Answers

Q1. What does the `nice` command do?

👉 It sets the **initial priority (nice value)** of a process when launching it.

Q2. What is the range of nice values in Linux?

👉 From `-20` (highest priority) to `+19` (lowest priority). Default is `0`.

Q3. Who can decrease the nice value (make higher priority)?

👉 Only **root (superuser)**. Normal users can only increase nice value (make process lower priority).

Q4. What is the difference between `nice` and `renice`?

👉 `nice` sets priority at the time of process creation, `renice` changes priority of an already running process.

Q5. What's the difference between `getpriority` and `setpriority` system calls?

👉 `getpriority` retrieves the current nice value of a process, while `setpriority` changes it.

Question 21:

Write a program, call fork and print the parent and child process id.

ANSWER:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
```

```
{
    pid_t q;
    q = fork();
    if (q == 0)
        printf("Child process with PID = %d has Parent process with PID = %d
\n", getpid(), getppid());
    else
        printf("Parent process with PID = %d has Child process with PID = %d
\n", getpid(), q);
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que21 % ./21
Parent process with PID = 7231 has Child process with PID = 7232
Child process with PID = 7232 has Parent process with PID = 7231
```

CONCEPT:

fork() system call

- `fork()` creates a new process by duplicating the calling process.
- The new process is called the **child process**.
- After `fork()`, we have two processes running concurrently.

👉 Return values of `fork()`:

- `0` → in the **child process**.
- Child's PID (>0) → in the **parent process**.
- `1` → error (no new process created).

---

getpid() & getppid()

- `getpid()` → returns the current process's PID.
- `getppid()` → returns parent's PID.

Execution order

- Parent and child run independently after `fork()`.
- Which one runs first depends on the **scheduler**.
- That's why sometimes output order can change (child first or parent first).

Viva Questions with Answers

Q1. What does `fork()` do?

👉 It creates a new process (child) by duplicating the parent. Both processes continue execution after the fork.

Q2. How do parent and child differentiate themselves after `fork()`?

👉 By checking the return value of `fork()`:

- `0` → child process.
- Positive PID → parent process.

Q3. Can the parent process PID and child PID be the same?

👉 No, every process has a unique PID. Parent gets child's PID, and child uses `getppid()` to know its parent.

Q4. What happens if you call `fork()` multiple times?

👉 Each call doubles the number of processes. For example, 2 calls create 4 processes in total.

Q5. What happens if parent does not call `wait()`?

👉 The child process becomes a **zombie** (terminated but not reaped) until the parent reads its exit status.

Question 22:

Write a program, open a file, call fork, and then write to the file by both the child as well as the
parent processes. Check output of the file.

ANSWER:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

int main()
{
    int file_desc;
    file_desc = open("output_file.txt", O_CREAT | O_WRONLY | O_APPEND, 064
4);
    char child_buffer[99] = "This sentence is written by child process.\n";
    char parent_buffer[99] = "This sentence is written by parent process.\n";

    pid_t q;
    q = fork();

    if (q == 0)
    {
        write(file_desc, child_buffer, strlen(child_buffer));
        printf("Successfully Written by child process on file 'output_file.txt'\n");
    }
    else
    {
        write(file_desc, parent_buffer, strlen(parent_buffer));
        printf("Successfully Written by parent process on file 'output_file.txt'\n");
```

```
    }

    close(file_desc);
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que22 % ./22
Successfully Written by parent process on file 'output_file.txt'
Successfully Written by child process on file 'output_file.txt'
```

CONCEPT:

What happens when we `fork()` ?

- `fork()` duplicates the **entire process address space**.
- This includes:
    - Code
    - Data
    - File descriptors

👉 Parent and child **share the same open file description** in the kernel, meaning:

- They point to the same open file.
- They share the same file offset.

Writes after fork

- If both parent and child write to the file descriptor:
    - The writes are handled by the kernel.

    - Since `O_APPEND` was used, both processes append safely at the end of the file.
- If `O_APPEND` wasn't used, they might overwrite each other depending on timing.

Why is output order different sometimes?

- Parent and child execute **concurrently**.
- Depending on scheduling, parent might write first or child might write first.
- That's why the order in file or terminal messages may change across runs.

Viva Questions with Answers

Q1. What happens to file descriptors when `fork()` is called?

👉 They are duplicated, but both parent and child share the same open file description (same offset, same flags).

Q2. Why did we use `O_APPEND` in `open()`?

👉 To ensure that each write goes to the end of the file, preventing overwriting due to concurrent writes.

Q3. What would happen if `O_APPEND` wasn't used?

👉 Parent and child would share the same file offset, so their writes might overlap depending on execution order, corrupting file content.

Q4. Why does the order of "parent wrote" and "child wrote" messages change between runs?

👉 Because parent and child run concurrently, and the CPU scheduler decides which runs first.

Q5. How could we force parent to always write first?

👉 By making the parent call `wait(NULL)` to wait for the child, or by synchronizing using pipes/semaphores.

Question 23:

Write a program to create a Zombie state of the running program.

ANSWER:

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

int main()
{
    pid_t pid = fork();

    if (pid == 0)
    {
        printf("Child process with PID=%d\n", getpid());
        printf("Child process executed.");
    }
    else
    {
        sleep(10);
        printf("Parent process with PID=%d created child with PID=%d\n", getpid
(), pid);
    }

    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que23 % ./23
Child process with PID=8619
Child process executed.Parent process with PID=8618 created child with PID=
```

```
8619

adityadave@Adityas-MacBook-Air-3 Que23 % ps -l -p 8619
  UID   PID  PPID        F CPU PRI NI       SZ    RSS WCHAN     S             ADDR TT
Y           TIME CMD
  501  8619  8618     2006   0   0  0        0      0 -      Z+                 0 ttys003    0:
00.00 <defunct>
```

CONCEPT:

Process states recap

- **Running** → process is executing on CPU.
- **Ready** → waiting for CPU.
- **Waiting/Blocked** → waiting for I/O or event.
- **Terminated (Zombie)** → process has finished execution, but its parent has **not yet collected its exit status**.
- **Orphan** → parent exits while child is still alive; child gets re-parented to `init` (`systemd`).

Zombie Process

- When a child process finishes, it still has an entry in the process table (PID, exit code).
- Parent must call `wait()` or `waitpid()` to read this exit status.
- Until then, the child stays in **zombie (Z)** state.

👉 Zombies don't use CPU or memory, but they occupy **PID slots**. If too many zombies exist, the system may run out of PIDs.

How this program creates a zombie

1. `fork()` → creates child.

2. Child executes and exits immediately.
3. Parent **does not call `wait()`** → so child's exit status is not collected.
4. Parent sleeps for 10 seconds → giving enough time to check with `ps`.
5. During that time, child is shown as `<defunct>` in process table (Zombie).

Viva Questions with Answers

Q1. What is a zombie process?

👉 A terminated child process whose exit status has not yet been collected by its parent. It remains in the process table as `<defunct>`.

Q2. How do you create a zombie process?

👉 By forking a child, letting it exit, and making the parent not call `wait()`.

Q3. Do zombies consume CPU or memory?

👉 No, but they consume a PID slot in the process table.

Q4. How can you remove a zombie process?

👉 The parent must call `wait()` or `waitpid()`. If the parent dies, `init` (PID 1) adopts the zombie and reaps it automatically.

Q5. Difference between Zombie and Orphan process?

👉 **Zombie** → Child finished, parent still alive but didn't `wait()`.

- **Orphan** → Parent finished, child still running.

Question 24:

Write a program to create an orphan process.

ANSWER:

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

int main()
{
    pid_t pid = fork();

    if (pid == 0)
    {
        sleep(10);
        printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());
    }
    else
    {
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
        exit(0);
    }

    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que24 % ./24
Parent process: PID = 8868, Child PID = 8869
Child process: PID = 8869, Parent PID = 1
```

CONCEPT:

Orphan Process

- A child process whose **parent terminates before the child** finishes.

- Such a child becomes an **orphan**.
- Orphans are automatically **adopted by `init` (PID 1)** or by the modern `systemd` in Linux.
- After adoption, `init` periodically calls `wait()` to clean up the orphan's exit status.

Zombie vs Orphan

- **Zombie** → Parent is alive but didn't collect child's exit status.
- **Orphan** → Parent is dead, child is still running.

How this program works

1. `fork()` creates parent + child.
2. Parent executes `exit(0)` → terminates immediately.
3. Child sleeps for 10 seconds (parent already dead).
4. When child wakes up and calls `getppid()`, it sees its parent PID = **1 (`init/systemd`).**

Viva Questions with Answers

Q1. What is an orphan process?

👉 A process whose parent has terminated before it finishes execution. It is adopted by `init/systemd`.

Q2. What happens to the orphan process after parent exits?

👉 It is immediately re-parented to `init` (PID 1), which takes responsibility for reaping it when it finishes.

Q3. Difference between Zombie and Orphan process?

👉

- Zombie: child finished, parent alive but didn't `wait()`.
- Orphan: parent finished, child still alive.

Q4. Why doesn't orphan become zombie?

👉 Because `init` (or `systemd`) always reaps its children, preventing zombies.

Q5. How can you observe orphan adoption in Linux?

👉 Run program, `ps -l -p <child_pid>` after parent dies. Parent PID will show as `1`.

Question 25:

Write a program to create three child processes. The parent should wait for a particular child (use
waitpid system call).

ANSWER:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    pid_t child_pid_1, child_pid_2, child_pid_3;
    child_pid_1 = fork();
    if (child_pid_1 == 0)
    {
        sleep(1);
        printf("1st child process with pid %d execute\n", getpid());
        return 0;
    }
    child_pid_2 = fork();
    if (child_pid_2 == 0)
    {
        sleep(7);
        printf("2nd child process with pid %d executed\n", getpid());
        return 0;
    }
    child_pid_3 = fork();
```

```
    if (child_pid_3 == 0)
    {
        sleep(4);
        printf("3rd child process with pid %d executed\n", getpid());
        return 0;
    }
    int pid3_res = waitpid(child_pid_3, NULL, 0);
    printf("Parent process with pid %d executed after 3rd child process: %d\n",
getpid(), pid3_res);
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que25 % ./25
1st child process with pid 9046 execute
3rd child process with pid 9048 executed
Parent process with pid 9045 executed after 3rd child process: 9048
adityadave@Adityas-MacBook-Air-3 Que25 % 2nd child process with pid 904
7 executed
```

CONCEPT:

Parent-Child process & termination

- Normally, when children terminate, parent must collect their exit status.
- With `wait()` → parent waits for *any* child to finish.
- With `waitpid()` → parent can wait for a *specific child* (by PID).

`waitpid` system call

Prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **pid**
  - `> 0` → wait for that specific child.
  - `1` → wait for any child (same as `wait()`).
- **status** → stores child exit info (can be `NULL`).
- **options**
  - `0` → block until child terminates.
  - `WNOHANG` → return immediately if no child has exited.

Return value:

- PID of child that terminated.
- `0` if WNOHANG and no child exited.
- `1` on error.

Viva Questions with Answers

Q1. Difference between `wait()` and `waitpid()`?

👉 `wait()` waits for *any child*. `waitpid()` can wait for a *specific child* (by PID).

Q2. What happens if you don't use `wait()` or `waitpid()` in the parent?

👉 Child processes become **zombies** after termination, until the parent exits or collects them.

Q3. In this code, why is `sleep()` used in children?

👉 To simulate different execution times so we can clearly see which child finishes first and how the parent waits specifically for child 3.

Q4. Can `waitpid()` return immediately?

👉 Yes, if we pass `WNOHANG` as an option. Then it won't block if the child hasn't finished.

Q5. What's the return value of `waitpid()`?

👉 The PID of the terminated child, `0` if no child finished (with `WNOHANG`), or `-1` on error.

Question 26

Write a program to execute an executable program.
a. use some executable program
b. pass some input to an executable program. (for example execute an executable of $./a.out name)

ANSWER:

```
// 26.c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("Question 26: Trying command execl to execute another program\n");
    execl("./multiply", "multiply", argv[1], argv[2], NULL);
    return 0;
}
```

multiply.c:

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

int main(int argc, char *argv[])
{
    int number1 = atoi(argv[1]);
    int number2 = atoi(argv[2]);
```

```
    int result = number1 * number2;
    printf("%d X %d = %d\n", number1, number2, result);
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que26 % ./26 3 4
Question 26: Trying command execl to execute another program
3 X 4 = 12
```

CONCEPT:

What is `exec`?

- After a `fork()`, the child process may want to replace its code with another program.
- The `exec` **family** of system calls does exactly this: it **replaces the current process image with a new program**.
- After a successful `exec`, the old code is gone — only the new program runs in the same process.

Exec Family Variants

- `execl(path, arg0, arg1, ..., NULL)` → pass args as a list.
- `execv(path, argv[])` → pass args as an array.
- `execlp(file, arg0, arg1, ..., NULL)` → searches PATH.
- `execvp(file, argv[])` → array + PATH search.

👉 Common feature: *On success, they do not return to the calling program.*

How it works here

1. Main program prints `"Trying command execl..."`.
2. Calls:

```
execl("./multiply", "multiply", argv[1], argv[2], NULL);
```

   - `./multiply` → path of new program.
   - `"multiply"` → `argv[0]` of new program.
   - `argv[1], argv[2]` → numbers from command line.
   - `NULL` → end of argument list.
3. `multiply.c` executes → multiplies the two numbers and prints result.

Viva Questions with Answers

Q1. What happens to the original program after a successful `exec`?

👉 It is replaced entirely by the new program — only the new program runs.

Q2. Difference between `fork()` and `exec()`?

👉

- `fork()` → creates a *new process* (child).
- `exec()` → *replaces* the current process image with a new program.

Q3. Why do we pass `"multiply"` as the second argument in `execl`?

👉 That becomes `argv[0]` in the new program, by convention the program name.

Q4. What happens if `execl()` fails?

👉 It returns `-1`, and the old program continues (so you must handle error).

Q5. How does `execlp` differ from `execl`?

👉 `execlp` searches the `PATH` environment variable for the executable, while `execl` requires the full path.

}

Question 27:

Write a program to execute ls -Rl by the following system calls

a. execl

b. execlp

c. execle

d. execv

e. execvp

ANSWER:

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

int main(int argc, char *argv[])
{
    if (strcmp(argv[1], "execl") == 0)
    {
        execl("/bin/ls", "ls", "-Rl", NULL);
    }
    else if (strcmp(argv[1], "execlp") == 0)
    {
        execlp("ls", "ls", "-Rl", NULL);
    }
    else if (strcmp(argv[1], "execle") == 0)
    {
        char *envp[] = {"PATH=/bin", NULL};
        execle("/bin/ls", "ls", "-Rl", NULL, envp);
    }
    else if (strcmp(argv[1], "execv") == 0)
    {
        char *args[] = {"ls", "-Rl", NULL};
        execv("/bin/ls", args);
    }
```

```
    else if (strcmp(argv[1], "execvp") == 0)
    {
        char *args[] = {"ls", "-Rl", NULL};
        execvp("ls", args);
    }

    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que27 % ./27 execl
total 80
-rwxr-xr-x@ 1 adityadave  staff  33760 Sep  1 15:30 27
-rw-r--r--@ 1 adityadave  staff   1204 Sep  3 16:40 27.c
adityadave@Adityas-MacBook-Air-3 Que27 % ./27 execvp
total 80
-rwxr-xr-x@ 1 adityadave  staff  33760 Sep  1 15:30 27
-rw-r--r--@ 1 adityadave  staff   1391 Sep  3 16:45 27.c
```

CONCEPT:

The `exec` family

All `exec*` functions replace the current process image with a new one. The differences are mainly in:

- **How arguments are passed** (list vs array).
- **Whether PATH is searched**.
- **Whether you pass custom environment variables**.

Variants

1. **`execl(path, arg0, arg1, ..., NULL)`**

- Arguments passed as a *list*.
- No PATH search.
- Example:

```
execl("/bin/ls", "ls", "-Rl", NULL);
```

2. `execlp(file, arg0, arg1, ..., NULL)`
   - Like `execl`, but `file` is searched in PATH.
   - Example:

```
execlp("ls", "ls", "-Rl", NULL);
```

3. `execle(path, arg0, arg1, ..., NULL, envp[])`
   - Like `execl`, but lets you provide a custom environment (`envp`).
   - Example:

```
char *envp[] = {"PATH=/bin", NULL};
execle("/bin/ls", "ls", "-Rl", NULL, envp);
```

4. `execv(path, argv[])`
   - Arguments passed as an array (`argv[]`).
   - No PATH search.
   - Example:

```
char *args[] = {"ls", "-Rl", NULL};
execv("/bin/ls", args);
```

5. `execvp(file, argv[])`
   - Like `execv`, but `file` is searched in PATH.
   - Example:

```
char *args[] = {"ls", "-Rl", NULL};
execvp("ls", args);
```

Viva Questions with Answers

Q1. Difference between `execl` and `execv`?

👉 `execl` takes arguments as a list, while `execv` takes arguments as an array.

Q2. Difference between `execl` and `execlp`?

👉 `execl` requires the full path to the executable. `execlp` searches the PATH environment variable for the executable.

Q3. What does `execle` allow that others don't?

👉 It allows the caller to specify a custom environment (`envp`) for the new process.

Q4. What happens if `exec*` succeeds?

👉 The current process image is replaced; control never returns to the old code.

Q5. In your program, what would happen if `/bin/ls` is missing but PATH is set correctly?

👉 `execl` and `execv` would fail (need explicit path), but `execlp` and `execvp` would succeed because they search PATH.

📊 Summary Table

Function	Args format	PATH search?	Custom env?
execl	List	❌ (need full path)	Uses current env
execlp	List	✅	Uses current env
execle	List	❌	✅ (custom envp)
execv	Vector	❌	Uses current env
execvp	Vector	✅	Uses current env

Question 28:

Write a program to get maximum and minimum real time priority.

ANSWER:

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

int main()
{
    int min_prio_fifo = sched_get_priority_min(SCHED_FIFO);
    int max_prio_fifo = sched_get_priority_max(SCHED_FIFO);

    int min_prio_rr = sched_get_priority_min(SCHED_RR);
    int max_prio_rr = sched_get_priority_max(SCHED_RR);

    printf("FIFO POLICY ⇒ Min Priority = %d, Max Priority = %d\n", min_prio_fif
o, max_prio_fifo);
    printf("ROUND ROBIN POLICY ⇒ Min Priority = %d, Max Priority = %d\n",
min_prio_rr, max_prio_rr);

    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que28 % ./28
SCHED_FIFO: Min Priority = 15, Max Priority = 47
SCHED_RR:   Min Priority = 15, Max Priority = 47
```

CONCEPT:

Scheduling Policies in Linux

Linux has two main classes of scheduling:

1. **Normal (time-sharing)** → `SCHED_OTHER` (default policy, used for regular processes).
   - Priorities are handled by **nice values** (−20 to +19).
2. **Real-time** → `SCHED_FIFO` and `SCHED_RR`.
   - Priorities are explicit integers set by the kernel, usually in the range **1 to 99**.
   - Higher number = higher priority.

Real-time policies

- **SCHED_FIFO (First-In, First-Out)**
  - No time slice, runs until it voluntarily yields, blocks, or is preempted by a higher-priority task.
- **SCHED_RR (Round Robin)**
  - Similar to FIFO, but each process at the same priority gets a fixed time quantum in turn.

👉 Both allow *fine-grained control* over scheduling priority using `sched_get_priority_min()` and `sched_get_priority_max()`.

System calls used

- `sched_get_priority_min(int policy)` → returns the minimum priority value allowed for that scheduling policy.
- `sched_get_priority_max(int policy)` → returns the maximum priority value allowed.

Values vary by system, e.g., on many Linux distros it's **1–99**, but your output shows **15–47** (depends on kernel or macOS).

Viva Questions with Answers

Q1. What is the difference between SCHED_FIFO and SCHED_RR?

👉 Both are real-time policies.

- SCHED_FIFO: process runs until it blocks or a higher priority preempts.
- SCHED_RR: same as FIFO, but each process at the same priority gets a fixed quantum in turn.

Q2. How are priorities different in normal vs real-time scheduling?

👉

- Normal (`SCHED_OTHER`) → priorities based on nice values (−20 = highest, +19 = lowest).
- Real-time (FIFO, RR) → kernel-assigned integer priorities (e.g., 1–99).

Q3. What do `sched_get_priority_min()` and `sched_get_priority_max()` return?

👉 They return the lowest and highest valid real-time priorities for a given policy.

Q4. Why do values differ across systems (e.g., 1–99 vs 15–47)?

👉 It depends on the OS and kernel configuration. Some platforms restrict the range to ensure system stability.

Q5. Can a normal user change their process to real-time scheduling?

👉 Usually **no** — it requires root privileges, because real-time processes can starve other tasks if misused.

Question 29:

Write a program to get scheduling policy and modify the scheduling policy (SCHED_FIFO, SCHED_RR).

ANSWER:

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

```
#include <unistd.h>

int main(int argc, char *argv[])
{
    int policy;
    struct sched_param param;

    policy = sched_getscheduler(0);
    if (policy == SCHED_OTHER)
        printf("Current Scheduling Policy: SCHED_OTHER (normal)\n");
    else if (policy == SCHED_FIFO)
        printf("Current Scheduling Policy: SCHED_FIFO (real-time FIFO)\n");
    else if (policy == SCHED_RR)
        printf("Current Scheduling Policy: SCHED_RR (real-time Round Robin)
\n");

    if (strcmp(argv[1], "fifo") == 0)
    {
        param.sched_priority = 10;
        if (sched_setscheduler(0, SCHED_FIFO, &param) == 0)
            printf("Changed policy to SCHED_FIFO with priority %d\n", param.sche
d_priority);
    }
    else if (strcmp(argv[1], "rr") == 0)
    {
        param.sched_priority = 20;
        if (sched_setscheduler(0, SCHED_RR, &param) == 0)
            printf("Changed policy to SCHED_RR with priority %d\n", param.sched
_priority);
    }
    return 0;
}
```

OUTPUT:

```
adityadave@Adityas-MacBook-Air-3 Que29 % gcc 29.c -o 29
adityadave@Adityas-MacBook-Air-3 Que29 % ./29
 ret = -1
The scheduling policy is = 0
 ret = -1
The scheduling policy is = 0

adityadave@Adityas-MacBook-Air-3 Que29 % sudo ./29
 ret = 0
The scheduling policy is = 1
 ret = 0
The scheduling policy is = 2
In mac we don't have SCHED_FIFO and SCHED_RR support.
```

CONCEPT:

Scheduling Policies Recap

- **SCHED_OTHER** → Default Linux time-sharing policy, uses *nice* values (−20 to +19).
- **SCHED_FIFO** → Real-time, first-in-first-out. No time slice.
- **SCHED_RR** → Real-time, round-robin. Fixed time slice among equal priorities.

👉 Only **root** (or processes with `CAP_SYS_NICE` capability) can set **real-time policies** because misuse can starve the system.

System calls

1. **Get policy**

   ```
   int sched_getscheduler(pid_t pid);
   ```

   - Returns current policy for given PID (`0` → current process).

2. **Set policy**

```
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *pa
ram);
```

- Changes policy for given process.
- `struct sched_param` must contain at least `sched_priority`.

Why it failed on macOS?

- macOS and BSD-based systems **do not support SCHED_FIFO and SCHED_RR** the way Linux does.
- They only support `SCHED_OTHER`.
- That's why you saw errors without `sudo`, and even with `sudo`, values didn't match Linux expectations.

Viva Questions with Answers

Q1. What does `sched_getscheduler(0)` return?

👉 The current scheduling policy of the calling process.

Q2. Why do we need `struct sched_param`?

👉 To specify the priority when setting a real-time policy. Without it, `sched_setscheduler` will fail.

Q3. Who can change scheduling policy to real-time?

👉 Only root or processes with the `CAP_SYS_NICE` capability, because real-time processes can starve normal tasks.

Q4. Difference between SCHED_FIFO and SCHED_RR?

👉 Both are real-time. FIFO runs until it blocks or yields. RR gives equal-priority tasks fixed time slices.

Q5. Why did your code not work properly on macOS?

👉 macOS does not support POSIX real-time policies like SCHED_FIFO or SCHED_RR. Only Linux systems with real-time extensions support them.

Question 30:

Write a program to run a script at a specific time using a Daemon process.

ANSWER:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <signal.h>
#include <syslog.h>

#define LOGGING "Start Logging my task = %d\n"

int main()
{

    // I followed seven step to create daemon process.
    pid_t pid;
    int x_fd;

    // STEP 1
    pid = fork();
    if (pid > 0)
        exit(EXIT_SUCCESS);

    // STEP 2
    if (setsid() < 0)
        exit(EXIT_FAILURE);
```

```
    // STEP 3
    signal(SIGCHLD, SIG_IGN);
    signal(SIGHUP, SIG_IGN);

    // STEP 4
    pid = fork();
    if (pid > 0)
    {
        printf("Daemon PID: %d\n", pid);
        exit(EXIT_SUCCESS);
    }

    // STEP 5
    umask(077);

    // STEP 6
    chdir("/");

    // STEP 7
    for (x_fd = sysconf(_SC_OPEN_MAX); x_fd >= 0; x_fd--)
        close(x_fd);

    // STEP 8
    int count = 0;
    openlog("Logs", LOG_PID, LOG_USER);
    while (1)
    {
        sleep(2);
        syslog(LOG_INFO, LOGGING, count++);
    }
    closelog();
    return 0;
}
```

OUTPUT:

```
// ps -ef | grep mydaemon
// log show --predicate 'process == "Logs"' --last 2m
// kill -9 <PID>

/*
Output:
adityadave@Adityas-MacBook-Air-3 Que30 % ./daemon
Daemon PID: 23056
adityadave@Adityas-MacBook-Air-3 Que30 % ps -ef | grep daemon

    0   324     1   0 Fri11AM ??         0:06.64 /usr/sbin/systemstats --daemon
    0   330     1   0 Fri11AM ??         0:00.03 /usr/libexec/IOMFB_bics_daemon
  241   369     1   0 Fri11AM ??         0:14.70 /usr/sbin/distnoted daemon
   88   386     1   0 Fri11AM ??       113:46.26 /System/Library/PrivateFramework
s/SkyLight.framework/Resources/WindowServer -daemon
    0   388     1   0 Fri11AM ??         0:39.40 /usr/sbin/cfprefsd daemon
    0 23053     1   0 10:44AM ??         0:00.03 /System/Library/CoreServices/Re
portCrash daemon
  501 23056     1   0 10:44AM ??         0:00.01 ./daemon
  501 23073 21521   0 10:45AM ttys003    0:00.00 grep daemon
```

CONCEPT:

What is a Daemon Process?

- A **background process** that runs without a controlling terminal.
- Typically started at boot (e.g., system services like `sshd`, `cron`, `syslogd`).
- Runs independently of any user session.
- Often used for **long-running tasks** like logging, monitoring, scheduling jobs.

Steps to Create a Daemon

Your code correctly follows the **classic 7-step procedure**:

1. **Fork once** → parent exits, child continues (so daemon is not a session leader).
2. **Create a new session** with `setsid()` → child becomes session leader, detached from terminal.
3. **Ignore signals** like `SIGHUP` (so it won't terminate if the controlling terminal closes).
4. **Fork again** → ensures daemon is *not a session leader* → prevents it from acquiring a terminal again.
5. **Set umask(0 or restrictive)** → ensures daemon has well-defined file permissions.
6. **Change working directory to** `/` → so it doesn't block unmounting of the current directory.
7. **Close all open file descriptors** → detach from any inherited input/output channels.

After this → daemon runs in background.

Your Example

- After daemonization, it uses `syslog()` for logging.
- Runs an infinite loop, writing a message every 2 seconds.
- Logs can be checked with:

  ```
  log show --predicate 'process == "Logs"' --last 2m
  ```

- Process visible with:

  ```
  ps -ef | grep daemon
  ```

This mimics how real system daemons (like `cron`) work

Understanding your daemon code

Your program follows the **classic 7-step method to create a daemon**:

Step	Code	Purpose
1	`pid = fork(); if(pid > 0) exit(EXIT_SUCCESS);`	Parent exits → child continues. Detaches from terminal.
2	`setsid();`	Create a new session → child becomes session leader, detaches from controlling terminal.
3	`signal(SIGCHLD, SIG_IGN); signal(SIGHUP, SIG_IGN);`	Ignore signals that can kill the daemon.
4	`pid = fork(); if(pid > 0) exit(EXIT_SUCCESS);`	Second fork → ensures daemon **cannot acquire a terminal**.
5	`umask(077);`	Set file permissions for new files created by daemon.
6	`chdir("/");`	Change working directory to root → avoid locking directories.
7	`for(x_fd = sysconf(_SC_OPEN_MAX); x_fd >= 0; x_fd--) close(x_fd);`	Close all inherited file descriptors.
8	Infinite loop with logging: `syslog(LOG_INFO, LOGGING, count++);`	Core daemon work → here, just logs a counter every 2 seconds.