Hands on List-1 Explanation By Jils Patel MTech CSE'27

Q1. Program: Create soft link, hard link, and FIFO in Linux

---

1. Objective

The aim is to create different types of files in Linux using **system calls**:

- **Soft link (symbolic link):** Like a shortcut → points to the original file's path.
- **Hard link:** Another name for the same file → shares the same inode.
- **FIFO (named pipe):** A special file used for **inter-process communication (IPC)**.

We will also see how to create them using **shell commands** vs **system calls**.

---

2. Header Libraries (and why)

#include <stdio.h> // For printf(), fprintf(), fopen(), fclose()

#include <unistd.h> // For symlink(), link()

#include <sys/stat.h> // For mkfifo(), file permission modes

---

3. Equivalent Shell Commands

(i) Soft link

ln -s original.txt soft_link.txt

(ii) Hard link

ln original.txt hard_link.txt

(iii) FIFO

mkfifo myfifo

---

4. Full Code (with comments)

```
#include <stdio.h>
#include <unistd.h>     // symlink(), link()
#include <sys/stat.h>   // mkfifo()
int main() {
  // 1. Create a normal file
  FILE *fp = fopen("original.txt", "w");
  fprintf(fp, "This is the original file.\n");
  fclose(fp);
```

```
    // 2. Create a symbolic (soft) link
    symlink("original.txt", "soft_link.txt");
    printf("Soft link created: soft_link.txt -> original.txt\n");

    // 3. Create a hard link
    link("original.txt", "hard_link.txt");
    printf("Hard link created: hard_link.txt -> original.txt\n");

    // 4. Create a FIFO (named pipe)
    mkfifo("myfifo", 0666);
    printf("FIFO created: myfifo\n");

    return 0;
}
```

---

5. Step-by-step Explanation

1. **Normal file creation**
    - fopen("original.txt", "w"): Creates original.txt.
    - fprintf() writes some content.
    - fclose() closes file.
2. **Soft link**
    - symlink("original.txt", "soft_link.txt"): Creates a symbolic link.
    - soft_link.txt stores the **path** of original.txt.
    - If original.txt is deleted, the soft link becomes **dangling** (broken).
3. **Hard link**
    - link("original.txt", "hard_link.txt"): Creates a hard link.
    - Both files share the **same inode number**.
    - Even if original.txt is deleted, hard_link.txt still contains the data.

4. **FIFO**
    - mkfifo("myfifo", 0666): Creates a named pipe.
    - It's a special file for **process-to-process communication**.
    - 0666: Read & write permission for everyone.

---

6. Sample Output

When program runs:

Soft link created: soft_link.txt -> original.txt

Hard link created: hard_link.txt -> original.txt

FIFO created: myfifo

Then check with:

ls -l

Example:

-rw-r--r-- 1 user user 28 Sep 13 21:30 original.txt

lrwxrwxrwx 1 user user 12 Sep 13 21:30 soft_link.txt -> original.txt

-rw-r--r-- 2 user user 28 Sep 13 21:30 hard_link.txt

prw-rw-rw- 1 user user 0 Sep 13 21:30 myfifo

---

7. Viva Q&A (what examiner may ask)

- **Q: Difference between soft and hard link?**
  A: *Soft link*: Points to filename (can cross partitions). Broken if original file is deleted.

  Hard link: Points to same inode (cannot cross partitions). Survives even if original is deleted.
- **Q: What is FIFO used for?**
  A: FIFO (named pipe) is used for **IPC** so processes can read/write like a normal file but in a synchronized manner.
- **Q: Which system calls are used?**
  A: symlink(), link(), mkfifo().
- **Q: How is FIFO different from an unnamed pipe?**
  A: Unnamed pipe exists only between related processes and disappears when closed. FIFO is a special file visible in filesystem and can be used by unrelated processes.

---

8. Common Mistakes

- Forgetting to include <sys/stat.h> → mkfifo will not compile.
- Not checking return values of system calls → can fail if file already exists.
- Running multiple times → second run may fail because soft_link.txt, hard_link.txt, myfifo already exist. Use unlink() before recreating.

Q2: Infinite Loop & Process Info in /proc

1. Objective

- Write a process that runs continuously in the background.
- Then go to /proc/<PID>/ to explore all information about that process (memory, state, threads, permissions, etc.).
- This helps us understand how **Linux manages processes** and exposes details through the /proc virtual filesystem.

---

2. Header Libraries

#include <stdio.h> // printf()

#include <unistd.h> // getpid(), sleep()

---

3. Code (with comments)

```
#include <stdio.h>
#include <unistd.h>
int main() {
  pid_t pid = getpid();  // get process ID of this program
  printf("Process started. PID: %d\n", pid);

  // infinite loop: keep process alive
  while (1) {
    printf("Process %d is running\n", pid);
    sleep(2);  // sleep 2 seconds to avoid CPU hogging
  }
  return 0;
}
```

---

4. Step-by-Step Explanation

1. **getpid()** → returns current process ID (PID). This PID is used to look up details in /proc/<pid>/.
2. **Infinite loop (while(1))** → keeps the process alive. Without this, process would exit immediately.
3. **printf inside loop** → shows process is running.
4. **sleep(2)** → makes the process wait for 2 seconds between prints (saves CPU).

---

5. Running the Program

gcc 2.c -o myprog

./myprog &

- & runs the program in the background.
- You will see the **PID** printed, e.g., PID: 422.

---

6. Checking Process in /proc

- /proc is a **virtual filesystem** that contains a directory for each running process, named by its PID.
- Example:
- cat /proc/422/status

will show all info about PID **422**.

---

7. Understanding /proc/<pid>/status Output

Some important fields you should mention in viva:

- **Name:** Executable name (a.out in this case).
- **Pid:** Process ID.
- **PPid:** Parent process ID.
- **State:** Current state (R running, S sleeping, Z zombie, etc.).
- **Uid / Gid:** User and group IDs that own the process.
- **VmSize, VmRSS, VmData:** Memory usage (Virtual Memory, Resident Set Size, Data segment).
- **Threads:** Number of threads used.
- **FDSize:** Number of file descriptors allocated.

- **voluntary_ctxt_switches / nonvoluntary_ctxt_switches:** How many times the process switched context voluntarily or due to scheduler.

---

8. Viva Q&A

- **Q: What is /proc?**
  A: /proc is a virtual filesystem in Linux that provides kernel and process information in file form. It doesn't exist on disk; it's generated by the kernel dynamically.
- **Q: What is difference between /proc/<pid>/status and /proc/<pid>/stat?**
  A: status: Human-readable summary.

  stat: Raw numbers used by system tools like ps.
- **Q: What happens if you kill the process?**
  A: Its /proc/<pid>/ directory disappears immediately, since /proc reflects live processes.
- **Q: What is process state S?**
  A: Sleeping — process is waiting (e.g., due to sleep() system call).
- **Q: How to run process in background?**
  A: Append & after command (e.g., ./a.out &).

Q3. Program: Create a File and Print File Descriptor using creat()

1. Objective

Write a program that:

- Creates a new file using the **creat() system call**.
- Prints the **file descriptor value** returned.
- Demonstrates file permissions (0644) and concept of file descriptors.

---

2. Required Header Libraries

- **<stdio.h>** → for printf(), perror().
- **<fcntl.h>** → contains declaration of creat() and file permission flags.
- **<unistd.h>** → for close() system call.

3. Code

```
#include <stdio.h>

#include <fcntl.h>

#include <unistd.h

int main() {

  // Create a file named "myfile.txt" with permission 0644
```

```
  int fd = creat("myfile.txt", 0644);
  // Print file descriptor value
  printf("File created successfully, descriptor value: %d\n", fd);

  // Close the file
 close(fd);
  return 0;
}
```

---

4. Step-by-Step Explanation

1. **creat("myfile.txt", 0644)**
   - Creates a file myfile.txt.
   - If it already exists, it truncates (empties) the file.
   - 0644 = permission mode (rw-r--r--).
2. **Return value**
   - On success → returns a **file descriptor** (non-negative int).
   - On failure → returns -1.
3. **File Descriptor (FD)**
   - An integer index into the kernel's open file table.
   - Standard FDs:
     - 0 → stdin
     - 1 → stdout
     - 2 → stderr
   - First new file → usually gets 3.
4. **close(fd)**
   - Releases the file descriptor.

---

5. Compilation & Execution

gcc 3.c -o filecreate

./filecreate

Sample Output

File created successfully, descriptor value: 3

Check file:

ls -l myfile.txt

Output:

-rw-r--r-- 1 user user 0 Sep 13 22:10 myfile.txt

---

6. Viva Questions & Answers

- **Q: What is creat() used for?**
  A: To create a file (or truncate if it exists). Equivalent to open() with flags O_CREAT | O_WRONLY | O_TRUNC.
- **Q: What is a file descriptor?**
  A: A non-negative integer returned by the kernel that uniquely identifies an open file within a process.
- **Q: Why 0644?**
  A: Permission bits in octal. It means → Owner: read+write, Group: read, Others: read.
- **Q: What are file descriptors 0, 1, 2?**
  A: stdin, stdout, stderr.
- **Q: How is creat() different from open()?**
  A: creat() is a simpler wrapper; internally it just calls open() with fixed flags.

Q4. Write a program to open an existing file with read write mode. Try O_EXCL flag also.

Program Goal

- Demonstrate how to **open/create files** in **read–write mode**.
- Show the effect of using the **O_EXCL flag** with open().

Header Files

#include <stdio.h> // for printf()

#include <fcntl.h> // for open() and file control flags (O_RDWR, O_CREAT, O_EXCL)

#include <unistd.h> // for close()

System Call: open()

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

- **pathname** → name of the file (e.g., "txtFile4.txt").
- **flags** → specify how the file should be opened.
- **mode** → used only if the file is created (defines permissions like 0644).

Returns:

- File Descriptor (FD) ≥ 0 on success.
- -1 on failure.

Flags Used

1. **O_RDWR** → Open file for **both reading and writing**.
2. **O_CREAT** → Create the file if it does not already exist.
    - Needs mode (here 0644 → rw-r--r--).
3. **O_EXCL** → Used with O_CREAT.
    - If the file already exists → **open fails**.
    - If the file does not exist → file is created.

Code Explanation

Part 1:

```
fd = open("txtFile4.txt", O_RDWR | O_CREAT, 0644);
printf("file created/opened. FD: %d\n", fd);
close(fd);
```

- Opens txtFile4.txt in read–write mode.
- If file doesn't exist → creates it with permissions 0644 (owner can read/write, others can read).
- Prints the FD (e.g., 3).

Part 2:

```
fd = open("txtFile.txt", O_CREAT | O_EXCL | O_RDWR, 0644);
printf("O_EXCL succeeded. FD = %d\n", fd);
close(fd);
```

- Tries to create txtFile.txt with O_EXCL.
- If file **already exists** → this call **fails**.

- If file does **not exist** → creates successfully and prints FD.

Output Example

file created/opened. FD: 3

O_EXCL succeeded. FD = 3

- First line: txtFile4.txt created/opened.
- Second line: txtFile.txt created since it didn't exist before.

If you run again without deleting txtFile.txt, the second open() would fail (because of O_EXCL).

Viva Questions

1. **Q:** What is the purpose of O_EXCL?
   A: It ensures the call fails if the file already exists, preventing accidental overwrite.
2. **Q:** What is the difference between creat() and open() with O_CREAT?
   A: creat() is just a simplified version of open() that always uses O_CREAT | O_WRONLY | O_TRUNC.
3. **Q:** What does 0644 mean?
   A: File permissions → Owner: read/write, Group: read, Others: read.

Q5. 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.

1. Creating multiple files

```
int fd[5];
for (int i = 0; i < 5; i++) {
  char filename[20];
  sprintf(filename, "file%d.txt", i + 1);
  fd[i] = open(filename, O_CREAT | O_RDWR, 0644);
  printf("created %s with fd= %d\n", filename, fd[i]);
}
```

- An array fd[5] stores the **file descriptors** of the 5 files.
- Loop runs from 0 to 4.
- Uses sprintf() to generate file names:
  - file1.txt, file2.txt, ..., file5.txt.

- Each file is opened with:
    - O_CREAT → create file if not present.
    - O_RDWR → open in read-write mode.
    - 0644 → file permission rw-r--r--.
- The returned file descriptor (an integer) is printed.

---

2. Printing Process ID

```
printf("process ID: %d\n", getpid());
```

- getpid() returns the **process ID** of the running program.
- Important because we'll need it to check /proc/<pid>/fd.

---

3. Infinite Loop

```
while (1) {
  // keeps the program alive
}
```

- Keeps the process running in background.
- Ensures files remain open so we can inspect /proc/<pid>/fd.

How to Run

1. Compile:
2. gcc 5.c -o q5
3. Run in background:
4. ./q5 &

Example output:

created file1.txt with fd= 3

created file2.txt with fd= 4

created file3.txt with fd= 5

created file4.txt with fd= 6

created file5.txt with fd= 7

process ID: 5123

Checking File Descriptor Table

Go to process directory:

cd /proc/5123/fd

ls -l

Output (example):

0 -> /dev/pts/0 (stdin)

1 -> /dev/pts/0 (stdout)

2 -> /dev/pts/0 (stderr)

3 -> /home/jils/file1.txt

4 -> /home/jils/file2.txt

5 -> /home/jils/file3.txt

6 -> /home/jils/file4.txt

7 -> /home/jils/file5.txt

- FDs **0,1,2** are standard input/output/error.
- FDs **3–7** correspond to file1.txt ... file5.txt.

Viva Questions

1. **Q:** What are file descriptors 0, 1, 2?
   A: 0 = stdin, 1 = stdout, 2 = stderr.
2. **Q:** Why use an infinite loop here?
   A: To keep process alive, so we can check /proc/<pid>/fd.
3. **Q:** What happens if we close a file descriptor?
   A: Its entry disappears from /proc/<pid>/fd.
4. **Q:** How does Linux represent open files in /proc/<pid>/fd?
   A: As **symbolic links** pointing to the actual file.

Q6. Write a program to take input from STDIN and display on STDOUT. Use only read/write system calls

Program Goal

- Take input from **standard input (stdin)** and display it on **standard output (stdout)**.
- Use only **low-level system calls**: read() and write().
- No use of printf() or scanf() (high-level C I/O).

Header Files

#include <unistd.h> // for read() and write() system calls

- read() → reads data from a file descriptor.
- write() → writes data to a file descriptor.

Code (Consolas style)

```
#include <unistd.h>
int main() {
  char c;
  // Infinite loop to read one character at a time from stdin
  while (read(0, &c, 1) > 0) {
    // Write the character to stdout
    write(1, &c, 1);
  }
  return 0;
}
```

Step-by-Step Explanation

1. **read(0, &c, 1)**
    - 0 → file descriptor for **stdin**.
    - &c → address of variable c where the input character will be stored.
    - 1 → read **1 byte at a time**.
    - Returns the number of bytes read (>0) or 0 at EOF.
2. **write(1, &c, 1)**
    - 1 → file descriptor for **stdout**.
    - Writes **1 byte** (the character read) to output.
3. **Loop continues until EOF**
    - Press Ctrl+D (on Linux) to signal **End of File** and terminate input.

How to Run

gcc 6.c -o echo_syscall

./echo_syscall

Example Interaction

Hello Linux

Hello Linux

- Whatever you type appears immediately on the screen.

Viva Questions

1. **Q:** What are 0 and 1 in read(0,...) and write(1,...)?
   A: File descriptors: 0 = stdin, 1 = stdout.
2. **Q:** Difference between read()/write() and scanf()/printf()?
   A:read/write → low-level system calls, unbuffered.

   scanf/printf → high-level C library functions, buffered.
3. **Q:** Why read 1 byte at a time?
   A: To echo each character immediately; can be changed to read multiple bytes at once.
4. **Q:** How to terminate input?
   A: Press Ctrl+D (EOF) in Linux terminal.

Q7. Write a program to copy file1 into file2 ($cp file1 file2).

Program Goal

- Copy the contents of **file1.txt** into **file2.txt** (similar to $ cp file1 file2).
- Demonstrates use of **low-level system calls**: open(), read(), write(), close().

Header Files

#include <fcntl.h> // for open() and file flags

#include <stdio.h> // for printf()

#include <unistd.h> // for read(), write(), close()

Constants

#define BUF_SIZE 1024

- Size of buffer used to read and write data.
- Reads up to 1024 bytes at a time.

Code (Consolas style)

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#define BUF_SIZE 1024
int main() {
  // Open source file in read-only mode
  int fd_src = open("file1.txt", O_RDONLY)
  // Open destination file in write mode, create if not exists, truncate if exists
  int fd_dest = open("file2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  char buffer[BUF_SIZE];
  int bytesRead;
  // Read from source and write to destination
  while ((bytesRead = read(fd_src, buffer, BUF_SIZE)) > 0) {
    write(fd_dest, buffer, bytesRead);
  }
  printf("Copy Successfully\n");
  // Close both files
  close(fd_src);
  close(fd_dest);
  return 0;
}
```

Step-by-Step Explanation

1. Open Files

int fd_src = open("file1.txt", O_RDONLY);

int fd_dest = open("file2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

- fd_src → file descriptor for **source file** (O_RDONLY = read only).
- fd_dest → file descriptor for **destination file**:
    - O_WRONLY → write only
    - O_CREAT → create if not exists

- O_TRUNC → truncate if file exists
- 0644 → file permission (rw-r--r--)

---

2. Read-Write Loop

```
while ((bytesRead = read(fd_src, buffer, BUF_SIZE)) > 0) {
   write(fd_dest, buffer, bytesRead);
}
```

- read() → reads up to BUF_SIZE bytes from fd_src into buffer.
- Returns number of bytes read.
- write() → writes **exact number of bytes read** to destination.
- Loop continues until read() returns 0 (EOF).

---

3. Print Confirmation

```
printf("Copy Successfully\n");
```

- Indicates copying finished.

---

4. Close Files

```
close(fd_src);
close(fd_dest);
```

- Always close file descriptors to **release kernel resources**.

How to Run

```
gcc 7.c -o filecopy
./filecopy
```

- Check files:

```
ls -l file1.txt file2.txt
```

- Contents of file2.txt will match file1.txt.

Viva Questions

1. **Q:** Why use O_TRUNC?
   A: To empty destination file if it already exists before copying.

2. **Q:** What happens if O_CREAT is not used and file doesn't exist?
   A: open() fails; fd_dest will be -1.
3. **Q:** Why use a buffer?
   A: Reading/writing byte by byte is slow; buffer allows **block transfer**.
4. **Q:** Difference between read/write and fread/fwrite?
   A: read/write → low-level, unbuffered system calls.
   fread/fwrite → high-level buffered library functions.
5. **Q:** Why do we pass bytesRead to write() instead of BUF_SIZE?
   A: Last read may contain fewer bytes than buffer size. Writing exact bytes avoids garbage data.

Q8. 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.

Program Goal

- Open a file in **read-only mode**.
- Read its contents **line by line**.
- Display each line on **standard output**.
- Close the file after reaching **end-of-file (EOF)**.

Header Files

#include <fcntl.h> // for open() and file flags

#include <unistd.h> // for read() and write()

- open() → to open a file
- read() → to read data from file descriptor
- write() → to write data to stdout
- close() → to close file descriptor

Code (Consolas Style)

```
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  int fd;
  char c;
  // Check command line arguments
  if (argc != 2) {
    write(1, "this is linux\n", 13);
    return 1;
  }
  // Open file in read-only mode
  fd = open(argv[1], O_RDONLY);
  if (fd == -1) return 1;
  // Read file character by character
  while (read(fd, &c, 1) > 0) {
    write(1, &c, 1);  // Display each character
    if (c == '\n') {
      // End of line reached, next iteration reads next line
    }
  }
  // Close file
  close(fd);
  return 0;
}
```

Step-by-Step Explanation

1. Command-line Argument

```
if (argc != 2) {
  write(1, "this is linux\n", 13);
  return 1;
```

}

- Program expects **file name as argument** (argv[1]).
- If not provided, prints "this is linux" and exits.

---

2. Open File

fd = open(argv[1], O_RDONLY);

if (fd == -1) return 1;

- Opens the specified file in **read-only mode**.
- Returns a **file descriptor** (fd ≥ 0) on success.
- Returns -1 on failure (file not found or permission denied).

---

3. Read Character by Character

```
while (read(fd, &c, 1) > 0) {
  write(1, &c, 1);
}
```

- read(fd, &c, 1) → reads **one character at a time** from file.
- write(1, &c, 1) → writes the character to **stdout**.
- Loop continues until read() returns 0 (EOF).

Note: The if(c == '\n') check is present but not doing anything here. Its purpose could be to **process lines separately** if needed.

---

4. Close File

close(fd);

- Releases the file descriptor and cleans up resources.

How to Run

gcc 8.c -o readline

./readline myfile.txt

Sample Output

(line-by-line content of myfile.txt displayed)

If no file provided:

./readline

Output:

this is linux

Viva Questions

1. **Q:** Why do we use O_RDONLY?
   A: To open the file only for reading; writing is not allowed.
2. **Q:** Why read 1 byte at a time instead of line at a time?
   A: Using read() system call reads raw bytes. To read a full line, you need to check for newline ('\n').
3. **Q:** What does write(1, &c, 1) do?
   A: Writes one character to **stdout** (file descriptor 1).
4. **Q:** How is EOF detected?
   A: read() returns **0** when end-of-file is reached.
5. **Q:** What happens if file cannot be opened?
   A: open() returns -1; program exits.

Q9. 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

Header Files

```
#include <stdio.h>     // printf(), perror()
#include <sys/stat.h>  // struct stat, stat()
#include <stdlib.h>    // exit()
#include <time.h>      // ctime()
#include <unistd.h>    // file system related functions
```

Code (Consolas Style)

```
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
int main() {
  struct stat fileStat;
  const char *filename = "file1.txt";

  // Retrieve file metadata
  if (stat(filename, &fileStat) < 0) {
    perror("stat");
    exit(1);
  }

  // Display file information
  printf("File: %s\n", filename);
  printf("Inode: %ld\n", (long)fileStat.st_ino);
  printf("Number of hard links: %ld\n", (long)fileStat.st_nlink);
  printf("UID: %ld\n", (long)fileStat.st_uid);
  printf("GID: %ld\n", (long)fileStat.st_gid);
  printf("Size: %ld bytes\n", (long)fileStat.st_size);
  printf("Block size: %ld\n", (long)fileStat.st_blksize);
  printf("Number of blocks: %ld\n", (long)fileStat.st_blocks);
  printf("Time of last access: %s", ctime(&fileStat.st_atime));
  printf("Time of last modification: %s", ctime(&fileStat.st_mtime));
  printf("Time of last change: %s", ctime(&fileStat.st_ctime))
  return 0;
}
```

Step-by-Step Explanation

1. stat() system call

if (stat(filename, &fileStat) < 0) { perror("stat"); exit(1); }

- Retrieves **metadata** of the file and stores in struct stat.
- Returns 0 on success, -1 on failure.

---

2. File Metadata in struct stat

Field	Description
st_ino	Inode number (unique file identifier)
st_nlink	Number of hard links
st_uid	User ID of owner
st_gid	Group ID of owner
st_size	File size in bytes
st_blksize	Block size for I/O operations
st_blocks	Number of allocated blocks
st_atime	Last access time
st_mtime	Last modification time
st_ctime	Last status change time

---

3. Display times

ctime(&fileStat.st_atime)

- Converts **epoch time** to human-readable string.
- Similarly for st_mtime and st_ctime.

---

4. Type Casting

(long)fileStat.st_ino

- Casts fields to long to ensure correct printing on 64-bit systems.

Sample Output

File: file1.txt

Inode: 281474976710963

Number of hard links: 1

UID: 1000

GID: 1000

Size: 12 bytes

Block size: 512

Number of blocks: 0

Time of last access: Fri Sep 5 11:10:28 2025

Time of last modification: Fri Sep 5 11:10:28 2025

Time of last change: Fri Sep 5 11:10:28 2025

Viva Questions

1. **Q:** What is an inode?
   A: A unique identifier for a file in the filesystem storing metadata (permissions, ownership, timestamps).
2. **Q:** Difference between st_mtime and st_ctime?
   A:st_mtime → last modification of file content

st_ctime → last change in file **metadata** (ownership, permissions, link count)

3. **Q:** What is st_blksize?
   A: Optimal block size for file I/O operations.
4. **Q:** How many hard links does a newly created file have?
   A: Usually 1.
5. **Q:** Why cast to (long) when printing?
   A: To safely print 64-bit fields like inode on all systems.

Q10. 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.

Program Goal

- Open a file in **read-write mode**.
- Write 10 bytes of data.
- Move the **file pointer** forward by 10 bytes using lseek().
- Write another 10 bytes.
- Observe the **empty spaces (holes)** created in the file.

Header Files

#include <fcntl.h> // open() flags

#include <unistd.h> // write(), lseek(), close()

#include <stdio.h> // printf(), perror()

Code (Consolas Style)

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main() {
  int fd
  // Open or create file in read-write mode, truncate if exists
  fd = open("test10.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);

  // Write first 10 bytes
  write(fd, "ABCDEFGHIJ", 10)
  // Move file pointer 10 bytes ahead from current position
  lseek(fd, 10, SEEK_CUR);
  // Write next 10 bytes
  write(fd, "1234567890", 10);
  close(fd);
  return 0;
}
```

Step-by-Step Explanation

1. Open File

fd = open("test10.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);

- O_RDWR → open for reading and writing
- O_CREAT → create file if it does not exist
- O_TRUNC → truncate file if it exists
- 0644 → file permissions (rw-r--r--)

---

2. Write First 10 Bytes

write(fd, "ABCDEFGHIJ", 10);

- Writes "ABCDEFGHIJ" at **beginning of file** (offset 0–9).

---

3. Move File Pointer

lseek(fd, 10, SEEK_CUR);

- Moves the **file pointer 10 bytes forward** from the current position.
- This creates a **hole** of 10 bytes between the first and second write.
- The return value of lseek() is the **new offset** (useful to verify position).

---

4. Write Next 10 Bytes

write(fd, "1234567890", 10);

- Writes "1234567890" starting from offset 20–29.
- The 10 bytes between offsets 10–19 remain as **empty spaces (filled with zeros)**.

---

5. Close File

close(fd);

- Releases file descriptor.

Checking the File

Use od (octal dump) to visualize file contents:

od -c test10.txt

Output

0000000 A B C D E F G H I J \0 \0 \0 \0 \0 \0

0000020 \0 \0 \0 \0 1 2 3 4 5 6 7 8 9 0

0000036

- Shows **hole (zeros)** between the first and second write.

Viva Questions

1. **Q:** What does lseek() do?
   A: Moves the **file pointer** to a specific location in a file.
2. **Q:** What are the arguments of lseek(fd, offset, whence)?
   A:fd → file descriptor

   offset → number of bytes to move

   whence → reference point (SEEK_SET, SEEK_CUR, SEEK_END)
3. **Q:** What is a **hole** in a file?
   A: Unwritten part of a file (contains zeros) created when file pointer is moved beyond current file end.
4. **Q:** Why use O_TRUNC here?
   A: Ensures file starts empty each time program runs.
5. **Q:** How to verify empty spaces in the file?
   A: Use od -c or hexdump -C.

Q11. 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.(1) use dup (2) use dup2 (3) use fcntl

Program Goal

- Open a file.
- Duplicate its file descriptor in **three different ways**:
  1. dup()
  2. dup2()
  3. fcntl() with F_DUPFD.
- Write to the file using all descriptors.
- Verify that the file is updated properly (all writes go into the same file).

Header Files

#include <fcntl.h> // open(), fcntl()

#include <unistd.h> // dup(), dup2(), write(), close()

#include <stdio.h> // printf(), perror()

Code (Consolas style)

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main() {
  int fd1, fd2, fd3, fd4;
  // Open file in append mode
  fd1 = open("test11.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
  // Duplicate file descriptor using dup()
  fd2 = dup(fd1);
  // Duplicate file descriptor using dup2(), fixed target descriptor (100)
  fd3 = dup2(fd1, 100);
  // Duplicate file descriptor using fcntl()
  fd4 = fcntl(fd1, F_DUPFD, 0);
  // Write to the file using all descriptors
  write(fd1, "Written from fd1\n", 17);
  write(fd2, "Written from fd2 copy from fd1\n", 31);
  write(fd3, "Written from fd3 copy from fd1\n", 31);
  write(fd4, "Written from fd4 copy from fd1\n", 31);
  // Close all descriptors
  close(fd1);
  close(fd2);
  close(fd3);
  close(fd4);

  return 0;
}
```

Step-by-Step Explanation

1. Open the File

fd1 = open("test11.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);

- Opens (or creates) test11.txt in write-only **append mode**.
- 0644 → permissions rw-r--r--.
- First file descriptor returned is fd1.

---

2. Duplicate Using dup()

fd2 = dup(fd1);

- dup() creates a **copy** of fd1.
- The **lowest available file descriptor number** is returned.
- fd1 and fd2 point to the **same open file description** (same file offset, same flags).

---

3. Duplicate Using dup2()

fd3 = dup2(fd1, 100);

- Creates a duplicate of fd1 in a **specific file descriptor number** (here, 100).
- If 100 was already open, it would be closed first.

---

4. Duplicate Using fcntl()

fd4 = fcntl(fd1, F_DUPFD, 0);

- fcntl() with F_DUPFD duplicates fd1 into the **lowest available descriptor greater than or equal to 0**.
- Works similar to dup(), but gives you control of the minimum value of new FD.

---

5. Writing to the File

write(fd1, "Written from fd1\n", 17);

write(fd2, "Written from fd2 copy from fd1\n", 31);

write(fd3, "Written from fd3 copy from fd1\n", 31);

write(fd4, "Written from fd4 copy from fd1\n", 31);

- All descriptors (fd1, fd2, fd3, fd4) point to the **same file**.
- Since the file was opened with **O_APPEND**, every write goes at the end of the file.

---

6. Close File Descriptors

close(fd1);

close(fd2);

close(fd3);

close(fd4);

- Always good practice to free system resources.

Sample File Output (test11.txt)

After running the program, the file will contain:

Written from fd1

Written from fd2 copy from fd1

Written from fd3 copy from fd1

Written from fd4 copy from fd1

Viva Questions

1. **Q:** What is the difference between dup() and dup2()?
   A:dup() → returns lowest available new file descriptor.

   dup2() → lets you specify exactly which file descriptor to duplicate into.

2. **Q:** What is the difference between dup() and fcntl(fd, F_DUPFD, x)?
   A:dup() → always returns the lowest available FD.

   fcntl(F_DUPFD) → lets you specify a **minimum value** for the new FD.

3. **Q:** Do duplicated file descriptors share file offset?
   A: Yes. They share the same **open file description** (offset, flags).

4. **Q:** Why did we use O_APPEND?
   A: To ensure each write always goes to the end of the file, preventing overwriting.

5. **Q:** What happens if you close fd1 but keep writing with fd2?
   A: The file remains open via fd2, because file remains open until **all duplicated descriptors are closed**.

Q12. Write a program to find out the opening mode of a file. Use fcntl.

1. File Opening

```
int fd = open("test12.txt", O_RDWR | O_CREAT, 0644);
```

- Opens (or creates if not present) test12.txt with **read & write** mode.
- Mode 0644 = owner can read/write, group & others can only read.

---

2. Get File Flags

```
int flags = fcntl(fd, F_GETFL);
```

- F_GETFL → gets **file status flags**.
- Flags include: access mode (O_RDONLY, O_WRONLY, O_RDWR) + other options like O_APPEND, O_NONBLOCK.

---

3. Check Access Mode

```
if ((flags & O_ACCMODE) == O_RDONLY)
   printf("READ ONLY\n");
else if ((flags & O_ACCMODE) == O_WRONLY)
   printf("WRITE ONLY\n");
else if ((flags & O_ACCMODE) == O_RDWR)
   printf("READ & WRITE\n");
```

- O_ACCMODE mask extracts only the **access mode bits**.
- Program prints whether the file is **read only / write only / read & write**.
- Since we used O_RDWR, output is:

READ & WRITE

🖥️ **Sample Output**

$ gcc 12.c -o q12

$ ./q12

READ & WRITE

Viva Questions

1. Q: What does fcntl stand for?
   A: File Control – it provides control operations on file descriptors.
2. Q: What is the purpose of F_GETFL in fcntl?
   A: It retrieves the file status flags (access mode + other flags).
3. Q: Why do we use the mask O_ACCMODE?
   A: To extract only the access mode bits (O_RDONLY, O_WRONLY, O_RDWR).
4. Q: What are some common file status flags?
   A: O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_NONBLOCK, etc.
5. Q: What's the difference between file status flags and file descriptor flags?
   A:Status flags → control how I/O works (e.g., read/write, append).

   Descriptor flags → control FD behavior (e.g., FD_CLOEXEC).
6. Q: Difference between open() and fopen()?
   A:open() → system call, low-level, returns int FD.

   fopen() → library function, returns FILE*, internally uses open().
7. Q: If you open a file with O_WRONLY | O_APPEND, what will happen?
   A: You can only write, and every write goes to the end of the file automatically.

Q13. Write a program to wait for STDIN for 10 seconds using select. Print whether data is available within 10 seconds or not.

Code Explanation

```
#include <stdio.h>

#include <unistd.h>

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

- Includes standard I/O, UNIX system calls (read), and the **select()** system call library.

```
int main(){

  fd_set set;

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

- fd_set set; → A special data structure used by select() to monitor file descriptors (FDs).
- struct timeval timeout = {10, 0}; → Sets the timeout to **10 seconds** (10 sec, 0 microseconds).

```
  FD_ZERO(&set);

  FD_SET(0, &set);
```

- FD_ZERO(&set) → Clears the set (removes all file descriptors).
- FD_SET(0, &set) → Adds **file descriptor 0** (which is STDIN) into the set.
  👉 Means we are monitoring **keyboard input**.

```
printf("Enter input within 10 seconds:\n");
if (select(1, &set, NULL, NULL, &timeout) > 0) {
```

- select(nfds, &readfds, &writefds, &exceptfds, &timeout) → Waits until one of the given FDs is **ready for I/O** or until timeout expires.
  - Here:
    - nfds = 1 → Highest FD to check is 0 (stdin), so we pass 1.
    - &set → monitor read events on stdin.
    - NULL, NULL → not checking write or exception conditions.
    - &timeout → wait at most 10 seconds.
- If return value > 0 → Some FD (here stdin) is ready for reading.

```
    char buf[100];
    read(0, buf, sizeof(buf));
    printf("You typed: %s", buf);
```

- If input is available → Read up to 100 characters from stdin into buf.
- Print back the user's input.

```
  } else {
    printf("No input in 10 seconds\n");
  }
  return 0;
}
```

- If timeout expires → Print message "No input in 10 seconds".

✅ **Program Behavior**

- If the user types something within **10 seconds**, it prints the input.
- If no input, it prints **timeout message**.

Viva Questions

1. **Q:** What is the purpose of select() system call?
   A: select() allows monitoring multiple file descriptors simultaneously to see if they are ready for reading, writing, or exceptional conditions.
2. **Q:** What is the role of struct timeval?
   A: It specifies the timeout value (in seconds and microseconds) for how long select() should wait.
4. **Q:** What happens if we pass NULL as timeout in select()?
   A: select() will wait **indefinitely** until at least one FD becomes ready.
5. **Q:** What is the difference between FD_ZERO, FD_SET, and FD_ISSET?
   A:FD_ZERO → Clears all FDs from the set.

   FD_SET(fd, &set) → Adds fd into the set.

   FD_ISSET(fd, &set) → Checks if fd is in the set after select() returns.
6. **Q:** Why is the first argument to select() 1 here?
   A: It should be the highest FD in any set + 1. Since only stdin (0) is monitored, the highest is 0, so 1 is passed.
7. **Q:** What are the possible return values of select()?
   A: > 0 → Number of ready FDs.

   0 → Timeout occurred.

   -1 → Error.
8. **Q:** How is select() different from poll() or epoll()?
   A: select() → Limited by FD_SETSIZE (typically 1024), less scalable.

   poll() → Uses an array of FDs, no fixed size limit.

   epoll() → More efficient for large number of FDs, available in Linux only.
9. **Q:** What happens if input arrives after the 10-second timeout?
   A: The program would already print "No input in 10 seconds" and exit, so late input is ignored.

Code Explanation

#include <stdio.h>

#include <sys/stat.h>

- stdio.h → Standard I/O.
- sys/stat.h → Declares the stat structure and macros to test file types.

---

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

```
struct stat st;
```

- argc, argv → Used to take filename as command-line argument.
- struct stat st; → Holds file metadata (permissions, type, size, etc.).

```
if (argc != 2) {
    printf("Usage: ./task14 <filename>\n");
    return 1;
}
```

- Ensures that exactly **one filename** is passed as argument.
- Example: ./task14 myfile.txt.

```
if (stat(argv[1], &st) == -1) {
    perror("stat");
    return 1;
}
```

- stat(path, &st) → Fills st with file info.
- If it fails (file not found, no permission, etc.), print error with perror.

```
if (S_ISREG(st.st_mode))
    printf("Regular file\n");
else if (S_ISDIR(st.st_mode))
    printf("Directory\n");
else if (S_ISLNK(st.st_mode))
    printf("Symbolic link\n");
else if (S_ISCHR(st.st_mode))
    printf("Character device\n");
else if (S_ISBLK(st.st_mode))
    printf("Block device\n");
else if (S_ISFIFO(st.st_mode))
    printf("FIFO (named pipe)\n");
else if (S_ISSOCK(st.st_mode))
```

```
    printf("Socket\n");
else
    printf("Unknown file type\n");
```

- These macros check st.st_mode bits to determine file type:
    - S_ISREG → Regular file
    - S_ISDIR → Directory
    - S_ISLNK → Symbolic link
    - S_ISCHR → Character device (e.g., /dev/tty)
    - S_ISBLK → Block device (e.g., /dev/sda)
    - S_ISFIFO → FIFO (named pipe)
    - S_ISSOCK → Socket

✅ **Program Behavior**

- If you run: ./task14 myfile.txt → Regular file
- If you run: ./task14 /etc → Directory
- If you run: ./task14 /dev/null → Character device

Viva Questions

1. **Q:** What is the purpose of stat()?
   A: stat() retrieves metadata about a file (size, type, permissions, timestamps, etc.).
2. **Q:** What happens if the file does not exist?
   A: stat() returns -1, and errno is set (e.g., ENOENT for "No such file").
3. **Q:** What does S_ISREG(st.st_mode) check?
   A: Whether the file is a **regular file** (like .txt, .c, etc.).

Q15. Write a program to display the environmental variable of the user (use environ).

Code Explanation

#include <stdio.h>

- Standard I/O functions like printf.

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

- environ is a **global variable** provided by the C runtime.

- It is an array of C strings (char*), each string representing an environment variable in the form:
- KEY=VALUE

Example:

PATH=/usr/bin:/bin

HOME=/home/jils

USER=jils

```
int main() {
  int i = 0;
  while (environ[i] != NULL) {
    printf("%s\n", environ[i]);
    i++;
  }
  return 0;
}
```

- Starts at index 0.
- Loops until it finds NULL (end of array).
- Prints each environment variable string.

✅ **Program Output Example**

PATH=/usr/bin:/bin:/usr/local/bin

HOME=/home/jils

USER=jils

SHELL=/bin/bash

LANG=en_US.UTF-8

...

Viva Questions

1. **Q:** What does environ contain?
   A: It contains the **environment variables** of the process, stored as an array of strings in KEY=VALUE format.

2. **Q:** What is an environment variable?
   A: It is a key-value pair that affects how processes run. Example: PATH defines where executables are searched.
3. **Q:** How do you access environment variables in C besides environ?
   A: Using getenv("VAR_NAME").
4. **Q:** Difference between argv and environ?
   A:
   - argv → Command-line arguments given by the user at program start.
   - environ → Inherited environment variables from the parent process (usually the shell).
5. **Q:** How can you set or modify an environment variable inside C?
   A: Using functions like setenv(), putenv(), or unsetenv().
6. **Q:** Where are environment variables stored in memory?
   A: Typically stored in the process's **user space** (part of the process's memory image set up by the OS when the program starts).
7. **Q:** What happens to environment variables when a process creates a child process?
   A: They are **inherited** by the child (copied).
8. **Q:** Are environment variables global across all processes?
   A: No. They are **per-process**, but inherited from parent to child. Changing them in one process does not affect others.
9. **Q:** Can we remove extern char **environ; and still get environment variables?
   A: Yes, if we declare main as int main(int argc, char *argv[], char *envp[]), then envp provides the same environment strings.

Q16. Write a program to perform mandatory locking. a. Implement write lock b. Implement read lock

16a.c – Write Lock

Purpose

This program demonstrates mandatory file locking in Linux using a write lock. A write lock ensures that no other process can read or write to the file until the lock is released.

Step-by-step Explanation

1. Header files

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

- stdio.h → for printf and scanf

- unistd.h → for write, read, close
- fcntl.h → for fcntl and file locking (struct flock)
- stdlib.h → for exit

---

2. Open the file

```
fd = open("text16a.txt", O_RDWR | O_CREAT, 0666);
```

- Opens text16a.txt in read-write mode.
- Creates the file if it doesn't exist (O_CREAT).
- Permission 0666 → read & write for all users.
- Returns file descriptor fd.

---

3. Set up the lock

```
lock.l_type = F_WRLCK;   // Write lock
lock.l_whence = SEEK_SET; // Start of file
lock.l_start = 0;        // Start offset
lock.l_len = 0;          // Lock entire file
lock.l_pid = getpid();   // Current process ID
```

- F_WRLCK → write lock
- SEEK_SET → lock starts from beginning
- l_len = 0 → lock whole file
- l_pid → needed for identifying the process holding the lock

---

4. Acquire the lock

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

- F_SETLKW → wait until lock is available (blocking call)
- Locks the file for writing.

---

5. Write data

```
scanf("%s", buffer);
write(fd, buffer, sizeof(buffer));
```

- User inputs text and writes it to the file.

- Only one process can write at a time due to the write lock.

---

6. Release the lock

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

- F_UNLCK → unlocks the file.
- F_SETLK → non-blocking unlock.

---

7. Close the file

```
close(fd);
```

✅ Key Points

- Write lock prevents other processes from reading or writing.
- Blocking vs non-blocking locks: F_SETLKW waits; F_SETLK fails if lock unavailable.

2️⃣ 16b.c – Read Lock

Purpose

This program demonstrates mandatory file locking in Linux using a read lock. A read lock allows multiple processes to read, but no process can write while the read lock is active.

Step-by-step Explanation

1. Open file

```
fd = open("text16a.txt", O_RDONLY);
```

- Open the file in read-only mode.

---

2. Set up the lock

```
lock.l_type = F_RDLCK;   // Read lock
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 0;
lock.l_pid = getpid();
```

- F_RDLCK → read lock

- Multiple read locks can coexist.

---

3. Acquire lock

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

- Blocks until write locks are released.
- Other processes can also acquire read locks simultaneously.

---

4. Read file

```
read(fd, buffer, sizeof(buffer));
```

- Reads the file content into buffer.

---

5. Release lock

```
lock.l_type = F_UNLCK;

fcntl(fd, F_SETLK, &lock);
```

---

6. Close file

```
close(fd);
```

✅ Key Points

- Read locks allow concurrent reading but block writing.
- Write locks are exclusive.

Q17. 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.

17a.c – Initialize Ticket Number

Purpose

- This program creates a file db and stores an initial ticket number (10).
- Acts as the database initialization for ticket reservation.

Step-by-Step Explanation

1. Header files

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

2. Structure

```
struct {
  int ticket_no;
} db;
```

- db holds the ticket number.

3. Initialize ticket number

```
db.ticket_no = 10;
```

4. Open or create file

```
fd = open("db", O_CREAT | O_RDWR, 0766);
```

- Opens db for read/write.
- Creates it if not present.
- Permissions 0766 → owner can read/write/execute, others read/write.

5. Write ticket number to file

```
write(fd, &db, sizeof(db));
```

- Stores the structure in the file.

6. Print confirmation

```
printf("Initial ticket number stored: %d\n", db.ticket_no);
```

7. Close file

```
close(fd);
```

✅ Key Point: This program does not use locking, because it just initializes the ticket number.

2️⃣ 17b.c – Ticket Reservation with Write Lock

Purpose

- Simulates online ticket booking using critical section protection.
- Ensures only one process can update ticket number at a time.

Step-by-Step Explanation

1. Open file

```
fd = open("db", O_RDWR);
```

- Open the file for reading and writing.

2. Set up write lock

```
lock.l_type = F_WRLCK;   // Write lock
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 0;        // Entire file
lock.l_pid = getpid();
```

- Prepares a write lock on the whole file.

3. Acquire lock

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

- Blocks if another process has a write lock.
- Only one process can enter critical section.

---

4. Move file pointer to start

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

- Ensures reading starts from the beginning of file.

5. Read current ticket number

```
read(fd, &db, sizeof(db));
printf("Current ticket Number: %d\n", db.ticket_no);
```

6. Increment ticket number

```
db.ticket_no++;
lseek(fd, 0, SEEK_SET);
write(fd, &db, sizeof(db));
printf("New ticket Number: %d\n", db.ticket_no);
```

- Updates the ticket number and writes back to the file.

7. Release lock

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

- Unlocks the file so other processes can enter the critical section.

8. Close file

```
close(fd);
```

✅ Key Point: The write lock ensures mutual exclusion, preventing race conditions if multiple processes try to book tickets simultaneously.

3️⃣ Important Concepts

Concept	Explanation
struct flock	Data structure for file locking.
F_WRLCK	Write lock (exclusive).
F_SETLKW	Blocking lock acquisition.
lseek(fd, 0, SEEK_SET)	Move file pointer to start before reading/writing.
Critical Section (CS)	Code that must be executed by only one process at a time.

4️⃣ **Expected Viva Questions with Answers**

Q1: What is the purpose of 17a.c?
A: It initializes the ticket number in a file db before any reservation.

Q2: What does 17b.c do?
A: It reads the ticket number, increments it (simulating ticket booking), and writes back using a write lock to prevent concurrent access.

Q3: Why do we use F_WRLCK?
A: To prevent other processes from reading or writing while updating the ticket number (mutual exclusion).

Q4: What is the purpose of lseek(fd, 0, SEEK_SET)?
A: To move the file pointer to the beginning before reading or writing.

Q5: Why do we use F_SETLKW instead of F_SETLK?
A: F_SETLKW blocks the process until the lock is available. Ensures no two processes enter the critical section simultaneously.

Q6: What happens if two processes run 17b.c simultaneously?
A: Only one enters the critical section. The second waits until the lock is released. This prevents race conditions.

Q7: What happens if we don't use a lock?
A: Multiple processes could read the same ticket number simultaneously, causing duplicate ticket numbers (race condition).

Q8: What is a critical section in this program?
A: The part where the ticket number is read, incremented, and written back. Only one process should execute it at a time.

Q9: Can multiple processes read the ticket number simultaneously?
A: If we use read lock (F_RDLCK) instead of write lock, yes. But bookijyang requires write lock to avoid duplication.

Q10: What is the role of getpid() in l_pid?
A: Identifies which process owns the lock.

Q18. 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.

18a.c – Create Train Records

Purpose

- Creates a file record.txt with 3 train records.
- Each train has:
    - train_num → train ID (1, 2, 3)
    - ticket_count → number of tickets booked (initially 0).

Explanation

1. Define a structure:

```
struct {
  int train_num;
  int ticket_count;
} db[3];
```

- Array of 3 trains.

2. Initialize trains:

```
for(int i=0;i<3;i++){
  db[i].train_num=i+1;
  db[i].ticket_count=0;
}
```

3. Create/open file record.txt:

```
fd=open("record.txt",O_CREAT|O_RDWR,0666);
```

4. Write all 3 records to file:

```
write(fd,db,sizeof(db));
```

✅ Result: A file with 3 train records is created.

🔹 18b.c – Book Ticket (Write Lock)

Purpose

- Allows user to select a train and book a ticket.
- Uses a write lock so only one process can update a train record at a time.

Explanation

1. Open file:

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

2. Ask user to choose train (1–3):

```
scanf("%d",&input);
```

3. Define lock for selected train:

```
lock.l_type=F_WRLCK; // Write lock
lock.l_whence=SEEK_SET;
lock.l_start=(input-1)*sizeof(db); // start offset
lock.l_len=sizeof(db);          // lock only that train's record
```

👉 This ensures only the chosen train record is locked, not the whole file.

4. Move pointer and read train record:

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

5. Acquire lock:

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

6. Update ticket count:

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

7. Release lock after booking.

✅ Result: Safely increments ticket count for selected train.

🔹 18c.c – View Ticket Count (Read Lock)

Purpose

- Allows user to view current ticket count of a train.
- Uses read lock → multiple people can read simultaneously, but no one can write while reading.

Explanation

1. Open file in read-only mode:

```
fd=open("record.txt",O_RDONLY);
```

2. Ask for train number.
3. Define read lock:

```
lock.l_type=F_RDLCK;   // Read lock
lock.l_start=(input-1)*sizeof(db);
lock.l_len=sizeof(db);
```

4. Acquire lock:

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

5. Read the ticket count and display:

```
read(fd,&db,sizeof(db));
printf("Train %d - Current ticket count: %d\n", db.train_num, db.ticket_count);
```

6. Unlock and exit.

✅ Result: Safely reads ticket count without disturbing booking.

🔹 Summary of Three Codes

- 18a.c → Initializes trains (database setup).
- 18b.c → Simulates ticket booking (with write lock).
- 18c.c → Simulates checking ticket count (with read lock).

Expected Viva Q&A

Q1: What is the purpose of 18a.c?
A: To create record.txt and initialize 3 train records with ticket_count=0.

Q2: Why do we need locks in 18b.c and 18c.c?
A: To avoid race conditions. Without locks, two people booking at the same time might overwrite each other's changes.

Q3: What is the difference between write lock and read lock here?

- Write lock (F_WRLCK) → Only 1 process can book a ticket at a time (exclusive).
- Read lock (F_RDLCK) → Many processes can view ticket count at the same time, but booking (write) is blocked.

Q4: Why do we use lseek in these programs?
A: To move the file pointer to the correct train record before reading/writing.

Q5: What does this mean?

lock.l_start=(input-1)*sizeof(db);

lock.l_len=sizeof(db);

A: It means lock only one train's record, not the whole file.

Q6: What will happen if two people run 18b.c for the same train at the same time?
A: One will enter critical section, the other will wait (because of F_SETLKW).

Q7: What happens if one process runs 18b.c (write lock) and another runs 18c.c (read lock) for the same train?
A: The read process will wait until the write lock is released.

Q8: What happens if two processes run 18c.c at the same time?
A: Both can read simultaneously since read locks are shared.

Q9: Why is sizeof(db) used in write and read?
A: To ensure the entire structure (train record) is read/written, not just part of it.

Q10: What is the role of getpid() in lock.l_pid?
A: To record which process holds the lock (useful for debugging).

Q19. Write a program to find out time taken to execute getpid system call. Use time stamp counter.

Explanation of the Code

```
#include <stdio.h>

#include <unistd.h>

#include <x86intrin.h>

int main(){

  unsigned long long start,end;

  start = _rdtsc();   // Read CPU cycle counter before getpid()
```

```
getpid();        // System call: get process ID
end = _rdtsc();    // Read CPU cycle counter after getpid(
printf("Total Cycles: %llu\n", end - start);
return 0;}
```

Step-by-step:

1. **_rdtsc()**:
    - Reads the **Time Stamp Counter** (TSC), which counts the number of CPU cycles since reset.
    - Returns a 64-bit unsigned integer.
2. **getpid()**:
    - A **system call** that returns the current process ID.
    - Since it's a **kernel interaction**, it takes more cycles than normal instructions.
3. **Cycle Calculation**:
    - end - start gives the number of CPU cycles taken by the getpid() system call.

🔹 **Sample Output**

If you run it, you may see output like:

Total Cycles: 1200

(The number of cycles varies depending on CPU, kernel version, caching, and system load.)

🔹 **Expected Viva Questions (with Answers)**

✅ **Conceptual**

1. **Q: What does getpid() do?**
   A: It returns the process ID of the calling process.
2. **Q: What is _rdtsc()?**
   A: _rdtsc() is an intrinsic instruction (based on the RDTSC assembly instruction) that reads the CPU's Time Stamp Counter, which counts the number of cycles since the CPU was powered on/reset.
3. **Q: Why use TSC instead of time() or clock()?**
   A: TSC gives **high-resolution timing in CPU cycles**, while time()/clock() give coarse-grained measurements in seconds or microseconds.

4. **Q: Is getpid() a system call or a library function?**
   A: It is a **system call**, but usually accessed through a wrapper function in the C standard library (glibc).
5. **Q: Why does a system call take more time than a normal function call?**
   A: Because it involves **switching from user mode to kernel mode** and then back to user mode, which adds overhead.

Q20.

Objective

The program is about:

1. Finding the **priority (nice value)** of the currently running process.
2. Modifying it by **increasing the nice value** (which effectively lowers the priority).

- **Code Explanation**

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

int main() {
  int priority;

  // Get current priority (nice value) of this process
  priority = getpriority(PRIO_PROCESS, 0);
  printf("Old priority: %d\n", priority);

  // Try to increase nice value by 5
  if (setpriority(PRIO_PROCESS, 0, priority + 5) == -1) {
    perror("setpriority");
  }
  else {
    printf("New priority to: %d\n", priority + 5);
```

```
  }


  return 0;
}
```

Step-by-Step Flow

1. **getpriority(PRIO_PROCESS, 0)**
    - PRIO_PROCESS means we are querying based on **process ID**.
    - 0 means **current process**.
    - Returns the current **nice value** of the process.
    - Default nice value = **0**.

👉 Nice value range: **-20 (highest priority)** to **19 (lowest priority)**.

---

2. **setpriority(PRIO_PROCESS, 0, priority + 5)**
    - Tries to set the nice value to priority + 5.
    - If old nice = 0, new nice = 5.
    - **Higher nice → lower scheduling priority.**
    - Only root (superuser) can decrease nice value (make it higher priority), but normal users can increase nice value (make it lower priority).

---

3. **Error Handling**
    - If setpriority fails (e.g., due to permissions), perror("setpriority") prints the error message.
    - Example: If a non-root user tries to reduce the nice value below 0 → Permission denied.

🔹 **Example Run**

$ ./a.out

Old priority: 0

New priority to: 5

If run as a normal user, you can **increase nice**, but cannot decrease it.

Viva Questions

1. **Q: What is a process priority in Linux?**
   A: It determines how often a process gets CPU time. Higher priority = more CPU time.
2. **Q: What is a nice value?**
   A: It's a user-space abstraction for process priority, ranging from -20 (highest) to 19 (lowest). Default is 0.
3. **Q: Difference between priority and nice value?**
   A: Nice value is a user-defined hint; the kernel converts it internally to a scheduling priority.
4. **Q: What does increasing the nice value mean?**
   A: It lowers the priority → process runs less frequently compared to others.
5. **Q: Why can't normal users set negative nice values?**
   A: Because giving high priority (-nice) can hog CPU and affect system stability. Only root can do that.
6. **Q: Alternative way to set nice value from shell?**
   A: Use nice or renice commands. Example:
   - nice -n 10 ./a.out (start with nice value 10)
   - renice 5 -p 1234 (change process 1234 to nice value 5).

Q21. Write a program, call fork and print the parent and child process id.

Code Explanation

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
  pid_t pid;
  pid = fork();   // create a new process
  if(pid < 0){    // fork failed
    printf("error");
    return 1;
  }
  else if(pid == 0){   // Child process (fork() returns 0 to child)
    printf("Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
```

```
    }
    else {    // Parent process (fork() returns child's PID)
        printf("Parent Process: PID = %d, Child PID = %d\n", getpid(), pid);
    }
    return 0;
}
```

Step-by-Step Working

1. **fork()**
    - Creates a new process (child) that is a duplicate of the parent.
    - Returns:
        - >0 → Child's PID (in parent process)
        - 0 → In child process
        - <0 → Error (fork failed)
2. **Parent Process Block (pid > 0)**
    - Executes the else block.
    - Prints its own PID (getpid()) and the PID of the child (pid).
3. **Child Process Block (pid == 0)**
    - Executes the else if block.
    - Prints its own PID (getpid()) and its parent's PID (getppid()).

Viva Questions

1. **Q: What does fork() do?**
   A: Creates a new process (child) which is almost identical to the parent.
2. **Q: How does the return value of fork() differ between parent and child?**
   A: 0 in child, child's PID in parent, -1 on error.
3. **Q: What are getpid() and getppid()?**
   A: getpid() → current process ID, getppid() → parent process ID.
4. **Q: Does child process get a copy of parent's memory?**
   A: Yes, but with **copy-on-write** mechanism (they share pages until modified).
5. **Q: Can the parent terminate before the child? What happens then?**
   A: Yes, then child becomes a **zombie** until parent collects its exit status, or gets re-parented to init/systemd.

6. **Q: How do you prevent zombie processes?**
   A: Use wait() or waitpid() in the parent.

Q22. 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.

Code Explanation

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main() {
  int fd;
  pid_t pid;
  // Open file in write-only + append mode, create if not exists
  fd = open("output22.txt", O_CREAT | O_WRONLY | O_APPEND, 0644);
  pid = fork();  // create child process

  if (pid == 0) {   // Child process
    char child_msg[] = "Message from Child process\n";
    write(fd, child_msg, sizeof(child_msg) - 1);
    printf("Child (PID=%d) wrote to file.\n", getpid());
  }
  else {   // Parent process
    char parent_msg[] = "Message from Parent process\n";
```

```
    write(fd, parent_msg, sizeof(parent_msg) - 1);
    printf("Parent (PID=%d) wrote to file.\n", getpid());
  }

  close(fd);  // both processes close their copy of fd
  return 0;
}
```

- **Important Concepts**

1. **open()**
   - Opens/creates the file output22.txt.
   - Flags:
     - O_CREAT: create file if it doesn't exist.
     - O_WRONLY: write-only mode.
     - O_APPEND: append new data to the end of the file.
   - 0644: file permissions (rw-r--r--).
2. **fork()**
   - Creates a child process.
   - Both parent and child share the **same file descriptor** (but in separate process tables).
3. **write()**
   - Both processes write to the same file.
   - Since file is opened in O_APPEND, each write goes at the **end of file**.
4. **close()**
   - Each process closes its file descriptor after writing.

- **Example Run**

$ ./a.out

Parent (PID=2345) wrote to file.

Child (PID=2346) wrote to file.

output22.txt content might look like:

Message from Parent process

Message from Child process

⚠️ **Order is not guaranteed** — sometimes the child may write first, sometimes the parent, depending on scheduling.

Viva Questions

1. **Q: What happens to file descriptors after a fork()?**
   A: They are **copied** into the child process. Both processes share the same open file description (file offset, flags, etc.).
2. **Q: Why use O_APPEND?**
   A: To ensure both parent and child write at the end of the file safely, without overwriting each other's data.
3. **Q: Can parent and child write simultaneously to the file?**
   A: Yes, but writes of less than PIPE_BUF (usually 4096 bytes) are **atomic** in Linux, so they won't interleave characters. However, the order is nondeterministic.
4. **Q: What if we removed O_APPEND?**
   A: Then both parent and child would share the same file offset → possible race conditions → overlapping writes.
5. **Q: What's the difference between open() and fopen()?**
   A: open() is a low-level system call (returns file descriptor), fopen() is a C library call (returns FILE* and provides buffering).
6. **Q: How can you ensure parent writes after child?**
   A: Use wait(NULL) in parent, so child finishes first.

Q23. Write a program to create a Zombie state of the running program.

Code Walkthrough

```
#include <stdio.h>

#include <sys/wait.h>

#include <unistd.h>


int main() {

  if (!fork()) {

    // Child process (fork returns 0)

    printf("In child - PID = %d\n", getpid());

  }

  else {
```

```
    // Parent process
    printf("This is parent process %d\n", getpid());
    sleep(30);    // Parent sleeps for 30 seconds
    wait(0);     // After sleep, parent reaps the child
  }
}
```

- **Step-by-Step Execution**
  1. **fork()**
     - Creates a child process.
     - Child executes the first block (printf("In child...")).
     - Parent executes the second block.
  2. **Child process**
     - Prints its PID.
     - Then immediately exits → becomes a **zombie** (since parent hasn't wait()ed yet).
  3. **Parent process**
     - Prints its PID.
     - Sleeps for **30 seconds**. During this time:
       - Child is already finished.
       - Entry of child remains in process table as **Zombie (Z)**.
     - After sleep, parent calls wait(0) → collects child's exit status → zombie is removed.

- **How to See Zombie in Action**
  1. Compile & run:
  2. gcc zombie_demo.c -o zombie_demo
  3. ./zombie_demo
  4. While the parent is sleeping (30s window), open another terminal and run:
  5. ps -l

You will see the child with **STAT = Z** (zombie).

Example:

F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD

0 Z 1000 12346 12345 0 80 0 - 0 exit pts/0 00:00:00 zombie_demo <defunct>

6. After 30 seconds, parent calls wait(0) → zombie disappears from the process table.

Key Points for Viva

1. **What's happening here?**
   - Child exits quickly → becomes zombie.
   - Parent is still sleeping → not calling wait() yet.
   - Hence zombie exists for ~30s.
2. **Why does zombie disappear after 30s?**
   - Because parent finally calls wait(0) → kernel removes child's entry.
3. **Why is wait() important?**
   - To reap child processes and prevent zombies.
4. **What if parent never calls wait()?**
   - Zombie stays until parent exits.
   - When parent exits, child gets re-parented to init (or systemd), which reaps it automatically.

Q24. Write a program to create an orphan process.

Code Walkthrough

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

int main() {
  if (!fork()) {  // Child process
    printf("In child - Before orphan PPID = %d\n", getppid());
    sleep(1);   // Wait so parent exits first
    printf("In child - After orphan PPID = %d\n", getppid());
  }
  else {  // Parent process
    printf("This is parent process %d\n", getpid());
```

```
    exit(0);   // Parent exits immediately
  }
}
```

- **Step-by-Step Execution**

1. **Parent creates child** using fork().
    - Child enters first block (!fork() is true → returns 0).
    - Parent enters the else block.
2. **Child (before orphaning)**
    - Prints its parent PID with getppid() → this will be the parent's PID.
3. **Parent exits**
    - Calls exit(0) right away.
    - Parent process is gone → child becomes **orphan**.
4. **Child (after orphaning)**
    - After sleep(1), child wakes up and calls getppid() again.
    - Now parent PID is 1 (on Linux, it's init or systemd), meaning child got **re-parented**.

- **Example Output**

This is parent process 2345

In child - Before orphan PPID = 2345

In child - After orphan PPID = 1

Key Viva Questions

1. **Q: What is an orphan process?**
   A: A child process whose parent has terminated. The orphan is re-parented to init (PID 1), which takes care of it.
2. **Q: What happens to an orphan process?**
   A: It continues execution normally. It's adopted by init/systemd.
3. **Q: Difference between Zombie and Orphan?**
    - **Zombie**: Child finished, parent still alive but hasn't collected its status → entry remains in process table (STAT=Z).
    - **Orphan**: Parent died first, child is still running → re-parented to init (or systemd).

4. **Q: How can you verify an orphan process?**
   A: Run ps -ef or ps -l → check child's PPID. It will be 1 if it's orphaned.

Q25. Write a program to create three child processes. The parentshould wait for a particular child (use waitpid system call).

Included Headers

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

- stdio.h → for printf()
- stdlib.h → for exit()
- sys/wait.h → for waitpid() and macros like WIFEXITED(), WEXITSTATUS()
- unistd.h → for fork(), getpid(), sleep()

Main Function

```
int main() {
  pid_t cpid[3];
  for (int i = 0; i < 3; i++) {
    cpid[i] = fork();
    if (cpid[i] == 0) {
      printf("Child %d (PID: %d) created\n", i + 1, getpid());
      sleep(2);
      exit(0);
    }
  }
```

- pid_t cpid[3]; → stores PIDs of the 3 child processes.

- for loop runs 3 times to create 3 children using fork().
- fork() returns:
    - 0 → if we are in the **child process**
    - 0 → **PID of child** in the parent process
- Inside child (cpid[i] == 0):
    - Prints a message
    - sleep(2) → delay to simulate work
    - exit(0) → terminate child with status 0
- After fork(), the **parent** continues the loop without executing the if block.

Parent Waiting for Child 2

```
printf("Parent PID: %d waiting for Child 2 (PID: %d)\n", getpid(), cpid[1]);


int status;
pid_t wpid = waitpid(cpid[1], &status, 0);


if (WIFEXITED(status)) {
   printf("Child 2 (PID: %d) exited with status %d\n", wpid, WEXITSTATUS(status));
}
```

- getpid() → gets the **parent's PID**
- cpid[1] → PID of the **second child** (arrays are 0-indexed)
- waitpid(cpid[1], &status, 0) → waits **specifically for Child 2** to terminate.
- WIFEXITED(status) → checks if the child exited normally.
- WEXITSTATUS(status) → returns the **exit code** of child.

Return Statement

```
return 0;
```

- Indicates the program ended successfully.

Output

Child 1 (PID: 390) created

Parent PID: 389 waiting for Child 2 (PID: 391)

Child 2 (PID: 391) created

Child 3 (PID: 392) created

Child 2 (PID: 391) exited with status 0

- Shows that **Child 2 finished** and parent waited specifically for it.
- Order of child creation may vary slightly due to process scheduling.

Key Points

1. fork() → creates a new process.
2. waitpid() → allows parent to wait for a **specific child**.
3. WIFEXITED() and WEXITSTATUS() → used to check **exit status**.
4. Each child should call exit() to terminate properly.
5. Parent continues execution for remaining children unless explicitly waited.

Viva Questions

1. **Q:** How many processes are created in this program?
   A: 4 processes in total – 1 parent and 3 children.
2. **Q:** What is waitpid()?
   A: A system call that allows the parent to wait for a **specific child** to terminate.
3. **Q:** Difference between wait() and waitpid()?
   A:
   - wait() → waits for **any child**.
   - waitpid() → waits for a **specific child** by PID.
4. **Q:** Why do we use exit(0) in child processes?
   A: To terminate the child cleanly and indicate successful completion.
5. **Q:** What is WIFEXITED(status) and WEXITSTATUS(status)?
   A:
   - WIFEXITED(status) → checks if child exited normally.
   - WEXITSTATUS(status) → returns the exit code of the child.
6. **Q:** Can parent wait for multiple children with waitpid()?
   A: Yes, by calling waitpid() separately for each child's PID.
7. **Q:** What happens if we use wait(NULL) instead of waitpid()?
   A: Parent waits for **any child** to terminate, not a specific one; output order may vary.

8. **Q:** Why is sleep(2) used in children?
   A: To simulate processing time and make child execution visible; helps observe output order.
9. **Q:** What is the type returned by fork()?
   A: pid_t (integer type representing process IDs).
10. **Q:** Can fork() fail? How do you handle it?
    A: Yes, if the system is out of resources. Handle using:

Q26. 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).

Code Explanation

```
#include <stdio.h>

#include <unistd.h>
```

- stdio.h → for standard input/output functions like printf() (though not used here).
- unistd.h → for **execl()** and other POSIX system calls.

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

- argc → counts the **number of command-line arguments**.
- argv → array of strings containing **command-line arguments**.
  - argv[0] → name of the program (26 if run as ./26).
  - argv[1], argv[2] → first and second arguments passed to program.

---

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

- **execl()** is a system call that **replaces the current process** with a new program.
- Arguments:
  1. "./function" → path to executable to run.
  2. "function" → argv[0] for the new program (can be the program name).
  3. argv[1], argv[2] → arguments passed from current program.
  4. NULL → marks the end of argument list (mandatory).

Important points:

- After execl(), the current program **does not continue** if successful.
- If execl() fails (e.g., file not found), it **returns -1**.

---

return 0;

- This runs **only if execl() fails**, since a successful execl() does **not return**.

Example Execution

gcc 26.c -o 26

./26 hello world

- The program will attempt to run ./function hello world.
- If ./function exists, it **replaces the current process**, so nothing else in 26.c runs.
- If it fails, program returns 0 (or better to handle failure with perror()).

Key Viva Questions

1. **Q:** What is execl()?
   A: It replaces the current process image with a new program, passing arguments as a list.
2. **Q:** How is execl() different from fork()?
   A:
   - fork() → creates a **new child process** that runs concurrently.
   - execl() → **replaces the current process**, does not create a child.
3. **Q:** What happens after a successful execl()?
   A: The current process is replaced; **no code after execl() executes**.
4. **Q:** Why is NULL required at the end of execl()?
   A: To indicate the **end of arguments**; otherwise, execl() doesn't know where the list ends.
5. **Q:** What are argv[1] and argv[2]?
   A: Command-line arguments passed to the program; in this case, forwarded to ./function.
6. **Q:** What happens if ./function does not exist?
   A: execl() fails and returns -1. Program continues (or should print an error with perror()).
7. **Q:** How is execl() different from execv()?
   A:
   - execl() → arguments passed as **list**.
   - execv() → arguments passed as **array of strings** (char* argv[]).

8. **Q:** Does the program create a new process?
   A: No, it **replaces the current process**. If you need a new process, you first call fork() then execl() in the child.
9. **Q:** Why might you include perror("execl failed")?
   A: To **debug** and see why execl() failed (e.g., file not found or permission denied).

Q27. Write a program to execute ls -Rl by the following system calls (a) execl (b) execlp (c) execle (d) execv (e) execvp

Code Explanation:

```
#include <stdio.h>

#include <unistd.h>
```

- unistd.h → contains all the **exec family system calls** (execl, execv, etc.)
- stdio.h → standard I/O (not used here but commonly included)

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

{
```

- Standard main function with command-line arguments, though **not used** in this program.

---

1. execl()

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

- **Replaces the current process** with /bin/ls.
- Arguments passed as a **list**:
  1. /bin/ls → path to executable
  2. ls → argv[0] (program name)
  3. -Rl → argument to ls
  4. NULL → end of list
- If successful, **code after this line is never executed**.

---

2. execlp()

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

- Similar to execl(), but **searches PATH** environment variable to locate ls.
- You don't need to give the full path.

---

3. execle()

```
char *envp[] = {"PATH=/bin", NULL};

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

- execle() → similar to execl(), but allows you to **specify a custom environment** (envp).
- envp[] defines **environment variables** for the new process.

---

4. execv()

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

execv("/bin/ls", args);
```

- execv() → arguments passed as **array** instead of a list.
- First element is argv[0] (program name), last element must be NULL.

---

5. execvp()

```
execvp("ls", args);
```

- execvp() → like execv(), but **searches PATH** to locate the program.
- Handy when you don't know the full path.

Important Notes

1. **Only the first exec that succeeds runs.**
    - After a successful exec*(), the current process is replaced; no code after it executes.
2. **If exec fails**, control moves to the next line.
    - Usually you should add perror("exec failed") to catch errors.
3. return 0; is executed only if **all exec calls fail**.

Viva Questions

1. **Q:** What is the exec family of functions?
   A: System calls that **replace the current process** with a new program.
2. **Q:** What's the difference between execl and execv?
   A:
   - execl → arguments as a **list**
   - execv → arguments as an **array**
3. **Q:** Difference between execl and execlp?
   A: execlp searches the **PATH** environment to locate the executable; execl needs the **full path**.
4. **Q:** Difference between execle and execl?
   A: execle allows you to **specify a custom environment** for the new process.
5. **Q:** Difference between execv and execvp?
   A: execvp searches PATH to find the executable; execv requires **full path**.
6. **Q:** What happens if an exec call succeeds?
   A: The **current process is replaced**, code after the exec **does not execute**.
7. **Q:** What happens if an exec call fails?
   A: It returns -1, and you can use perror() to print the error.
8. **Q:** Can you run multiple exec calls sequentially like this?
   A: Only if **previous exec calls fail**, because a successful exec replaces the process.
9. **Q:** Why is NULL required at the end of arguments or env array?
   A: It **terminates the argument or environment list**, required by the kernel.

Q28. Write a program to get maximum and minimum real time priority.

Code Explanation: sched_get_priority_min/max

```
#include <stdio.h>

#include <sched.h>
```

- sched.h → contains functions and macros for **real-time scheduling** in Linux (SCHED_FIFO, SCHED_RR, etc.)

```
int main()

{

  int mpf = sched_get_priority_min(SCHED_FIFO);

  int mxpf = sched_get_priority_max(SCHED_FIFO);


  int mpr = sched_get_priority_min(SCHED_RR);
```

```
int mxpr = sched_get_priority_max(SCHED_RR);
```

- sched_get_priority_min(policy) → returns **minimum priority** allowed for the given scheduling policy.
- sched_get_priority_max(policy) → returns **maximum priority** allowed.
- SCHED_FIFO → **First-In-First-Out** real-time scheduling policy.
- SCHED_RR → **Round-Robin** real-time scheduling policy.
- Example output (depends on system):
- 1 99
- 1 99
    - Real-time priorities in Linux usually range **1–99**.

```
  printf("%d %d\n", mpf, mxpf);
  printf("%d %d\n", mpr, mxpr);


  return 0;
}
```

- Prints **minimum and maximum priorities** for both SCHED_FIFO and SCHED_RR.

Important Notes

1. These functions **do not set the priority**; they only query the system limits.
2. Only **root users** can set real-time scheduling policies.

Viva Questions

1. **Q:** What is sched_get_priority_min()?
   A: Returns the **minimum priority** allowed for a specific scheduling policy.
2. **Q:** What is sched_get_priority_max()?
   A: Returns the **maximum priority** allowed for a specific scheduling policy.
3. **Q:** What are SCHED_FIFO and SCHED_RR?
   A:
    - SCHED_FIFO → First-In-First-Out real-time scheduling
    - SCHED_RR → Round-Robin real-time scheduling

4. **Q:** Can normal users change the priority of a process?
   A: No, **only root** can assign real-time priorities.
5. **Q:** What is the typical priority range in Linux for real-time policies?
   A: 1 (minimum) to 99 (maximum)
6. **Q:** What happens if you pass an invalid policy to sched_get_priority_min/max?
   A: It returns -1 and sets errno.
7. **Q:** Difference between SCHED_OTHER, SCHED_FIFO, and SCHED_RR?
   A:
   - SCHED_OTHER → normal time-sharing scheduling
   - SCHED_FIFO → first-in-first-out real-time
   - SCHED_RR → round-robin real-time

Q29. Write a program to get scheduling policy and modify the scheduling policy (SCHED_FIFO, SCHED_RR).

Code Explanation

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

- sched.h → provides **scheduling functions and structures** (sched_getscheduler, sched_setscheduler, sched_param, etc.)
- unistd.h → system calls
- stdio.h/stdlib.h → standard I/O and utilities

Function to Print Policy

```
void print_policy(int policy) {
  if (policy == SCHED_OTHER)
    printf("Policy: SCHED_OTHER (normal)\n");
  else if (policy == SCHED_FIFO)
    printf("Policy: SCHED_FIFO\n");
  else if (policy == SCHED_RR)
    printf("Policy: SCHED_RR\n");
```

```
  else
    printf("Policy: UNKNOWN (%d)\n", policy);
}
```

- Converts a **numeric policy** into a **readable string**.
- Useful for displaying results.

Main Function

```
int main() {
  int pid = 0;
  int old_policy, new_policy;
  struct sched_param param;
```

- pid = 0 → refers to **current process** in sched_getscheduler and sched_setscheduler.
- struct sched_param param → holds **priority** for real-time scheduling.

Get Current Scheduling Policy

```
old_policy = sched_getscheduler(pid);
printf("Old ");
print_policy(old_policy);
```

- sched_getscheduler(pid) → returns **current scheduling policy** for the process.
- Prints old/current policy using print_policy().

Set New Scheduling Policy

```
param.sched_priority = 10;
sched_setscheduler(pid, SCHED_FIFO, &param);
```

- Sets **new scheduling policy** to SCHED_FIFO (real-time FIFO).
- sched_priority = 10 → priority for the real-time process (must be between min/max for SCHED_FIFO).
- **Root privileges required** to set real-time scheduling.

Check New Policy

```
new_policy = sched_getscheduler(pid);
```

```
printf("New ");
print_policy(new_policy);
```

- Verifies that the scheduling policy has changed successfully.

Return Statement

```
return 0;
```

- Program ends successfully.

Important Notes

1. Only **root** can set real-time scheduling policies (SCHED_FIFO, SCHED_RR).
2. SCHED_OTHER → default Linux time-sharing policy for normal processes.
3. sched_getscheduler() returns -1 on failure (check errno).
4. sched_setscheduler() can fail if priority is **out of range** or not root.

Viva Questions

1. **Q:** What is sched_getscheduler()?
   A: Returns the **current scheduling policy** of a process.
2. **Q:** What is sched_setscheduler()?
   A: Sets a **new scheduling policy** and priority for a process.
3. **Q:** What is struct sched_param?
   A: Structure containing scheduling parameters like sched_priority.
4. **Q:** What are the scheduling policies in Linux?
   A:
   - SCHED_OTHER → normal time-sharing
   - SCHED_FIFO → first-in-first-out real-time
   - SCHED_RR → round-robin real-time
5. **Q:** What happens if a normal user tries to set SCHED_FIFO?
   A: sched_setscheduler() fails with **EPERM** (permission denied).
6. **Q:** What is the valid priority range for SCHED_FIFO?
   A: sched_get_priority_min(SCHED_FIFO) to sched_get_priority_max(SCHED_FIFO) (usually 1–99).
7. **Q:** What is the difference between SCHED_FIFO and SCHED_RR?
   A:

- SCHED_FIFO → executes in **first-come, first-served** order
- SCHED_RR → executes in **round-robin slices**

8. **Q:** What is the significance of pid = 0?
   A: Refers to the **calling process itself**.

Q30.Write a program to run a script at a specific time using a Daemon process.

Code Explanation: Creating a Daemon Process

```
#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>
```

- Includes necessary headers for process management, signals, logging, and file system operations.

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

- A macro for the log message format.

Step 1: First fork

```
pid = fork();
if (pid > 0)
  exit(EXIT_SUCCESS);
```

- Creates a **child process**.
- Parent exits (exit(EXIT_SUCCESS)), allowing the child to **run independently**.

Step 2: Create new session

```
if (setsid() < 0)
  exit(EXIT_FAILURE);
```

- setsid() creates a **new session**:
    - The child becomes **session leader**
    - Detached from controlling terminal
    - Becomes **leader of a new process group**

Step 3: Ignore signals

```
signal(SIGCHLD, SIG_IGN);
signal(SIGHUP, SIG_IGN);
```

- SIGCHLD → ignore termination of child processes.
- SIGHUP → ignore hangup signal (terminal closes).

Step 4: Second fork

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

- Second fork ensures **daemon is not session leader** → cannot accidentally acquire a controlling terminal.

Step 5: Set file permissions and working directory

```
umask(077);
chdir("/");
```

- umask(077) → new files are **accessible only by owner**.
- chdir("/") → sets working directory to root, avoiding locking a mounted filesystem.

Step 6: Close open file descriptors

```
for (x_fd = sysconf(_SC_OPEN_MAX); x_fd >= 0; x_fd--)
  close(x_fd);
```

- Closes all inherited file descriptors (stdin, stdout, stderr, etc.)

Step 7: Logging loop

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

- openlog() → initialize connection to syslog with identifier "Logs".
- Infinite loop logs a **counter every 2 seconds** using syslog(LOG_INFO, ...).
- closelog() → closes connection to syslog (though never reached in infinite loop).

Key Concepts

1. **Daemon process** → background process detached from terminal.
2. **Double fork technique** → ensures daemon cannot acquire controlling terminal.
3. **syslog** → used to log messages to system log.
4. **umask, chdir, close file descriptors** → best practices for daemons.

Viva Questions

1. **Q:** What is a daemon process?
   A: A background process that runs independently of the terminal.
2. **Q:** Why do we use double fork to create a daemon?
   A: First fork → child runs in background. Second fork → ensures **daemon cannot acquire controlling terminal**.
3. **Q:** What does setsid() do?
   A: Creates a **new session**, makes the process **session leader** and detaches it from terminal.
4. **Q:** Why do we ignore SIGHUP and SIGCHLD?
   A:
   - SIGHUP → prevents daemon termination when terminal closes
   - SIGCHLD → prevents zombie processes when child terminates

5. **Q:** Why set umask(077)?
   A: Ensures newly created files are **accessible only by owner**, for security.
6. **Q:** Why change working directory to /?
   A: Avoids locking the current directory, especially if it's on a **mounted filesystem**.
7. **Q:** Why close all file descriptors?
   A: Daemon should **not use stdin, stdout, stderr**; avoids interfering with terminal or files.
8. **Q:** What is syslog used for?
   A: To log messages to the **system log** instead of console.
9. **Q:** How often does this daemon log messages?
   A: Every 2 seconds, increments a counter.
10. **Q:** What privileges are required to create a daemon?
    A: Normal user can create a daemon; **root privileges** may be needed for certain syslog operations or file permissions.