Commands and Function
Learning

---P_1-------------------------

Commands:

echo :
    -Print a message to the terminal

-Display the value of variables ex -
        $name="Shikhar"
        $echo "My
name is $name"
    -Formatting output (with options)
        -e ? Enable interpretation
of backslash escapes
        $echo -e "Line1\nLine2\tTabbed"
    -Redirect output to
a file
        $echo "Hello — original file (symlink demo)" > original.txt

ln : [Importent]
1.Hard Link (default)
$ln original.txt hardlink.txt
    -Creates another
name (hardlink.txt) pointing to the same inode as original.txt.
    -Both share the same data
blocks.
    -If one is modified, the other reflects changes.
    -If the original is deleted,
the data still exists (through the hard link).
2.Symbolic Link (Soft Link)
$ln -s original.txt
symlink.txt
    -Creates a shortcut (path reference) to the original file.
    -If the original
file is deleted, the symlink becomes broken.
    -Can link to directories and across
filesystems.
-s ? Create symbolic link (soft link).

ls (List File) : used to list file

-Simple list :$ls
    -Long listing :$ls -l
    -Hidden files included : $ls -a

-Human-readable sizes : $ls -lh
    -Recursive listing: $ls -R
    -List with inode number :
$ls -li

cat (concatenate) : used to view, create, append, merge, and copy files
    -Display
contents of a file : $cat file.txt
    -View multiple files together : $cat file1.txt
file2.txt
    -Create a new file : $cat > newfile.txt
    -Append content to a file : $cat
>> existing.txt
    -Combine multiple files into one : $cat file1.txt file2.txt >
merged.txt
    -Number the lines in output : $cat -n file.txt
    -Quickly copy a file : $cat
file.txt > copy.txt

chmod (change mode) :
    - $chmod 755 script.sh
    -> Owner (7),
Group (5), Others (5)

mkfifo (Make Named Pipe) :
    -creates a named pipe FIFO = First In,
First Out
    -mkfifo creates a named pipe FIFO = First In, First Out.
    -It is a special
file that processes can use for inter-process communication IPC.
    -Unlike normal pipes (|),
a named pipe exists as a file in the filesystem.
    =>$mkfifo mypipe
    =>$ls -l

prw-r--r-- 1 shikhar users 0 Sep 2 23:45 mypipe
    p at the start (prw-…) shows it's a
pipe.

Permission Codes:
Read = 4
Write = 2
Execute = 1
rwx = 4+2+1 = 7
rw- = 4+2+0 = 6
r-- =
4+0+0 = 4

-------------------- C Programming : --------------------

open(orig,
O_WRONLY | O_CREAT | O_TRUNC, 0644); --> Opens or creates file descriptor for
writing.
close(); --> close file descriptor

write(); --> Writes it into file.
read();
--> Read data into var from file.

symlink(orig, slink); --> Creates a symbolic link
named SoftLink named hp1as.txt pointing to file.
link(orig, hlink); --> Creates a hard
link
unlink(orig); --> Deletes the directory entry for
original_hard.txt.
mkfifo("myfifo", 0666); --> tries to create a named pipe with
read/write permissions for all.
fork(); --> duplicates the process.

perror(); --> used
to print a human-readable error message to stderr standard error.

lstat(slink, &st_link);
--> gets info about the symlink itself.
stat(orig, &st1); --> function fills st1 with
metadata about the file named by orig.

readlink(); --> reads the contents of a symlink the
path it points
to.

--P_2------------------------------
----

while(1) --> infinite
loop

--P_3------------------------------

-------------------- C Programming :
----

creat("myfile.txt", 0644); --> Calls creat to create or
truncate the file 'myfile.txt' and open it for writing. The mode 0644 is an octal

literal.

---P_4--------------------------

--------------------   C Programming :

O_EXCL(Exclusive creation) ensures that the open fails if the file already exists.

open("myfile.txt", O_RDWR | O_CREAT | O_EXCL, 0644); -->
O_EXCL only meaningful with O_CREAT ? fail if the file already exists.

---P_5---------------------------------

Commands:

after compilation run this
$./create_files
& --> runs the compiled program create_files as a background process, letting you continue using the terminal while it runs.
Still tied to your terminal session.
o/p --> [1] 40877   === job number (like [1]) and the PID (process ID)

$nohup ./create_files & --> nohup = no hangup.

Tells the system to ignore SIGHUP signals, so the program keeps running even after logout or terminal close.

--------------------   C Programming :
----
snprintf(filename, sizeof(filename), "file%d.txt", i + 1);
--> snprintf ensures the string fits in filename safe version of sprintf.

(Used to assign filename in char array.)

---P_6--------------------------

--------------------   C Programming :
----

1. read(int fd, void *buf, size_t count):
=>Purpose:

->read() is a low-level system call that reads raw bytes from a file descriptor into memory.
->Works for files, pipes, sockets, devices — basically anything that can be represented by a file descriptor (fd).
=>Parameters
->fd
? File descriptor
-Obtained from open(), pipe(), socket(), etc.

- 0 = standard input (stdin)
- 1 = standard output (stdout)

- 2 = standard error (stderr)
->buf ? Pointer to a buffer (array in memory) where data will be stored.
->count ? Maximum number of bytes to read.
=> Return Value
-> > 0 ? Number of bytes actually read.
-> 0 ? End of file (EOF).
-> -1 ? Error (sets errno).

// Calls the low-level read() system call:
// 0 is the file descriptor for STDIN.
//
buffer is where bytes will be stored.
// sizeof(buffer) = 100 is the maximum number of
bytes to read.
// read() blocks until input is available, EOF is reached, or an error
occurs.
// Return value:
-->n > 0 ? number of bytes actually read (?
100).
-->n == 0 ? EOF (no more input).
-->n == -1 ? error occurred
(and errno is set).

2. write(int fd, const void *buf, size_t count):
=>Purpose:

->write() is a low-level system call that writes raw bytes from memory (buf) to a file
descriptor (fd).
->Works for files, pipes, sockets, devices — basically anything
that can be represented by a file descriptor.

=>Parameters:
1. fd ? File
descriptor
Obtained from open(), pipe(), socket(), etc.
0 = standard
input (stdin)
1 = standard output (stdout)
2 = standard error (stderr)

2. buf ? Pointer to a buffer (array in memory) where data will be stored.
3.
count ? Maximum number of bytes to read.
4. Return Value
> 0 ?
Number of bytes actually read.
0 ? End of file (EOF).
-1 ? Error
(sets
errno).

---P_7--------------------------

Commands:
copy <filename> <destination>

-------------------- C Programming : --------------------

1.main(int argc, char
*argv[]) :
=> Parameters:
1.argc ? Argument Count
An integer
representing the number of command-line arguments passed to the program.
Always ? 1
because the program name itself counts as the first argument.
2.argv ? Argument
Vector
An array of strings (char pointers), holding the arguments.
argv[0]
? name of the program (string).
argv[1] ... argv[argc-1] ? actual arguments
passed by the user.
argv[argc] ? guaranteed to be NULL (marks the end).
ex -
./mycp <source> <destination>

```
2.while ((n = read(fd1, buffer, SIZE)) > 0)
{
    if (write(fd2, buffer, n) != n) {}
```

}

--P_8--

-------------------- C Programming : --------------------

```
    Read file line by line
    while ((n = read(fd, buffer, SIZE)) >
0) { // Read Line by Line
        for (int i = 0; i < n; i++) {
            line[idx++] =
buffer[i];
            if (buffer[i] == '\n') {       // found a line

write(1, line, idx);       // print to STDOUT
                idx = 0;                   //
reset line buffer
            }
        }

}
```

--P_9--

-------------------- C Programming : --------------------

fprintf(stderr, "Usage: %s <filename>\n", argv[0]) :

->It is just like printf(), but instead of printing to stdout (screen), it prints (writes formatted text) to a file stream.

->You control where the output goes (file, terminal, socket, etc.).

struct stat fileStat

->used to define struct of file.

stat(filename, &fileStat):

->fills fileStat with metadata about the file argv[1].

argv[1] : Prints the name of the file being examined.
(long)fileStat.st_ino : Prints inode number (unique identifier for the file on disk).
(long)fileStat.st_nlink : Number of hard links ? how many directory entries (names) point to this file.
fileStat.st_uid : UID (User ID) of file’s owner.
fileStat.st_gid : GID (Group ID) of file’s group.

(long)fileStat.st_size : File size in bytes (actual data length).

(long)fileStat.st_blksize : Preferred block size for filesystem I/O (not file size). Helps optimize read()/write()
(long)fileStat.st_blocks : Number of disk blocks allocated to the file.
ctime(&fileStat.st_atime) : when the file was last read.

ctime(&fileStat.st_mtime) : when file content last changed.

ctime(&fileStat.st_ctime) : when metadata (permissions, ownership, links) last changed.

--P_10---

Commands :

od :
Octal Dump
(but it can also show hex, chars, etc.).
It shows the raw contents of a file in
a readable way.
-Default: octal representation.
-With
-c: show characters (printable ASCII or \0, \n, etc.).
-With -x: show
hexadecimal.
? Example:
echo "ABC" > f.txt

od -c f.txt

-------------------- C Programming :
----

open("holefile.txt", O_RDWR | O_CREAT | O_TRUNC, 0644)

O_TRUNC :
->If the file already exists ? its contents are erased
(truncated to length 0).
->If the file does not exist ? nothing to truncate,
but since O_CREAT is also specified, a new empty file will be created.
->File
permissions come from the 3rd argument (0644 here) when creating a new file.

lseek(fd,
10, SEEK_CUR); lseek return value (new offset)
// move file pointer forward by 10 bytes
(creating a hole)
lseek(int fd, off_t offset, int whence);
=>Purpose

-Moves (or queries) the file offset (also called the "file pointer").

-The file offset tells the kernel where in the file the next read() or write() will happen.

=>Parameters
-fd ? file descriptor (from open()).
-offset ?
number of bytes to move.
-whence ? starting position, one of:

-SEEK_SET ? from beginning of file.
-SEEK_CUR ? from current file offset.

-SEEK_END ? from end of
file.

--P_11---------------------------

-------------------- C Programming :
----

dup(fd) :
->dup(oldfd) creates a new file
descriptor that refers to the same open file description as oldfd. It returns the

lowest-numbered unused
file descriptor on success, or -1 on error.

dup2(fd, 10) :
-> If fd is invalid ? returns -1 with errno = EBADF.

-> If fd == 10 ? nothing changes, return 10 immediately.
-> If 10 is

already open ? it is closed first (to avoid resource leaks).
-> 10 is then
made a duplicate of fd.
-Now fd and 10 share the same open file description
(file offset, O_APPEND, etc.).
-Writes/reads on one affect the other.

fcntl(fd, F_DUPFD, 20) :
-> Duplicates fd into a new file descriptor.

-> The new file descriptor will be greater than or equal to start_fd (20 in this case).
-> Kernel finds the lowest available fd ? 20 and assigns it.

-> Both descriptors (fd and newfd) point to the same open file description (same offset, same flags).

--P_12---

-------------------- C Programming : --------------------

- fcntl(): is a Swiss army knife for file descriptors.
- It can duplicate, modify, query, or lock file descriptors.
- It is often used in:
  - Non-blocking I/O
  - File locking
  - Descriptor duplication (like in shells, pipes, redirection)
  - Signal-driven I/O

=> flags = fcntl(fd, F_GETFL);

->Get file status flags using fcntl
->Duplicating File Descriptors

int newfd = fcntl(fd, F_DUPFD, 20);
->File Descriptor Flags

F_GETFD ? Get per-FD flags (like FD_CLOEXEC).
F_SETFD ? Set per-FD flags.
Example:
fcntl(fd, F_SETFD, FD_CLOEXEC);
->File Status Flags

F_GETFL ? Get file status flags (set during open()).
Includes access mode (O_RDONLY, O_WRONLY, O_RDWR) and status flags (O_APPEND, O_NONBLOCK, O_SYNC, etc.).

F_SETFL ? Change file status flags.
int flags = fcntl(fd, F_GETFL);

fcntl(fd, F_SETFL, flags | O_NONBLOCK)

Summary of Common Commands :

Command	Action
`F_DUPFD`	Duplicate fd ? given number
`F_DUPFD_CLOEXEC`	Duplicate fd with `FD_CLOEXEC`
`F_GETFD`	Get fd flags (like `FD_CLOEXEC`)
`F_SETFD`	Set fd flags
`F_GETFL`	Get file status flags (`O_APPEND`, `O_NONBLOCK`, etc.)

| `F_SETFL` | Set file status flags
|
| `F_GETLK` | Get lock info |

| `F_SETLK` | Set lock (non-blocking) |
| `F_SETLKW`
| Set lock (blocking) |
| `F_SETOWN` | Set
owner process for async I/O |
| `F_GETOWN` | Get owner
process for async I/O |

=> accmode = flags & O_ACCMODE :

-> // Extract access mode (O_RDONLY / O_WRONLY / O_RDWR)
-> O_ACCMODE is a
mask defined in <fcntl.h>.
-> flags & O_ACCMODE extracts only the access
mode bits (ignores other flags like O_APPEND).

accmode == O_RDONLY --> "Read
only\n"
accmode == O_WRONLY --> "Write only\n"
accmode == O_RDWR
--> "Read/Write\n"

flags & O_APPEND --> "O_APPEND flag is
set\n"
flags & O_NONBLOCK --> "O_NONBLOCK flag is set\n"
flags
& O_SYNC --> "O_SYNC flag is set\n"
flags & O_CREAT -->
"O_CREAT flag is set (only relevant at open
time)\n"

-------------------- C Programming :
----

fd_set readfds; ---> fd_set is implemented as a bit
array.
->fd_set readfds; — a bitmap structure used by select() to represent a
set of file descriptors to monitor for readability.
struct timeval tv;

->struct timeval tv; — timeout structure with seconds (tv_sec) and microseconds
(tv_usec).

FD_ZERO(&readfds); ---> clears the set of file descriptors so you can
safely add the ones you want to monitor with select().
-> This initializes
(clears) the fd_set structure.
-> It sets all bits in readfds to 0, meaning no
file descriptors are being monitored initially.
-> clears the fd set
(initializes to empty).

FD_SET(STDIN_FILENO, &readfds); ---> sets the bit
corresponding to fd to 1 && add fd to monitor.
-> adds STDIN (file
descriptor 0) to the set of descriptors select() should watch for readability.

-> After this, readfds represents the set {0}.
-> FD_SET(fd, &fdset) is a
macro defined in <sys/select.h>.
-> It adds the file descriptor fd to the
file descriptor set fdset.
-> In your case:
- STDIN_FILENO ?
this is 0 (standard input).
- readfds ? the set of FDs you want select() to

monitor for readability.
-> So this line means:
"Monitor standard input (keyboard input) for readability in the select() system call."

tv.tv_sec = 10;
tv.tv_usec = 0;
->Sets the timeout to 10.0 seconds. tv_sec = whole seconds, tv_usec = microseconds.
->Note: On many systems select() may modify tv to reflect remaining time — so if you plan to reuse tv in a loop, reset it before each select() call.

fflush(stdout);
->ensures the message appears before select() blocks (stdout is often line-buffered only when connected to terminal, but flushing is a safe practice).

retval = select(STDIN_FILENO + 1, &readfds, NULL, NULL, &tv);
->select(nfds, &readfds, NULL, NULL, &tv) waits until one of the monitored file descriptors is ready for reading, or until the timeout expires.

FD_ISSET(STDIN_FILENO, &readfds):
->tests if STDIN was marked ready by select().

read(STDIN_FILENO, ...) to actually read the available bytes.

read()
returns:
- n > 0 = number of bytes read,
- n == 0 = EOF (e.g., user pressed Ctrl-D or input closed),
- n == -1 = error.

->select() modifies its fd_set and timeval arguments. If you want to call select() again, you must reinitialize readfds and tv before each call.
->EOF behavior:
If the user sends EOF (Ctrl-D on an empty line), select() will report STDIN as readable and read() will return 0 (EOF).
Your code currently ignores n == 0 case, so nothing will be printed; you might want to handle EOF explicitly.
->Signals: If a signal interrupts select() it returns -1 and errno == EINTR. Many programs retry select() in that case.

->Atomicity & line discipline: When STDIN is a terminal, input is typically line-buffered — the kernel won't mark STDIN readable until the user hits Enter.

->Alternatives: For large numbers of descriptors or high-performance servers, poll(), epoll (Linux) or kqueue (BSD/macOS) are often preferred.

--P_14--

-------------------- C Programming : --------------------

fprintf()
->used to print formatted output into a file (or stream).
->Similar to printf(), but instead of always writing to stdout (screen),

you can write to any file or stream (FILE *).
printf() --> always prints to stdout.
sprintf() --> writes formatted text into a string buffer.
fprintf() --> writes formatted text into a file or stream.

S_ISREG(st.st_mode) -->
Type: Regular file\n"
S_ISDIR(st.st_mode) -->   Type:
Directory\n"
S_ISCHR(st.st_mode) -->   Type: Character device\n"

S_ISBLK(st.st_mode) -->   Type: Block device\n"

S_ISFIFO(st.st_mode) -->   Type: FIFO/pipe\n"
S_ISLNK(st.st_mode)
-->   Type: Symbolic link\n"
S_ISSOCK(st.st_mode) -->   Type:
Socket\n"
else -->   Type:
Unknown\n"

--P_15--

-------------------- C Programming : --------------------

extern char **environ;
->environ is a global variable defined by libc

```
char **env = environ;
while (*env) {
    printf("%s\n",
*env);
    env++;
} :
```

-> loop continues while *env is not NULL.

-> Prints the current environment string (format: KEY=VALUE) followed by a newline.

--P_16--

-------------------- C Programming : --------------------

======================== reader.c ========================

struct flock lock;
->Defines lock of type struct flock, which describes a file lock (type, region, etc.).

Fill the struct flock for READING :
- l_type = F_RDLCK --> request a read lock (shared lock: multiple readers allowed, but no writers).
- l_whence = SEEK_SET --> offset is relative to the start of the file.
- l_start = 0 --> lock starts at byte 0.
- l_len = 0 --> lock to the end of the file (special meaning: "lock entire file").

fcntl(fd, F_SETLKW, &lock) == -1 :
- fcntl() with command F_SETLKW = "Set Lock, Wait":
- Tries to apply the lock (lock struct).
- If another process holds a conflicting lock (e.g., a write lock), this call blocks until it becomes available.

getchar();

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

- Changes
the lock type to F_UNLCK (unlock).
- Calls fcntl() with F_SETLK (non-blocking) to
release the lock.

======================= writer.c =======================

Fill in the fields of the struct flock:
- l_type = F_WRLCK --> request a write
lock (exclusive). Only one writer is allowed; no other readers or writers can hold locks at the same time.
- l_whence = SEEK_SET --> lock start is relative to the beginning of the
file.
- l_start = 0 --> start from byte 0.
- l_len = 0 --> lock until the
end of the file (special meaning = "entire file").

fcntl(fd, F_SETLKW,
&lock) == -1
- F_SETLKW = Set Lock and Wait.
- If another process already
holds a lock that conflicts (e.g., another write lock or a read lock), this call will block
until the lock can be acquired.
- If fcntl fails, print error, close the file, and
exit.

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

-
Changes l_type to F_UNLCK ? unlock.
- Calls fcntl() with F_SETLK to release the
lock.
- This does not block; it just removes the
lock.

-------------------- C Programming : --------------------

======================= init_ticket.c =======================

write(fd, &ticket, sizeof(ticket)) == -1:

->Writes the value of ticket (which is 0) into the file.
->&ticket ?
address of the integer variable.
->sizeof(ticket) ? number of bytes to write
(usually 4 bytes for int).
->If write() fails, prints error, closes the file,
and exits.

======================= reserve_ticket.c =======================

```
  // Setup write lock
        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;

lock.l_start = 0;
        lock.l_len = 0; // Lock entire file
```

Configures a write lock:

```
   short l_type;    // Type of lock: F_RDLCK, F_WRLCK, F_UNLCK
        short l_whence;  // How
to interpret l_start (SEEK_SET (beginning) , SEEK_CUR, SEEK_END)
        off_t l_start;   //
Starting offset for lock
        off_t l_len;     // Number of bytes to lock (0 means till EOF)
```

```
        pid_t l_pid;     // PID of process holding lock (for F_GETLK)

    fcntl(fd,
F_SETLKW, &lock) == -1


    lseek(fd, 0, SEEK_SET);
        ->Moves the file
pointer to the beginning of the file.

    read(fd, &ticket, sizeof(ticket)) == -1 -->
Read current ticket number.
    ticket++;  --> inc ticket num

    write(fd, &ticket,
sizeof(ticket)) == -1 --> write incremented ticket number.

    lock.l_type = F_UNLCK;

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

--P_18----------------------

```
    --------------------     C Programming :
--------------------

    int lock_record(int fd, int recno, short lock_type)
        -fd –
an open file descriptor for the file you want to lock a record inside.
        -recno – the
record number (0,1,2,…) you want to lock.
        -lock_type – F_RDLCK (shared/read lock)
or F_WRLCK (exclusive/write lock).
        -Returns 0 on success, -1 on failure (and errno will
be set).

    {
        struct flock lock;          // file lock variable
        lock.l_type =
lock_type;    // F_RDLCK or F_WRLCK or F_UNLCK
        lock.l_whence = SEEK_SET;           //
l_start is offset from file start.
        lock.l_start = recno * sizeof(int); // offset to
record
        lock.l_len = sizeof(int);           // lock only this record
        lock.l_pid
= getpid();

        fcntl(fd, F_SETLKW, &lock) == -1
    }

    int unlock_record(int fd,
int recno)
        -fd – an open file descriptor for the file you want to lock a record
inside.
        -recno – the record number (0,1,2,…) you want to lock.
    {
        struct
flock lock;
        lock.l_type = F_UNLCK;
        lock.l_whence = SEEK_SET;

lock.l_start = recno * sizeof(int);     // offset to record
        lock.l_len = sizeof(int);

        lock.l_pid = getpid();

        fcntl(fd, F_SETLK, &lock) == -1
    }


off_t size = lseek(fd, 0, SEEK_END);
    off_t lseek(int fd, off_t offset, int whence);
```

```
    -> fd ? File descriptor (from open()).
            -> offset ? A number (how much
to move).
            -> whence ? Where to start measuring from. Common values:

 -> SEEK_SET ? from the beginning of file.
            -> SEEK_CUR ? from the current
position.
            -> SEEK_END ? from the end of
file.
```

--P_19---------------------------

```
        --------------------     C Programming :
--------------------

        static __inline__ uint64_t rdtsc(void)
            -> static
? restricts visibility to the translation unit (source file).
            -> __inline__
? suggests compiler to inline the function (replace call with code).
            -> Return
type: uint64_t ? 64-bit unsigned integer.

        __asm__ __volatile__("rdtsc" :
"=a"(lo), "=d"(hi));
            -> __asm__ __volatile__ ? inline
assembly that the compiler should not optimize away or reorder.
            ->
"rdtsc" ? the CPU instruction Read Time Stamp Counter.
            -> It reads
the number of cycles since the last reset of the CPU.
            -> The result is a 64-bit
value:
                -> Lower 32 bits ? stored in register EAX (=a constraint ?
variable lo).
                -> Upper 32 bits ? stored in register EDX (=d constraint ?
variable hi).

        Combining High and Low
            return  (uint64_t)hi << 32 |
lo;
                -> (uint64_t)hi << 32 ? shift the high part left by 32 bits.

            -> | lo ? OR with the low part.
                -> Together ? produce
full 64-bit timestamp.

        USE of RDTSC :
                -> rdtsc gives the
raw CPU cycle counter.
                -> Common uses:
                    ->
High-resolution performance measurement (benchmarking).
                    -> Profiling
code execution time at the CPU cycle level.
                    -> Implementing timers or
random number seeds.

        start = rdtsc();-> Read timestamp before

        pid =
getpid();-> Call getpid()

        end = rdtsc();-> Read timestamp after

------------------------------  Get Freq  ----------------------------------------

struct timespec ts_start, ts_end; // ts_start / ts_end: store wall-clock time before and after
a 1-second sleep.
        uint64_t start, end;               // start / end: store the CPU cycle
count from rdtsc() at those times.
```

clock_gettime(CLOCK_MONOTONIC, &ts_start); //
gets a monotonic timestamp (wall-clock time that never goes backward, not affected by system clock changes).
start = rdtsc(); // read the time-stamp
counter (number of CPU cycles since reset).

sleep(1); // wait 1 second

clock_gettime(CLOCK_MONOTONIC, &ts_end); // gets a monotonic timestamp (wall-clock time that never goes backward, not affected by system clock changes).
end = rdtsc();
// read the time-stamp counter (number of CPU cycles since reset).

// Add them to get total elapsed time as a double (in seconds).
double elapsed_sec =
(ts_end.tv_sec - ts_start.tv_sec) +
(ts_end.tv_nsec -
ts_start.tv_nsec) / 1e9; // Convert nanoseconds ? seconds (/ 1e9).
return (end -
start) / elapsed_sec; // Hz (cycles per
second)

--P_20--

-------------------- Commands:

./prio &:

=>The & at the end:

-> In
Linux/Unix shells, appending & tells the shell to run the command in the background.

-> This means:

-> The process starts executing, but your terminal
does not wait for it to finish.

-> You immediately get back your shell
prompt so you can type more commands.

=>ps command:

->Stands for process status.

->Shows information about running processes.

->Without options, ps just lists processes belonging to the current shell.

->With options, you can customize what info to show.

-o pid,pri,ni,comm:

->-o flag lets you specify which columns/fields to display.

->
pid : Process ID

-> pri : priority value

-> ni : Nice
value (user-space priority hint).

-> Range: -20 (highest priority) to 19
(lowest priority). Default is 0.

-> comm : command name that started the
process

->Example: prio.

=> nice command:

->
Used to start a program with a given nice value (process priority hint).

->
Syntax:

-> nice -n <value> command [args...]

-> If no
-n is given, it defaults to 10.

$nice -n 10 ./prio

=> -n 10
? sets nice value to +10.

=>./prio ? runs the executable prio from the current directory.

=> What is "nice value"?

->The nice value (NI) is a user-space concept that influences how much CPU time the scheduler gives to the process.
->Range: -20 ? highest priority (needs root privileges). 0 ? default priority. +19 ? lowest priority.
=>
Linux kernel computes an internal priority (PRI) based on base priority + nice value.

-------------------- C Programming : --------------------
infinite loop

--P_21---

-------------------- C Programming : --------------------

fork() :
-> It is a system call that creates a new process.
-> After this line, two processes exist:
Parent process ? The original process.
Child process ? A duplicate of the parent.
-> Both processes resume execution from the point of the fork() call.
-> The return value of fork() is how each process distinguishes itself:
In the parent process, fork() returns the child's PID (positive integer).
In the child process, fork() returns 0.
If an error occurs, it returns -1 in the parent (no child is created).

getpid() ? returns the process ID of the child itself.
getppid() ? returns the parent's process ID.

--P_22---

-------------------- C Programming : --------------------

```
fd = open("output.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
```
-> Open file (create if it doesn't exist, write-only, append)

```
pid = fork();
```
-> Fork a child process

```
pid == 0? write(fd, child_text, strlen(child_text)) : write(fd, parent_text, strlen(parent_text));
```

--P_23---

-------------------- C Programming : --------------------

```
pid = fork(); // Create a child process

// Child process
printf("Child process (PID = %d) is exiting...\n", getpid());
```

```
        // Parent process
        printf("Parent process (PID = %d) sleeping...\n",
getpid());
        printf("Child PID = %d will become a zombie until parent calls
wait()\n", pid);
        // Sleep to keep parent alive and child in zombie state

sleep(30);
```

---------------------- Commands: ----------------------

ps aux :
->This is a very common way to list processes in Linux:

->a ? show processes for all users (not just you).
->u ? show
processes in a user-oriented format (includes username, CPU%, MEM%, etc.).

->x ? include processes not attached to a terminal (daemons, background services).

->So ps aux lists all processes on the system, with lots of useful info.

| (pipe): [Important] [i/p Sequence of commands]
=> Sends the output of
the first command (ps aux) as input to the next command (grep Z).

grep Z :

->grep searches for lines containing Z.

? What is a Zombie Process?
-> A zombie
process is a process that has finished execution (it has exited), but still has an entry in the process table.
-> It remains because its parent process hasn't collected its exit
status yet using wait() or waitpid().
? In ps output, its state appears as Z (Zombie) and
often as <defunct> in the command column.

? Life Cycle of a Normal Process
->
Parent creates a child using fork().
-> Child executes some code (maybe runs another
program via exec).
-> Child finishes and calls exit().
-> The kernel saves the
child's exit status in the process table.
-> Parent calls wait() or waitpid().

-> Kernel gives the exit status to the parent and then removes the child's entry ?
child is fully gone.
? Life Cycle of a Zombie
-> Parent creates a child with fork().

-> Child finishes execution and calls exit().
-> Kernel marks it as zombie and
keeps its entry in the process table (to store exit code).
-> Parent does not call
wait() (maybe by mistake or bad programming).
-> The child stays as a zombie (status Z)
forever until parent dies or finally collects the status.
? Why Kernel Keeps Zombies?

-> The exit status contains info like:
-> Exit code (success/failure).
->
Resources used (CPU time, etc.).
-> Parent needs this info (via wait()), so the kernel
cannot throw it away immediately.
? Problems with Zombies
->A single zombie is
harmless (it takes no CPU, just an entry in process table).
->Many zombies = process

table fills up = new processes cannot be created.
->Indicates buggy parent code (not reaping children properly).

? Summary:
-> A zombie process = "dead but not cleaned up".
-> It's created when a child exits, but the parent doesn't read its exit status.

---P_24---

-------------------- C Programming : --------------------

```
 pid = fork();  // Create a child process

    // Child process
    printf("Child process (PID = %d) running. Parent PID = %d\n", getpid(), getppid());
    sleep(10);  // Sleep so parent exits first
    printf("Child process (PID = %d) after parent exit. New Parent PID = %d\n", getpid(), getppid());

    // Parent process
    printf("Parent process (PID = %d) exiting...\n", getpid());
    exit(0);  // Parent exits immediately
```

Perfect ?, let’s clear up Orphan processes and compare them with Zombie processes.

? What is an Orphan Process?
-> An orphan process is a process whose parent has terminated (died) while the process itself is still running.
-> In Linux/Unix, when a parent dies, the orphan process is automatically adopted by init (PID 1, or nowadays systemd).

-> init becomes the new parent and is responsible for cleaning up when the orphan finishes.

? Life Cycle of an Orphan
-> A parent creates a child (via fork()).

-> Parent terminates without waiting for the child.
-> The child is still running ? becomes an orphan.
-> init (PID 1) adopts the orphan.
-> When the orphan finishes, init calls wait() ? no zombies left behind.

Aspect	**Zombie** ?	**Orphan** ?
Definition	Child finished, parent didn’t `wait()`.	Child still running, parent finished.
State in `ps`	`Z` (Zombie, `<defunct>`).	`S`, `R`, etc. (normal running/sleeping).
Parent	Still alive but ignoring child.	Dead, replaced by `init`.
Cleanup	Parent must `wait()`, else entry stays.	`init` reaps when child ends ? no permanent issue.
Problem	Can accumulate, filling process table.	Harmless, system handles automatically.

--P_25--

    --------------------   C Programming : --------------------

```
    pid1 = fork();
    pid2 = fork();
    pid3 = fork();
```

? Step 1:
wait() / waitpid() :

```
    waited_pid = waitpid(pid2, &status, 0);
    pid_t waited_pid =
waitpid(child_pid, &status, 0);
```

        waitpid() returns:
            >0 ? PID of the terminated child.
            0 ? no child has exited yet (only in non-blocking mode).

     -1 ? error (e.g., no children).

```
    if (waited_pid > 0) {
        if
(WIFEXITED(status)) {
            printf("Parent: Child 2 exited with status %d\n",
WEXITSTATUS(status));
        }
    }
```

? Step 2: status
    -> status is an integer that encodes information about how the child ended.
    -> We don't read it directly; instead, we use macros like WIFEXITED, WEXITSTATUS, WIFSIGNALED, etc.

? Step 3:
WIFEXITED(status)
    -> Returns true (nonzero) if the child terminated normally (via exit() or by returning from main).
    -> Returns false if the child was killed by a signal (e.g., SIGKILL).

? Step 4: WEXITSTATUS(status)
    -> If WIFEXITED(status) is true, then WEXITSTATUS(status) gives the exit code passed by the child.
        ->Example:

->exit(5);
    ? Parent sees WEXITSTATUS(status) == 5.

? Summary:
    -> waitpid() tells the parent which child ended.
    -> WIFEXITED(status) checks if child exited normally.
    -> WEXITSTATUS(status) extracts the exit code from the child.

--P_26--

    --------------------   C Programming : --------------------

Program A: use some executable program

```
    char *program =
"./a.out";      // Program to execute
    char *argv[] = {program, NULL}; // Argument
list (terminated with NULL)

    if(execv(program, argv) < 0) {
        perror("execv
```

failed");
        return 1;
    }

    -> execv() replaces the current process image
with a new one (here ./a.out).
    => On success:
        -> The original program code is
discarded.
        -> Memory is overwritten by the new program.
        -> Execution
restarts from the new program’s main().
        -> Importantly: execv() does not create a
new process. The PID stays the same, but the code and data are replaced.

    => On
failure:
        -> Returns -1.
        -> Sets errno.
        -> perror("execv
failed") prints the error message.
        -> Program exits with status 1.
    =>
This line only executes if execv() fails, because on success, the current process is completely
replaced by the new program image.
    => Normal termination if reached (which won’t
happen if execv() succeeds).

   ? Summary
        => This program demonstrates execv()
replacing the current process with another program.
        => Steps:
            - Print
the PID and the target program.
            - Call execv("./a.out", argv).

 - If it succeeds, the current process is replaced, so execution continues inside the new
./a.out.
            - If it fails, perror reports the error, and the program exits with code
1.
        => Key point: Unlike fork(), execv() does not create a new process. The process
is the same, but its code and memory are replaced.

Program B: pass some input to an
executable program. (for example execute an executable of $./a.out name)

```
        char *program
= "./a.out";         // Executable to run
        char *arg1 = "Shikhar";
         // Input argument
        char *argv[] = {program, arg1, NULL};  // Argument list
(NULL-terminated)

    // Replace current process with executable
    if(execv(program, argv)
< 0) {
        perror("execv failed");
        return 1;

}
```

--P_27------------------------------

    --------------------   C Programming :

P - 1 ;
    => execl("/bin/ls", "ls",
"-Rl", NULL);
        execl() is one of the exec family of functions.

Syntax:

```
            int execl(const char *path, const char *arg0, ..., NULL);
```

-> path ? full path of the program to run (/bin/ls).
-> arg0 ?
traditionally the program name (ls).
-> arg1, arg2, ... ? command-line
arguments (-Rl).
-> NULL ? marks the end of the argument list.

? Here:
Path = /bin/ls (actual program file on disk).
First argument
(argv[0]) = "ls".
Second argument = "-Rl".
Equivalent shell
command =

=> What Happens Inside execl?
execl() replaces the current
process image with the new program (/bin/ls).
-> That means:

-> The code of your program (main()) is overwritten.
-> From this point,
your program ceases to exist.
-> The process now becomes /bin/ls.

-> It does not return if successful.
So, after execl(), the process starts
executing ls -Rl inside the same process ID.

P - 2 :
=>char *envp[] = {
"PATH=/bin:/usr/bin", NULL };
- Here you define a custom environment for the
new program.
- Each string is of the form:
=> execle("/bin/ls",
"ls", "-Rl", NULL, envp);
- execle() is like execl(), but with
extra environment parameter.
=> int execle(const char *path, const char *arg0, ...,
NULL, char * const envp[]);
- path ? full path of the program (/bin/ls).
-
arg0 ? program name (ls).
- arg1 ? "-Rl".
- NULL ? marks the
end of arguments.
- envp ? environment variables for the new program.

P - 3 :

=> execlp("ls", "ls", "-Rl", NULL); // Uses PATH to find ls

- Calls execlp() from the exec family. Prototype (simplified): int execlp(const char *file,
const char *arg0, ..., (char *)NULL);
-> Behavior:
-> execlp searches
for the executable named by the first parameter ("ls") using the PATH environment
variable (so you don't need to provide /bin/ls explicitly).
-> The
remaining arguments form the new process's argv[] list. By convention argv[0] is the
program name — here also "ls".
-> "-Rl" is passed as
argv[1] (it is equivalent to -R -l for ls).
-> The argument list must be
terminated with NULL.
-> On success: the current process image is replaced by
the ls program. Execution does not return to this program; ls runs in the same PID, inheriting
open file descriptors and environment.
-> On failure: execlp returns -1 and
errno is set.

P - 4 :
    => char *argv[] = { "ls", "-Rl", NULL };
Entry point.
        -> Defines an argument vector argv:
            -> argv[0] =
"ls" ? by convention, the first element is the program name.
            ->
argv[1] = "-Rl" ? option to ls.
            -> argv[2] = NULL ? terminates the
array (required).
    => execv("/bin/ls", argv); // argv array, first element =
program name
        => Calls execv() with:
            -> The absolute path /bin/ls ?
no PATH lookup (unlike execlp).
            -> The argument vector argv.

        =>
Behavior:
            -> If successful, the current process image is replaced by /bin/ls.

         -> PID stays the same, but code, data, and stack are replaced.
            ->
Execution does not return to this program — instead, it starts running the new program's main().
    ? Summary:
        -> This program prints a message, then replaces itself with
/bin/ls -Rl. If successful, you'll only see the ls output — the perror and return 1 lines won't execute.

P - 5 :

    => char *argv[] = { "ls", "-Rl", NULL };

       Program entry point.
        Defines the argument vector argv for the new program:

      ->argv[0] = "ls" ? by convention, the program name.

->argv[1] = "-Rl" ? options passed to ls (recursive, long format).

->argv[2] = NULL ? marks the end of the argument list.

    =>
execvp("ls", argv); // Uses PATH to find ls
        Calls execvp() to replace the
current process with a new one running ls.
    => Parameters:
        "ls" ? the
filename to execute (does not need a full path).
        argv ? the argument vector (same as
would be passed in main(int argc, char *argv[])).
    => execvp() behavior:
        ->
Uses the PATH environment variable to locate the ls executable (so it finds /bin/ls or /usr/bin/ls).
        -> If successful:
            -> The current process image is
replaced by ls.
            -> The PID remains the same.
            -> Control never
returns to this program — instead, ls starts running immediately.
        -> If it fails
(e.g., "ls" not found in PATH), it returns
-1.

--P_28--

    --------------------   C Programming :

    => Maximum and minimum priority for SCHED_FIFO
        ->

max_fifo = sched_get_priority_max(SCHED_FIFO);
-> min_fifo = sched_get_priority_min(SCHED_FIFO);

sched_get_priority_max(SCHED_FIFO) ? returns the highest priority value available for the SCHED_FIFO scheduling policy.

sched_get_priority_min(SCHED_FIFO) ? returns the lowest priority value available for the same policy.

? These functions are necessary because the numeric range of real-time priorities is platform-dependent. On Linux, typically:
(but you shouldn't hard-code these values, hence the API calls).

=> Maximum and minimum priority for SCHED_RR

-> max_rr = sched_get_priority_max(SCHED_RR);
-> min_rr = sched_get_priority_min(SCHED_RR);
Same thing, but for the Round-Robin real-time policy.

On Linux, SCHED_FIFO and SCHED_RR usually share the same priority range (1 to 99).

? Key concepts
SCHED_FIFO:
-> First-In, First-Out scheduling for real-time tasks.
-> A task runs until it voluntarily yields, blocks, or is preempted by a higher-priority task.
-> No time slicing between tasks of equal priority.

SCHED_RR:
-> Round-Robin scheduling for real-time tasks.
-> Similar to SCHED_FIFO, but tasks of the same priority take turns (each gets a timeslice).
Priorities:

-> Real-time scheduling policies use integer priorities.
-> Higher number = higher priority.
-> Range (on Linux): 1–99.
-> Normal (time-sharing) processes with SCHED_OTHER all use priority 0 (their “niceness” value is separate from real-time priorities).
? Summary:
-> This program queries the OS for the valid priority ranges for the two main real-time scheduling policies (SCHED_FIFO and SCHED_RR) and prints them.

-------------------- C Programming : --------------------

```
    pid_t pid = getpid();  // Current process ID
    int policy;

struct sched_param param;
```
->sched_param ? a struct that holds scheduling parameters. Its main field is sched_priority (the priority of the process under real-time policies).

```
    -> policy = sched_getscheduler(pid);
    if(policy == -1) {
        perror("sched_getscheduler failed");
        return 1;
    }
```

sched_getscheduler(pid) ? queries the current scheduling policy of the given process (here, pid = current process).

Returns one of:
-> SCHED_OTHER ? normal time-sharing policy (default for most processes).
-> SCHED_FIFO ? first-in, first-out real-time scheduling.
-> SCHED_RR ? round-robin real-time scheduling.
If it fails, prints an error.

-> Set new scheduling policy to SCHED_FIFO with maximum priority

-> param.sched_priority = sched_get_priority_max(SCHED_FIFO);
-> sched_get_priority_max(SCHED_FIFO) ? returns the maximum priority allowed for the SCHED_FIFO policy (usually 99 on Linux).

-> Sets param.sched_priority to that value.

->
```
if(sched_setscheduler(pid, SCHED_FIFO, &param) == -1) {
perror("sched_setscheduler failed");
        return 1;
    }
```

-> sched_setscheduler(pid, SCHED_FIFO, &param) ? attempts to set the scheduling policy of this process (pid) to SCHED_FIFO with the given priority.
-> Requires root (superuser) privileges. If run as a normal user, you'll get:

? Key Takeaways
-> Default Linux policy: Processes usually run under SCHED_OTHER (normal, time-sharing).
-> Real-time policies: SCHED_FIFO and SCHED_RR require root privileges because they can hog the CPU if misused.
-> Priorities:
SCHED_OTHER ignores sched_priority.

SCHED_FIFO / SCHED_RR use a range of 1–99 (higher = more urgent).
-> Effect: If successful, your process will run with real-time priority, which means it could starve normal processes.

-------------------- C Programming : --------------------

=> umask(0); // Set file permissions mask
-> umask(0) removes any default file permission restrictions.
-> Daemon-created files will have exactly the permissions requested.

=> sid = setsid();
-> setsid() creates a new session and makes this process the session leader.
-> This detaches the process from the controlling terminal ? crucial for daemons.
-> Now it runs independently.

=> chdir("/"); // Commented out to allow relative paths
Normally, a daemon changes working directory to / to avoid blocking file systems from being unmounted.
-> Here it's commented out so the daemon can still use relative paths.

=> Close standard
file descriptors
close(STDIN_FILENO);
close(STDOUT_FILENO);

close(STDERR_FILENO);
-> A daemon doesn’t need a terminal.
->
Closing these prevents accidental reads/writes to terminal.

=> Daemon loop:

```
while (1) {
            time_t now = time(NULL);
            struct tm *t = localtime(&now);

       }
```

-> Infinite loop: the daemon keeps running forever.
->
time(NULL) ? current time in seconds.
-> localtime() ? convert to local time
(hours, minutes, etc.).

```
        // Set target time (for testing, a minute ahead)

      int target_hour = t->tm_hour;
            int target_minute = (t->tm_min + 1) %
60;
```

-> For demonstration: it sets the target time to one minute ahead of
current time.
-> target_hour = current hour.
->
target_minute = current minute + 1 (wraps around with % 60).

```
        if (t->tm_hour ==
target_hour && t->tm_min == target_minute) {
        // Run the script (absolute
path, no spaces)
        system("/home/shikhar/myscript.sh");

            // Avoid
multiple runs in the same minute
            sleep(60);
        }
        else {

sleep(10);
        }
```

-> Checks if current time matches the scheduled time.

-> If yes ? runs the script /home/shikhar/myscript.sh.
-> Then sleeps for 60
seconds so it doesn’t run multiple times in the same minute.
-> Otherwise, sleeps
for 10 seconds and checks again.

? Key Concepts
Daemonization steps:

-> Fork and exit parent.
-> Create new session (setsid).

-> Optionally change directory to /.
-> Set umask(0).
-> Close
standard file descriptors.
Daemon behavior:
-> Runs in background
without terminal.
-> Independent of user sessions.
-> Keeps
looping, checking system time, and executing tasks.
In this program:
->

It executes a script exactly one minute later, then every hour/minute condition can be extended.
-> Works like a simplified custom cron service.