Experiment -1 Linux Commands

1. ls (List Directory Contents)

- **Purpose**: Lists files and directories.
- **Example**:
  - ls — List files.
  - ls -l — Detailed list (permissions, size, etc.).

2. cd (Change Directory)

- **Purpose**: Changes the current directory.
- **Example**:
  - cd Documents — Go to the "Documents" directory.
  - cd .. — Move up one directory level.

3. pwd (Print Working Directory)

- **Purpose**: Displays the full path of the current directory.
- **Example**:
  - pwd — Shows current directory path (e.g., /home/user/Documents).

4. mkdir (Make Directory)

- **Purpose**: Creates a new directory.
- **Example**:
  - mkdir new_folder — Creates "new_folder".

5. rmdir (Remove Directory)

- **Purpose**: Removes an empty directory.
- **Example**:
  - rmdir old_folder — Deletes "old_folder" if empty.

6. cp (Copy)

- **Purpose**: Copies files or directories.
- **Example**:
  - cp file1.txt file2.txt — Copies "file1.txt" to "file2.txt".

7. mv (Move or Rename)

- **Purpose**: Moves or renames files.
- **Example**:
  - mv file1.txt new_folder/ — Moves "file1.txt" to "new_folder".
  - mv oldname.txt newname.txt — Renames a file.

8. rm (Remove)

- **Purpose**: Removes files or directories.
- **Example**:
  - rm file.txt — Deletes "file.txt".
  - rm -r folder — Deletes a folder and its contents.

9. touch (Create Empty File)

- **Purpose**: Creates an empty file or updates the timestamp.
- **Example**:
    - touch newfile.txt — Creates "newfile.txt".

10. cat (Concatenate and Display)

- **Purpose**: Displays file content.
- **Example**:
    - cat file.txt — Shows content of "file.txt".

11. head (Display Beginning of File)

- **Purpose**: Shows the first few lines of a file.
- **Example**:
    - head -n 5 file.txt — Displays the first 5 lines.

12. tail (Display End of File)

- **Purpose**: Displays the last few lines of a file.
- **Example**:
    - tail -n 10 file.txt — Displays the last 10 lines.

13. grep (Search Text in Files)

- **Purpose**: Searches for a pattern in files.
- **Example**:
    - grep "hello" file.txt — Searches for "hello" in the file.

14. chmod (Change File Permissions)

- **Purpose**: Changes file permissions.
- **Example**:
    - chmod 755 file.txt — Sets full permissions for the owner, read-execute for others.

15. chown (Change File Ownership)

- **Purpose**: Changes file ownership.
- **Example**:
    - chown user:group file.txt — Changes ownership of the file.

16. ln (Create Links Between Files)

- **Purpose**: Creates hard or symbolic links.
- **Example**:
    - ln -s /path/to/file linkname — Creates a symbolic link to a file

3. File Manipulation with System Calls

File Manipulation :- Means there are different types of operation that we can perform in the file.
Example – we create file , delete file, write some content in file , Some update inside the file.

System calls : - System call is a way for a user program to request services from O.S. since user program or instruction cannot directly interact with hardware or the kernel.

A **file descriptor** is just a number assigned by the system when you open a file.

How to compile and Run file

Firstly check the GCC installed in your system or not **gcc --version**

Install the GCC

sudo apt install gcc –y
sudo apt update //for update.

gcc readfile.c -o readfile //Command to compile file

./readfile to executable file.

Gcc →Calls the **GNU C Compiler** to compile C programs.
readfile.c →The **source code file** (your C program).
-o →Tells the compiler to specify an **output file name**.
readfile →The **name of the output executable file**.

Here some common system calls :

1. open() – Open a File

Purpose: Opens a file for reading, writing, or both.

Example:

Int fd = open("file.txt", O_CREAT | O_WRONLY, 0644);

O_CREAT → Create the file if it doesn't exist.

O_WRONLY → Open for writing only.

0644 → Set file permissions.

2. close() – Close a File

Purpose: Closes an open file to free system resources.

Example:

close(fd);

Always close files after use to prevent memory issues.

3. read() – Read Data from a File

Purpose: Reads data from a file into memory.

Example:

char buffer[100];
read(fd, buffer, 100);

Reads 100 bytes from fd (file descriptor) into buffer.

4. write() – Write Data to a File

Purpose: Writes data from memory to a file.

Example:

```
write(fd, "Hello, System Call!", 20);
```

Writes 20 characters to the file.

5. lseek() – Move the File Pointer

Purpose: Moves the reading/writing position in a file.

Example:

```
lseek(fd, 10, SEEK_SET);
```

Moves 10 bytes forward from the start of the file.

Useful for editing specific parts of a file.

6. unlink() – Delete a File

Purpose: Removes a file from the system.
Example:

```
unlink("file.txt");
```

Deletes file.txt permanently.

7. rename() – Rename a File

Purpose: Changes a file's name.

Example:

```
rename("old.txt", "new.txt");
```

Renames old.txt to new.txt

8. stat() – Get File Information

Purpose: Fetches file details like size, permissions, and last modified time.

Example:

```
struct stat fileStat;
stat("file.txt", &fileStat);
printf("File size: %ld bytes\n", fileStat.st_size);
```

Shows the file size in bytes.

9. chmod() – Change File Permissions

Purpose: Modifies who can read, write, or execute a file.

Example:

chmod("file.txt", 0777);

Makes file.txt readable, writable, and executable by everyone.

10. chown() – Change File Owner

Purpose: Changes the owner of a file.

Example:

chown("file.txt", 1001, 1001);

Changes owner and group to user ID 1001.

11. link() – Create a Hard Link

Purpose: Creates another name (link) for a file.

Example:

link("file.txt", "file_link.txt");

Now file_link.txt points to the same data as file.txt.

12. symlink() – Create a Soft Link (Shortcut)

Purpose: Creates a symbolic link (shortcut) to another file.

Example:

symlink("file.txt", "file_symlink.txt");

file_symlink.txt behaves like a shortcut to file.txt.

Example for open , create file and write in system call

```
#include <fcntl.h>                                  // For open()
#include <unistd.h>                                 // For read(), write(), close()

int main() {
 int fd = open("file.txt", O_CREAT | O_WRONLY, 0644);
```

`open()` is used to **create or . open** a file.
`"file.txt"` → The name of the file.
`O_CREAT` → **Create the file if it doesn't exist .**

`O_WRONLY` → **Opens the file in write-only mode** (no reading

`0644` → **File permission //**

```
 write(fd, "Hello, System Call!\n", 21);
```

`write(fd, buffer, size)` **writes data to the file.**

`fd` → **File descriptor (the file**

to write to).

`"Hello, System Call!\n"` →

The text to write.

`21` → **The number of bytes**

(characters) to write, including `\n`**.**

```
  close(fd);  // Close file
```

```
  return 0;
}
```

Example read the file in system calls.

```
#include<stdio.h>
#include <fcntl.h>                      // For open()
#include <unistd.h>                     // For read(), write(), close()

int main() {
 int fd;                                // File descriptor

   char buffer[100];
                                        // Buffer to store file content, This is an array (temporary storage) to store the file's content while reading.

   fd = open("file.txt", O_RDONLY);     // Open file in read mode

   if (fd < 0) return 1;                // If error, exit

                                        // If fd < 0, that means the file couldn't be opened
                                        return 1; stops the program with an error.


   int bytes = read(fd, buffer, sizeof(buffer));  // read(fd, buffer, sizeof(buffer)) reads from the file into buffer.
                                                  It stores how many bytes were actually read in the variable bytes.

   write(1, buffer, bytes);             //write(1, buffer, bytes) writes the read data to the screen.
                                        1 is the file descriptor for the screen to hold data content.

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

OR (If you want to create ,write and read in single program.)

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

int main() {
 int fd;
  char buffer[100];

   // Create and write to the file
   fd = open("file.txt", O_CREAT | O_WRONLY, 0777);
   write(fd, "Hello, System Calls!\n", 21);
   close(fd);
   // Open and read the file
   fd = open("file.txt", O_RDONLY);
   read(fd, buffer, sizeof(buffer));
   write(1, buffer, sizeof(buffer));    // Print to terminal (stdout)
   close(fd);
   return 0;
}
```

Example of Lseek

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

int main() {
 int fd = open("file.txt", O_RDWR | O_CREAT, 0644);

//open("file.txt", O_RDWR | O_CREAT, 0644);
```

- Opens **"file.txt"**.
- **O_RDWR** → Open for **reading and writing**.
- **O_CREAT** → Create the file if it **does not exist**.
- **0644** → Sets **file permissions** (Owner: Read & Write, Others: Read).
- **Returns fd (file descriptor)** or -1 if an error occurs //

```
lseek(fd, 0, SEEK_END);  // Move to end of file
```

//Moves the file pointer to the end of the file.

- **lseek(fd, 0, SEEK_END);**
- fd → File descriptor of "file.txt".
- 0 → Moves **0 bytes** from the reference point.
- **SEEK_END** → Sets position **at the end of the file**.
- **This ensures the next write() appends data at the end.//**

```
write(fd, "XYZ", 3);  // Write "XYZ" at the end
```

// Writes "XYZ" (3 characters) at the end of the file.

- "XYZ" is **appended** to the existing content.//

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

===Delete file In system call=======

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

int main()
{
//

unlink("fileum.txt"); → file name – “fileum.txt
return 0;
}
```

====== Rename any file in system call =========

```
#include <stdio.h>
 int main()
{
rename("1.txt", "1new.txt");
return 0;
}
```

==========Lseek==========

1 Example on lseek.

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

```
int main()
{
int fd;
char buffer[20];

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

write(fd, "Hello, world",13);

lseek(fd, 0, SEEK_SET);    // lseek() → Moves the file pointer.

read(fd, buffer,13);

lseek(fd,6, SEEK_SET);       //SEEK_SET is a constant used in the lseek() function to set the file pointer to a specific position from the beginning of the file.

write(fd, "chatgpt", 7);
close(fd);
return 0;
}
```

After running the program, `testfile.txt` will contain: → Hello ,chatgpt

Example -2 on lseek

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

int main()
{
int fd = open("testfile1.txt", O_RDWR | O_CREAT, 0666);
lseek(fd,0, SEEK_END);                    //SEEK_END → Moves the pointer relative to the end of the file.
write(fd, "Kumar" ,5);
lseek(fd,5,SEEK_END);

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

=======Stat========

```
#include <stdio.h>
#include <sys/stat.h>   //#include <sys/stat.h> → Includes file status information functions, such as stat().

int main()
{
struct stat s;   //The stat structure stores information about a file, such as:st_size → File size (in bytes).


stat("testfile.txt", &s);
printf("size: %ld bytes\n", s.st_size);   //s.st_size contains the file size in bytes
return 0;
}
```

========chmod============

```
#include <stdio.h>
#include <sys/stat.h>
int main()
{
chmod("testfile.txt", 0777); // Set full read, write, execute permissions
printf("Permissions changed.\n");
return 0;
```

}

=====Hard Link (`link()` system call)=======

```
#include <unistd.h>

int main() {
   link("original.txt", "hardlink.txt");
   return 0;
}
```

=======Symbolic (Soft) Link (`symlink()` system call)======

```
#include <unistd.h>

int main() {
   symlink("original.txt", "symlink.txt");
   return 0;
}
```

4. Directory Manipulation using System Calls

Function	Description	Header
mkdir()	Create a new directory	<sys/stat.h>
rmdir()	Remove a directory	<unistd.h>
chdir()	Change the current working directory	<unistd.h>
getcwd()	Get the current working directory	<unistd.h>
opendir()	Open a directory	<dirent.h>
readdir()	Read a directory	<dirent.h>
closedir()	Close a directory	<dirent.h>

Open and Read Directory (`opendir()`, `readdir()`, `closedir()`)

```
#include <stdio.h>

#include <dirent.h>

int main() {

   DIR *d = opendir(".");   //that is used to open directory in c.

   struct dirent *e;    // that is the pointer which is declare e to store directory entry , in c programming.

   while ((e = readdir(d)) != NULL)

      printf("%s\n", e->d_name);     //  e->d_name struct dirent  member (field) that store file and folder name.

   closedir(d);

}
```

Change Directory (`chdir()`)

```
#include <unistd.h>

int main() {

   chdir("/home");

}
```

Create Directory (`mkdir()`)

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

int main() {

   mkdir("newdir", 0777);

}
```

Remove Directory (`rmdir()`)

```
#include <unistd.h>

int main() {

   rmdir("newdir");

}
```

Get Current Directory (`getcwd()`)

```
#include <stdio.h>

#include <unistd.h>

int main() {

   char path[100];

   getcwd(path, sizeof(path));

   printf("%s\n", path);

}
```

Question - C Program to Demonstrate the Use of getcwd, chdir, and rmdir System Calls.

```
#include <stdio.h>
#include <unistd.h>
int main() {
   char cwd[1024];

   // Get current directory, change to /home, and print the new directory
   if (getcwd(cwd, sizeof(cwd)) && printf("Current dir: %s\n", cwd) && chdir("/home") == 0 && getcwd(cwd, sizeof(cwd))
&& printf("New dir: %s\n", cwd) == 0 && rmdir("empty_directory") == 0) {
```

```
        printf("Removed empty directory.\n");
    } else {
        printf("Error occurred.\n");
    }
return 0;
}

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

int main() {
    char cwd[1024];

    // Get current directory, change to /home, and print the new directory
    if (getcwd(cwd, sizeof(cwd)) && printf("Current dir: %s\n", cwd) && chdir("/home") == 0 && getcwd(cwd, sizeof(cwd))
&& printf("New dir: %s\n", cwd) == 0 && rmdir("empty_directory") == 0) {
        printf("Removed empty directory.\n");
    } else {
        printf("Error occurred.\n");
    }

    return 0;
}
#include <stdio.h>
#include <unistd.h>

int main() {
    char cwd[1024];

    // Get current directory, change to /home, and print the new directory
    if (getcwd(cwd, sizeof(cwd)) && printf("Current dir: %s\n", cwd) && chdir("/home") == 0 && getcwd(cwd, sizeof(cwd))
&& printf("New dir: %s\n", cwd) == 0 && rmdir("empty_directory") == 0) {
        printf("Removed empty directory.\n");
    } else {
        printf("Error occurred.\n");
    }

    return 0;
}
```

5. Process Management using System Calls

Process Management is an important function of the operating system that helps in creating,exeuting, controlling and , executing, controlling and terminating processes.

Process Management is a key function of the OS that controls **process creation, execution, and termination**.

fork() **Creates a new process**

exec() **Replaces current process with another program**

wait() **Makes the parent process wait for the child process**

exit() **Terminates a process**

getpid() **Returns the Process ID (PID) of the current process**

getppid() **Returns the Parent Process ID (PPID)**

kill() **Terminates a process forcefully**

nice() **Sets process priority**

fork()-Creating a New Process

- fork() is used to **create a new child process** from the parent process.
- The child process is an exact copy of the parent process

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

int main() {
   fork();  // Create a child process

   printf("Process ID: %d\n", getpid());
   return 0;
}
```

exec() – Replacing a Process

- `exec()` is used to **replace the current process with another program**.
- Example: Running the `ls -l` command from a C program.

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("Executing 'ls -l' command..\n");
   execl("/bin/ls", "ls", "-l", NULL);
   printf("exec() failed!\n");  // This will execute only if exec fails
   return 1;
}
```

getpid() & **getppid()** – Getting Process ID

- `getpid()` returns **the process ID (PID) of the current process**.
- `getppid()` returns **the parent process ID (PPID)**.

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("Current Process ID: %d\n", getpid());
   printf("Parent Process ID: %d\n", getppid());
   return 0;
}
```

sleep() – Delaying Execution

- `sleep(seconds)` **pauses execution for the given time**.

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("Sleeping for 3 seconds...\n");
   sleep(3);
  printf("Process woke up!\n");
   return 0;
}
```

exit() – Terminating a Process

- `exit()` is used to **terminate a process safely**.
- Used to terminate the **current process**.
- Directly **exits** the program without sending any signal.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   printf("Process is terminating...\n");
   exit(0);  // Successful termination, 0 indicates successful termination.
}
```

wait() – Waiting for Child Process

- `wait()` makes the parent **wait until the child process finishes execution**.
- Without `wait()`, the parent may finish first, leaving the child as an orphan process.
- If pid > 0, we are in the **parent** process.
- If pid == 0, we are in the **child** process.
- If pid < 0, an error occurred (not handled in this code).

```
#include <stdio.h>
#include<unistd.h>
#include<sys/wait.h>
int main()
{
int pid = fork();
if(pid>0)          //If pid > 0, this is the parent process.
{
wait(NULL);    //makes the parent wait for the child to complete before proceeding.
printf("child process completed\n");
}
else
{
printf("child process running\n");
}
return 0;
```

```
}
```

nice() – Changing Process Priority

- `nice(value)` **adjusts the priority** of a process.
- **Lower value → Higher priority.**

```
#include <stdio.h>
#include <unistd.h>
int main() {
   int priority = nice(5);  // Increases priority value
   printf("New priority: %d\n", priority);
   return 0;
}
```

Kill()

- Used to terminate **another process**.
- **Sends a signal** to the target process to stop it.
- Can forcefully terminate a process using `kill(pid, SIGKILL);`.

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

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

   if (pid == 0) {                          // Child process
      printf("Child running with PID: %d\n", getpid());
      sleep(5);                        // Sleep so parent can kill it
      printf("Child process exiting...\n");
   } else {                               // Parent process
      sleep(2);
      printf("Killing child process...\n");
      kill(pid, SIGKILL);  // Forcefully terminate the child process
   }
   return 0;
}
```

6- Creation of Multithreaded Processes using PThread Library

A Thread is smallest unit of execution in a Process. Process can have multiple threads ,Threads can help the program run faster by doing multiple things at once.

Single thread = One task at a time.

Multiple threads = Multiple tasks at the same time to improve performance by utilizing cpu efficiently. make programs **faster** and **more responsive.**

`pthread` stands for **POSIX (Portable Operating System Interface) Threads**, which is a library used in C and C++ programming to allow a program to run **multiple threads** at the same time.

POSIX provides facilities for:

- **File operations** (like opening, reading, writing files)
- **Process management** (starting, stopping programs)
- **Thread management** (running tasks at the same time)
- **Input/Output (I/O)** operations (like printing things to the screen)

Compilation and execute threads.

gcc pthread_create.c –o pthread_create -pthread

-pthread: This is flag , It tells the compiler to include pthread support when compiling the program.

Where is the Main Thread Created?

When we run a C program, the operating system automatically creates the main thread. This thread runs inside the main() function. So, in a C program, the main thread is the thread that executes the main() function.

How to Identify the Main Thread?

We can use the **pthread_self()** function to check which thread is the main thread.

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

int main() {
   printf("Main Thread ID: %lu\n", pthread_self());
   return 0;
}
```

Commonly used library functions related to POSIX threads (pthread) :-

1. **pthread_create () -** Create a new thread

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

void* thread_function(void* arg) {
   printf("Thread is running\n");
   return NULL;
}
int main() {
   pthread_t   thread_id;

   // Create the thread
   pthread_create(&thread_id, NULL, thread_function, NULL);

   printf("Created thread ID: %lu\n", thread_id);
```

```
//lu -> print for Unsigned long integer with format specifier.

  // Wait for the thread to finish
  pthread_join(thread_id, NULL);

  printf("Main thread completed\n");

  return 0;  }
```

2- pthread_join - Wait for termination of a specific thread

Syntax - int pthread_join(pthread_t thread, void **retval);

3 - **pthread_exit()**

- **pthread_exit()** is a command that makes the **current thread** stop running.
- It doesn't stop the entire program, just the thread that calls it.

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

// Function the thread will run
void* thread_function(void* arg) {
printf("Thread is running!\n");

// Stop the thread here
pthread_exit(NULL);

// This will NOT run
printf("This won't be printed!\n");
}

int main() {
pthread_t thread_id;

// Create the thread
pthread_create(&thread_id, NULL, thread_function, NULL);

// Wait for the thread to finish
pthread_join(thread_id, NULL);

printf("Main thread is done!\n");
return 0;}
```

4 - `pthread_cancel()`

It is used to **cancel** another thread that is running.

It **asks** the target thread to stop. However, it doesn't immediately stop the thread, it will stop when

it,reaches a **cancellation point** (a safe point where it can stop).

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>  // For sleep()

void* thread_function(void* arg) {  // Function that the thread will run
printf("Thread is running!\n");
sleep(5);                           // Simulate work by sleeping for 5 seconds
printf("Thread finished!\n");
return NULL;
 }

int main() {
pthread_t thread_id;

pthread_create(&thread_id, NULL, thread_function, NULL);  // Create the thread

sleep(2);                            // Wait for 2 seconds

pthread_cancel(thread_id);           // Cancel the thread after 2 seconds

pthread_join(thread_id, NULL);       // Wait for the thread to finish

printf("Main thread done!\n");
return 0;}
```

[&thread_id-This is a pointer to a `pthread_t` variable that will store the thread ID of the newly created thread.
`NULL`, it means the thread will use the **default attributes**.
thread_function -This is a pointer to the **function** that the new thread will execute.
Null – argument passed to the thread_funtion)]

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

void* thread_function(void* arg) {
   static int value = 42;  // Static so it stays valid after function returns
   return &value;         // Return pointer to the value
}

int main() {
   pthread_t thread;
   int* result;

   pthread_create(&thread, NULL, thread_function, NULL);
   pthread_join(thread, (void**)&result);  // Get pointer from thread

   printf("Value returned from thread: %d\n", *result);

   return 0;
```

}

Create a pthread program to find the length of strings passed to the thread function.

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

// Thread function to find string length
void* find_length(void* arg) {
   char* str = (char*)arg;
   static int len;        // static to return pointer
   len = strlen(str);
   return &len;
}

int main() {
   pthread_t thread;
   char str[] = "Hello, pthread!";
   int* length;

   pthread_create(&thread, NULL, find_length, str);     // Pass string to thread
   pthread_join(thread, (void**)&length);              // Get returned length

   printf("Length of string: %d\n", *length);
   return 0;
}
```

Experiment 7: Process Synchronization using Semaphore/ Mutex

To compile process synchronization use –lpthread flag.

Synchronization

- Coordination of concurrent **threads/processes** to ensure **orderly execution**.
- Prevents inconsistent access to **shared resources**.

Race Condition

- Occurs when multiple threads/processes access **shared resources** without proper synchronization.
- Leads to **unpredictable or incorrect results** due to timing issues.

Semaphore

Definition: synchronization establish between processes or threads, semaphore is abstract data type.

Types:

- **Binary Semaphore** (0 or 1) – like a mutex.
- **Counting Semaphore** – allows access to multiple instances.

Functions:

Function	Description	Syntax
sem_init	Initialize semaphore	int sem_init(sem_t *sem, int pshared, unsigned int value);
sem_wait	Wait (decrement/block)	int sem_wait(sem_t *sem);
sem_post	Signal (increment/unblock)	int sem_post(sem_t *sem);
sem_destroy	Destroy semaphore	int sem_destroy(sem_t *sem);

Usage: Manages **critical sections**, prevents **race conditions**.

Mutex (Mutual Exclusion)

Definition: Ensures **exclusive access** to a shared resource – synchronization establish between threads.

Types:

- **Recursive Mutex**: Same thread can lock multiple times.
- **Non-recursive Mutex**: Deadlocks if locked multiple times by the same thread.

Functions:

Function	Description	Syntax
pthread_mutex_init	Initialize mutex	int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *att
pthread_mutex_lock	Lock mutex (wait)	int pthread_mutex_lock(pthread_mutex_t *mutex);
pthread_mutex_unlock	Unlock mutex	int pthread_mutex_unlock(pthread_mutex_t *mutex);
pthread_mutex_destroy	Destroy mutex	int pthread_mutex_destroy(pthread_mutex_t *mutex);

Usage: Ensures **mutual exclusion**, protects **shared data**, prevents **simultaneous access**.

Avoid Race condition using semaphore.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

int x = 0;
sem_t sem;  // Declare a semaphore

void* task() {
   sem_wait(&sem);    //  Acquire lock (enter critical section)
   x = x + 1;

   sem_post(&sem);    //  Release lock (exit critical section)
   return NULL;
}
```

```
    int main() {
       sem_init(&sem, 0, 1);  // Initialize semaphore with value 1

       for (int i = 0; i < 500; i++) {
          x = 0;  // Reset before each run

          pthread_t a, b;
          pthread_create(&a, NULL, task, NULL);
          pthread_create(&b, NULL, task, NULL);
          pthread_join(a, NULL);
          pthread_join(b, NULL);

          if (x != 2) {
             printf("Race condition detected! x = %d\n", x);  ;
    } else {
             printf(" x = %d\n", x);
          }
       }

  sem_destroy(&sem);  // Cleanup

return 0;

    }
```

Simulating race condition

```
    #include<stdio.h>
    #include<pthread.h>
    int x=0;
    void* task()
    {
    x=x+1;
    }
    int main()
    {
    int i;
    for (i = 0; i < 1000; i++) {
          x = 0;  // Reset before each run

    pthread_t a,b;
    pthread_create(&a,NULL,task,NULL);
    pthread_create(&b,NULL,task,NULL);
    pthread_join(a,NULL);
    pthread_join(b,NULL);
    if (x != 2) {
             printf("Race condition found x = %d\n", x);
          } else {
             printf(" x = %d\n", x);
    }       }
    return 0;
    }
```