LINUX

* **Thread** :- Threads are lightweight processes that can run concurrently within a single program. They share the same memory space but execute independently, making them ideal for tasks that requires parallilism.
  - **Single-threaded** :- One sequence of instructions executes at a time.
  - **Multi-threaded** :- Multiple threads executes different parts of the program simultaneously.

* **Why Threads** :-
  - **Efficiency** :- Threads uses less memory and resources than full-fledged processes.
  - **Parallelism** :- Tasks like sorting large datasets, performing calculations, or handling multiple I/O operations can be done faster using threads.
  - **Responsiveness** :- In GUI applications, threads can keep the interface responsive while performing background tasks.

* **Thread Creations** :- Linux provides several libraries and APIs to create and managed threads, the most common being the POSIX Threads (pthreads) library.
  - **Creating a Thread** :-

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

void *thread_function (void *arg) {
    printf ("Hello from the thread !\n");
    return NULL ;
}
```

```
int main () {
    pthread_t thread ;
    pthread_create (& thread , NULL , thread_function , NULL);
    pthread_join (thread , NULL) ; // wait for the thread to
                                      finish
    return 0 ;
}
```

Explaination :-

- `pthread_create ()` :- Creates a new threads . It takes the thread ID , thread attributes, the function to be executed by the thread , and the arguments passed to the function .
- `pthread_join ()` :- Waits for the thread to finish its execution .

★ Thread Synchronization :- Threads often need to access shared resources like variables , files , or data structures . Without proper synchronization , this can lead to race conditions .

- Mutexes :- Used to ensure that only one thread accesses a shared resources at a time .

```
#include <pthread.h>
#include <stdio.h>
int counter = 0 ;
pthread_mutex_t lock ;
void * increment (void * arg) {
  pthread_mutex_lock (& lock) ; // lock the mutex
  counter ++ ;
  printf ("Counter : %d\n", counter) ;
  pthread_mutex_unlock (& lock) ;
  return NULL ;
}
```

```
int main () {
    pthread_t thread1, thread2 ;
    pthread_mutex_init (&lock, NULL)
    pthread_create (&thread1, NULL, increment, NULL);
    pthread_create (&thread2, NULL, increment, NULL);
    pthread_join (thread1, NULL);
    pthread_join (thread2, NULL);
    pthread_mutex_destroy (&lock);
    return 0 ;
}
```

- Explanation :-

'pthread_mutex_lock()' :- Acquires the mutex lock.

'pthread_mutex_unlock()' :- Release the mutex lock.

★ Intex-Process Communication (IPC) :- It refers to mechanism provided by the operating system that allows processes to communicate and synchronize their actions.

★ Why IPC :-

- Data Sharing :- Processes can exchange data.
- Synchronization :- Processes can coordinate their actions
- Modularity :- Complex tasks can be divided among multiple processes.

★ Types of IPC :-

- Pipes :-

1) Named Pipes :- (FIFOs) Allow unrelated processes to communicate.

2) Unamed Pipes :- Used for communication between related processes, typically parent and child processes.

- Example of Pipe :-

```
#include <stdio.h>
#include <unistd.h>
int main () {
  int fd [2];
  pipe (fd);  // create a pipe
  if (fork () == 0) {
     close (fd [0]);  // close unused read end
     write (fd[1], "Hello from child", 17);
     close (fd [1]);
  } else {
     char buffer [20];
     close (fd [1]); // close unused write end
     read (fd [0], buffer, 17);
     close (fd [0]);
   }
   return 0;
}
```

Explanation :-

- 'pipe (fd)' :- Creates a pipe. 'fd [0] is for reading, and 'fd [1]' is for writing.
- 'fork ()' :- Creates a new process. The child process writes to the pipe, and the parent process reads from it.

- **Shared Memory** :- Allows multiple processes to access the same memory segment.

Example :-

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
int main () {
   int shmid = shmget ( IPC_PRIVATE , 1024 , 0666 | IPC_CREAT);
   char *str = (char*) shmat (shmid , NULL , 0) ;
   if (fork() == 0) {
    sprintf (str , "Hello from shared memory") ;
    shmdt (str) ;
   } else {
      wait (NULL) ;
      printf ("Data read from shared memory : %s\n", str);
      shmdt (str) ;
      shmctl (shmid , IPC_RMID , NULL) ;
    }
    return 0 ;
  }
```

Explanation :-

- '**shmget()**' :- Allocates a shared memory segment.
- '**shmat()**' :- Attaches the shared memory segment to the process's address space.
- '**shmdt()**' :- Detaches the shared memory.
- '**shmctl()**' :- Controls the shared memory (e.g., to remove it)

- Message Queues :- Allow processes to send and receive message in a queue.
- Semaphores :- Used for synchronization, controlling access to shared resources by multiple processes.
- Sockets :- Used for network communication between processes on the same or different machines.

★ Race Condition :- A race condition in thread synchronization occurs when multiple thread access and manipulate shared data concurrently, and the outcome of their operations depends on the order in which the threads are executed by the CPU. Because thread execution order is unpredictable and can vary from one run to another, this can lead to inconsistent, incorrect, or unexpected results.

By using mutexes or other synchronization techniques, you can prevent race conditions and ensures that shared data is manipulated safely by concurrent threads.

★ Conclusion :- Threads and IPC are essential concepts in Linux, enabling efficient parallelism and communication between processes. By understanding and utilizing these mechanisms, you can build complex, efficient, and responsive applications.

★ Difference between Mutex and Semaphores :-

Mutex	Semaphores
1. It is a locking mechanism used to synchronize access to a resource. A thread needs to lock the resource and unlock it as well.	1. It is a signalling mechanism. It uses wait() and signal() calls.
2. It is an object.	2. It is an integer variable.
3. It allows multiple threads to access the same resources but not concurrently.	3. It allows multiple processes to access the finite instance of resources.
4. It can be released only by the thread that has locked it.	4. It can be released by any process acquiring or releasing the resource.
5. It does not have different categories.	5. Semaphores are of two types: Binary and counting semaphores.

HAPPY LEARNING ☺