Threads and Inter-Process Communications

★ Threads :- Threads are light-weight 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 parallism.

- 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-fledge processes.
- Paralelism :- 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 API's to create and managed threads, the most common being the POSIX Threads (pthread) library.

- Creating a Threads :-

```
# 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); // waits for the thread to
                                   finish
    return 0;
}
```

Explanation :-

- '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 threads to finish its execution.

★ Thread Synchronization :- Threads often needs 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.

☆ Inter-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 co-ordinate 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) Unnamed 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;
}
```

fd[0] :- Reading
fd[1] :- Writing

Explanation :-

- 'pipe(fd)' :- Creates a pipe.
- '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 processes address space.
- 'shmdt()' :- Detaches the shared memory.
- 'shmctl()' :- Controls the shared memory (eg., 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 ensure 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.

☆ Differences 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 mechanisms. It uses wait() and signal() calls.
2. It is an object.	2. It is an integer value.
3. It allows multiple threads to access the same resources but not concurrently.	3. It can be released by any process acquiring or releasing the resource.
4. It can be released only by the thread that has locked it.	4. It allows multiple processes to access the finite instance of resource.
5. It does not have different categories.	5. Semaphores are of two-types :- Binary and counting semaphores.

☆ Real-Time examples of Mutex, Semaphore and IPC implementations :

1. Odd-Even Number Printing :- Ensures alternate printing of even and odd numbers using mutex for synchronization.
2. Barber Shop Problem :- Simulates a barbershop where customers wait for the barber or leave if the shop is full, managed using semaphore.
3. Bank Account Management System :- Manages deposits and withdrawals on a shared bank account using mutex to prevent race conditions.

Problem 1 :- Even Odd Mutex Sync

- Problem Description :

1. Two threads print even and odd numbers alternatively using shared memory.
2. Even thread : Prints even numbers when the flag is 0.
3. Odd thread : Prints odd numbers when the flag is 1.

- Code :-

```
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

#define MAX 10
int flag = 0 ;  // Flag to switch between even and odd threads
pthread_mutex_t m1 ;  // Mutex to ensure mutual exclusion

// Function for the even number printing thread
void * even_thread (void* arg) {
    int* shm = (int*) arg  // Pointer to shared memory
    while (*shm < MAX) {
        pthread_mutex_lock (&m1) ;  // Lock the mutex
        if (flag == 0) {  // check if it is the even thread's turn
            if ((*shm) % 2 == 0) {  // check if the number is even
                printf ("even %d\n", *shm) ;
                (*shm)++ ;  // Increment the shared value
            }
            flag = 1 ;  // switch the flag to odd
        }
        pthread_mutex_unlock (&m1) ;  // unlock the mutex
        usleep (100) ;  // small delay to prevent busy waiting
    }
```

```
    return NULL ;
}
// Function for the odd number printing thread
void* odd_thread (void* arg) {
    int* shm = (int*) arg ; // Pointing to shared memory
    while (*shm < MAX) {
        pthread_mutex_lock (&mt) ;
        if (flag == 1) { // check if it's odd thread's turn
            if ((*shm) % 2 != 0) { // Check if the number is odd
                printf ("odd: %d\n", *shm) ;
                (*shm)++ ;
            }
            flag = 0 ; // switch the flag to even
        }
        pthread_mutex_unlock (&mt) ;
        usleep (100) ;
    }
    return NULL ;
}
int main () {
    int shmid ;
    int* shm ;
    // Create shared memory to store an integer
    shmid = shmget (IPC_PRIVATE, sizeof (int), IPC_CREAT | 0666) ;
    if (shmid == -1) {
        perror ("shmget failed") ;
        return 1 ;
    }
    // Attach shared memory to the process
    shm = (int*) shmat (shmid, NULL, 0) ;
    if (shm == (void*) -1) {
        perror ("shmat failed") ;
        return 1 ;
    }
    *shm = 0 ; // Initialize shared memory value to 0
```

```
    pthread_t p1, p2;
    // Create the even number printing thread
    pthread_create(&p1, NULL, even_thread, (void*)shm);
    // Create the odd number printing thread
    pthread_create(&p2, NULL, odd_thread, (void*)shm);

    // Wait for both threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_mutex_destroy(&m1); // Destroy the mutex
    shmdt(shm);                 // Detach shared memory
    shmctl(shmid, IPC_RMID, NULL); // Remove shared memory segment
    return 0;
}
```

Problem 2 :- Bank Transaction System

- Problem Description :-

1. Multiple customers perform deposits and withdrawls on a shared bank account.
2. Mutex ensures safe balance updates, preventing race conditions.
3. Withdrawls fails if the balance is insufficient.

Code :-

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_CUSTOMERS 5
#define INITIAL_BALANCE 1000
pthread_mutex_t mutex; // mutex for protecting balance
int accountBalance = INITIAL_BALANCE;
// Function for depositing money
void *deposit(void* arg) {
    int amount = *(int*) arg;
```

```
    pthread_mutex_lock (&mutex);
    accountBalance += amount;
    printf ("deposited: %d | NEW Balance: %d\n", amount, accountBalance);
    pthread_mutex_unlock (&mutex);
    return NULL;
}
// Function for withdrawling money
void* withdraw (void* arg) {
    int amount = *(int*) arg;
    pthread_mutex_lock (&mutex);
    if (accountBalance >= amount) {
        accountBalance -= amount;
        printf ("Withdrawn: %d | New Balance: %d\n", amount, accountBa-
                                                              lance);
    } else {
        printf ("insufficient balance for withdrawl: %d current
                Balance: %d\n", amount, accountBalance);
    }
    pthread_mutex_unlock (&mutex);
    return NULL;
}
int main () {
    pthread_t customerThreads [NUM_CUSTOMERS];
    int transactions [NUM_CUSTOMERS] = {200, 500, 300, 700, 100};
    int transactionTypes [NUM_CUSTOMERS] =
                                        {1, 0, 1, 0, 1};
                                    ⇓ transaction amount
    // initialize the mutex
    pthread_mutex_init (&mutex, NULL);
    // Create customer threads
    for (int i=0; i < NUM_CUSTOMERS; i++) {
        if (transactionTypes [i] == 1) {
            pthread_create (&customerThreads [i], NULL, deposit,
                                                    &transactions [i]
        } else {
```

```
        pthread_create (&customerThreads[i], NULL, withdraw,
                                                &transaction[i]);
    }
    sleep(1);
}
// Join customer threads
for(int i=0; i<NUM_CUSTOMERS; i++) {
    pthread_join (customerThreads[i], NULL);
}

// destroy the mutex
pthread_mutex_destroy (&mutex);
printf ("Final account balance: %d\n", accountBalance);
return 0;
}.
```

Problem 3 :- Barber Shop System

- Problem Description :-

1. Barber : Sleeps when no customer is waiting.
2. Customers : If the shop is full, they leave. Otherwise they wait or get a hair cut.
3. Chairs : Limited number of chairs available for waiting customers.

- Code :-

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

#define CHAIRS 3          // number of waiting chairs
#define CUSTOMERS 10      // number of customers

sem_t waitingChairs;      // semaphore for waiting chairs
sem_t barberReady;        // semaphore to signal barber is ready
sem_t customerReady;      // semaphore to signal customer is ready
pthread_mutex_t mutex;    // mutex for accessing waiting count
```

```
int waitingCount = 0 ;  // numbers of customers waiting

void* barber (void* arg) {
    while (1) {
        sem_wait (&customersReady); // wait for customer to arrive
        pthread_mutex_lock (&mutex); // reduce waiting customer count
        waitingcount -- ;
        printf ("barber is cutting hair. waiting customers : %d\n",
                                                waitingCount) ;
        pthread_mutex_unlock (&mutex);
        sem_post (&barberReady) ;  // barber is ready to cut
        sleep (2) ;   // simulate haircut time
        printf ("barber finished cutting hair. \n") ;
      }
      return NULL ;
  }
void * customer (void* arg) {
    int id = *(int*) arg ;
    pthread_mutex_lock (&mutex) ;
    if (waitingCount < CHAIRS) {
      waitingCount ++ ;
      printf ("customer %d is waiting. waiting customers : %d \n", id,
                                                waitingCount ) ;
      pthread_mutex_unlock (&mutex) ;
      sem_post (&customersReady) ; // notify barber that customer is
                                                ready
      sem_wait (&barberReady) ; // wait for barber to ready
      printf ("customer %d is getting hair cut. \n ", id ) ;
    } else {
                un
       pthread_mutex_lock (&mutex) ;
       printf ("customer %d left because the shop is full. \n", id) ;
    }
    return NULL ;
}
```

```
int main() {
    pthread_t barberThread;
    pthread_t customerThreads[CUSTOMERS];
    int customerIDs[CUSTOMERS];
    // Initialize semaphore and mutex
   sem_init(&waitingChairs, 0, CHAIRS);
   sem_init(&barberReady, 0, 0);
   sem_init(&customerReady, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    // create barber thread
    pthread_create(&barberThread, NULL, barber, NULL);

    // create customer threads
   for(int i = 0; i < CUSTOMERS; i++) {
       customerIDs[i] = i+1;
       pthread_create(&customerThreads[i], NULL, customer,
                                                &customerIDs[i]);
      sleep(1);   // delay to simulate random customer arrival
    }
    // join customer threads
   for(int i=0; i < CUSTOMERS; i++) {
       pthread_join(customerThreads[i], NULL);
    }
    // clean up resources
    pthread_cancel(barberThread);
    pthread_mutex_destroy(&mutex);
   sem_destroy(&waitingChairs);
   sem_destroy(&barberReady);
   sem_destroy(&customerReady);
    return 0;
}.
```

HAPPY LEARNING ☺