Ex No. 6 Mutual exclusion using Peterson's algorithm

Dr S.Rajarajan
SASTRA

Introduction

- **Peterson's algorithm** (or **Peterson's solution**) is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.
- Peterson's algorithm is for mutual exclusion between two processes only
- It was formulated by Gary L. Peterson in 1981

The Algorithm

- The algorithm uses two variables: `flag` and `turn`.
- A `flag[n]` value of `true` indicates that the process `n` wants to enter the critical section.
- Entrance to the critical section is granted for process P0 if `turn` is `0`.

```
bool flag[2] = {false, false};
int turn;
```

```
P0:      flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] && turn == 1)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] && turn == 0)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[1] = false;
```

Sample Peterson Program

Two threads are created and mutual exclusion enforced between them when they try to access the count variable

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

```
int flag[2];
int count;
int turn;
int ans = 0;
void lock_init()
{
   // Initialize lock by reseting the desire of  both the threads to acquire the locks.
   // And, giving turn to one of them.
   flag[0] = flag[1] = 0;
   turn = 0;
}
// Executed before entering critical section
void lock(int self)
{
   // Set flag[self] = 1 saying you want to acquire lock
   flag[self] = 1;
   // But, first give the other thread the chance to acquire lock
   turn = 1-self;
   // Wait until the other thread looses the desire
   // to acquire lock or it is your turn to get the lock.
   while (flag[1-self]==1 && turn==1-self) ;
}
// Executed after leaving critical section
void unlock(int self)
{
    // This will allow the other thread to acquire  the lock.
   flag[self] = 0;
}
   // A Sample function run by two threads created in main()
void* thread0(void *s)
{
   int i = 0;
   int self = (int *)s;
   while(1)
   {
      lock(self); // try to enter critical section
      // Critical section (Only one thread can enter here at a time)
      printf("P%d is in Critical Section\n", self);
      if(count<100)
         count++;
```

```
        printf("Value of count: %d\n",count);
        unlock(self); //leave critical section
    }
}
void *thread1(void *s)
{
    int i = 0;
    int self = (int *)s;
    while(1)
    {
        lock(self); // try to enter critical section
        // Critical section (Only one thread can enter here at a time)
        printf("P%d is in Critical Section\n", self);
        if(count>0)
            count--;
        printf("Value of count: %d\n",count);
        unlock(self); //leave critical section
        i++;
    }
}
int main()
{
    // Initialized the lock then create 2 threads
    pthread_t p1, p2;
    lock_init();
    // Create two threads (both run func)
    pthread_create(&p1, NULL, thread0, (void*)0);
    pthread_create(&p2, NULL, thread1, (void*)1);
    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    return 0;
}
```