# Mutex

```
#include<pthread.h>

pthread_mutex_t  mutex ;
```

this is a mutex which acts like a lock in order to eliminate the race condition. mutex is like a variable of pthread_mutex_t type. It helps in giving a correct answer when we use multiples threads as sometimes one threads pause for no reason and the other thread starts executing which can cause error. So, mutex acts as a lock and helps first thread in executing properly so that the other thread can execute afterwards.

```
void*  routine ()
{
    for (int  i=0 ; i < 1000000 ; i++)
    {
        pthread_mutex_lock (&mutex) ;
```

it is a simple function for initializing the mutex lock. It has only one parameter that is the pointer to the mutex variable.

```
            mail++ ;

        pthread_mutex_unlock (&mutex) ;
```

it is a simple function for ending the mutex lock. It takes only one parameter that is the pointer to the mutex variable.

```
}
```

```
int main ()
{
    pthread_t p1 , p2 ;
    pthread_mutex_init(&mutex , Null);
```

this function is used to initialize a mutex and takes only two arguments. The first argument is the pointer to the mutex variable and second is Null.

```
    if(pthread_create(&p1, Null, &routine, Null) != 0)
    {
        return 1 ;
    }
    if(pthread_create(&p2, Null, &routine, Null) != 0)
    {
        return 2;
    }

    if(pthread_join(p1, Null) != 0 and pthread_join(p2, Null) != 0)
    {
        return 3;
    }

    pthread_mutex_destroy(&mutex) ;
```

this function is used to destroy the mutex after initializing it and takes only one parameter. It takes the pointer to the mutex variable.

```
}
```

## How to create threads in a loop:

```
int main()
{
    pthread_t th[4];
```

create an array of threads. size will be the number of threads you want to make.

```
    pthread_mutex_init(&mutex, Null);
    int i;
    for(i=0 ; i<4 ; i++)
    {
        if(pthread_create(&th[i], Null, &routine, Null)!=0)
        {
            return 1;
        }
    }
```

we will not join the thread in this for loop ~~as~~ because the threads would then execute sequentially but we want multiple threads to execute at the same time. For this we join the threads in another for loop.

```
    for(i=0 ; i<4 ; i++)
    {
        pthread_join(th[i], Null);
    }
    pthread_mutex_destroy(&mutex);
}
```

## Get return value from thread :

```
void*   roll_dice()
{
    int value = (rand() % 6) + 1 ;
```

it will generate random values from 1 to 6. Also we have the function type as void * because the pthread wants us to have a void* type.

```
    int* result= malloc(sizeof(int));
    *result = value;
```

We cannot return a reference to a local variable because that local variable will be deallocated because it's on the stack. So, we have to dynamically allocate the pointer we intend to return.

```
    return (void*) result;
```

we must return a void pointer as it is the demand of thread. The result is an int pointer so we typecaste it to a void pointer

```
}
```

```
int  main ()
{
    int*  res ;
   srand ( time (Null)) ;

   pthread_t  th ;

   if (pthread_create (&th , Null, &roll_dice, Null) != 0)
   {
        return 1 ;
   }

   if (pthread_join (th, (void**) & res) != 0)
   {
        return 2 ;
   }
```

↑ the second parameter demands a 2d void pointer so we type cast it.

```
    cout << * res ;

    free (res) ;
```

this function is for deleting the dynamically allocated memory

```
}
```

# Introduction to Semaphores

⇒ Generally it is used to eliminate the race condition, when we are dealing with multiple threads / multiple processes occuring at a single time.

⇒ Semaphores provide a more organised way of controlling the interaction of multiple processes than simple variables.

⇒ A semaphore is an integer variable used by processes to send signals to other processes so that synchronization can be achieved.

⇒ A semaphore can only be accessed by the following two operations:

- P (wait or down)
- V ( signal or up) / post

⇒ The definition of wait is as follows:

```
Wait (Semaphore S)
{
    while (S <= 0)  // will do nothing
    {
      S-- ;
    }
}
```

=) The definition of signal is as follows:

```
Signal (Semaphore S)
{
    S++;
}
```

=) If one process is changing the value of a semaphore, no other process is allowed to make any change simultaneously.

=) A section of code or collection of operations in which only one process may be executing at a given time, is called critical section.

=) Consider a system where there are n processes. Each process has a segment of code called a critical section in which the process may be changing common variables, updating a table, writing into files etc. When such a system works, only one process may be allowed to execute within a critical section.

=) Each process must request permission to enter its critical section. The section of code implementing this request is called entry section. The critical section may be followed by a section of code known as exit section.

```
while (1)
{
    entry section;
    critical section;
    exit section;
}
```

⇒ Semaphores are used to solve the critical section problem for n process i.e. they help in achieving synchronization/ mutual exclusion.

```
{ while (1)
      wait (mutex);
        critical section;
        signal (mutex);
  }
```

⇒ In the above code, mutex is a shared semaphore that is initialized to 1.

⇒ Now, for example P0 executes wait operation on semaphore. It can enter its critical section because the value of mutex is 1. It will be decreased by 1 and P0 will enter its critical section. At this point, if P1 also tries to enter its critical section and executes wait operation, it will have to wait because the value of mutex is now 0. It will wait untill P0 execute signal operation and increases the value of mutex by 1. So, only one process at a time can enter its critical section, which gurantees mutual exclusion.

# Semaphores in coding

```
#include <semaphore.h>

#define THREAD_NUM 4
```

it creates a variable with a static value of 4.

```
sem_t semaphore
void* routine (void* args)
{
    sem_wait (&semaphore);
```

it acts as lock just like the mutex. It takes only one parameter i.e. pointer to the semaphore variable. Sem_wait basically checks the semaphore value if the semaphore value is 0 and can no longer be decremented, the thread is gonna wait ~~for~~ on that semaphore and if it is above 0 then ~~a~~ value will be decremented and its gonna continue the execution below (will move to the next line of code)

```
sleep(1);
cout<< * (int*) args
```

↑ converts void* into int~~o~~ value

```
sem_post (&semaphore);
```

it basically increments the value of semaphore. It takes only one parameter i.e. pointer to the semaphore variable. It does not wait or do anything else just simply increments the value.

```
        free(args);
}

int main()
{
    pthread_t th[THREAD_NUM];
    sem_init(&semaphore, 0, 1);
```

this function is used to initialize the semaphore variable. It takes three parameters : first is the pointer to the semaphore variable, second is giving 0 or 1, 0 is if we are using multiple threads and 1 is if we are using multiple processes and third is the initial value of parameter which you want to give to the semaphore.

```
    Int i;
    for (i=0 ; i<4 ; i++)
    {
        int* a = malloc(sizeof(int));
        *a = i;
        pthread_create(&th[i], Null, &routine, a);
```

this is how we ~~assign~~ create thread when the function has arguments.

```
    for(i=0 ; i<4 ; i++)
    {
        pthread_join(th[i], Null);
    }
```

sem_destroy(&semaphore);

it is used to destroy the semaphore. It takes only one parameter i.e. pointer to the semaphore variable.