Muten

→ A muten is a mutual enclusion device & is useful for protecting shared data structures from concurrent modifications, and implementing critical sections.

→ A muten has 2 possible states: unlocked (not owned by any thread) & locked (owned by one thread). It can never be owned by 2 different threads simultaneously.

→ A thread attempting to lock a muten that is already locked by another thread is suspended until the owner thread unlocks the muten.

→ Linux guarantees that race condition do not occur among threads attempting to lock a muten.

Muten Initialisation

Static Initialization:- In case where default muten attributes are appropriate, the following macro can be used to initialize a muten that is statically allocated.

```
pthread_muten_t mut = PTHREAD_MUTEX_INITIALIZER;
```

Runtime initialization:- In all other cases, we must dynamically initialize the muten ~~using pthread_m~~

```
int pthread_muten_init (pthread_muten_t *mptr,
                        const pthread_muten_attr_t *attr);
```

This func initialites the muten pointed by mptr according to the muten attributes specified in attr. If attr is NULL, default attributes are used.

In producer - consumer problem, the producer process produces information that is consumed by consumer process. To allow producer & consumer run concurrently, we must have a buffer that can be filled by the producer & emptied by the consumer. This buffer can be unbounded or bounded buffer. In bounded buffer, size of the buffer will be limited or fixed. So, the consumer must wait if the buffer is empty and the producer must wait if buffer is full.
When an item is added to or removed from the buffer, the buffer will be in inconsistent state. So, the processes must have exclusive access to the buffer. If a consumer thread or process arrives while the buffer is empty, it blocks until a producer adds a new item.

As we know that in mutex locks, if lock is acquired on shared memory then if other threads or threads process threads want to get lock that mutex then it will be the threads will be blocked. The mutex lock can only be unlocked by the locking thread.

Now, let the consumer thread wants to consume an item from the shared memory. So, it will acquire the mutex lock. But ~~as item of the consumer~~ But, if the shared memory is empty, i.e. no item to consume, then the consumer process will be blocked inside the critical section. So, now the producer can't place an item inside the empty buffer bcoz it is blocked by the mutex locked which is not going to be unblocked bcoz consumer itself is blocked.

The solution to such problem is that let the consumer sleeps & unrelease the mutex lock so that producer can place item in the shared memory.

Similar situation occurs when producer wants to place an item in a full buffer.
Such solns are called Condition variables.
A condition variable is a synchronization construct that allows threads to suspend execution & relinquish the processor until some condition is satisfied.

2 basic operations on condition variables are signal() & wait().

Signal(): wake up a sleeping thread on this condition variable
wait() :- release lock, go to sleep, reacquire lock after thread is awaken up.
So, we can say that a condition variable enable a thread to sleep inside a critical section. Any lock held by the thread is automatically released when the thread is put to sleep.

→ A mutex is for locking & a condition variable is for waiting.

```
pthread_cond_t empty_cond = PTHREAD_COND_INITIALIZER
pthread_cond_wait()
pthread_cond_signal()
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER
```

Consumer thread
1. pthread_mutex_lock(&mut);
2. while (buffcount == 0)
3. pthread_cond_wait(&empty, &mut);
4. Consume item
5. pthread_mutex_unlock(&mut);

Producer thread
1. pthread_mutex_lock(&mut);
2. Produce item
3. if (buffcount == 1)
4. pthread_cond_signal(&empty);
5. pthread_mutex_unlock(&mut);

If line (2) of consumer thread is true then the thread will go to sleep under the cond^n variable 'empty' & it will unlock the 'mut' variable. As soon as 'mut' is unlocked, the producer thread will enter in like its line (1).

If line (3) of producer is true then consumer thread under 'empty' variable will wake up due to line (4) of producer. Now, producer will run its line (5), so mutex will be unblocked which will be accquired by the consumer due to its wait() func^n at line (3).

Semaphores can be used as both mutex & condition variables.

There are 2 types of semaphores.

(i) named semaphores (ii) Unnamed Semaphores.

named semaphore :- They are created as files inside the /dev/shm directory. So, if the name of the semaphore is name1 then a file sem.name1 will be created inside /dev/shm directory.

the named semaphore are available on /dev/shm

→ they can be deleted by using rm sem.sem1
(i.e. semaphore name)

$ cd /dev/shm
$ ls
$ fuser -u Sem.Sem1
it will give the pid of the process which has opened this file right now.

$ ls -l -p pid
this will give the list of opened files by the process pid.

So, here we can see the binary semaphores also.