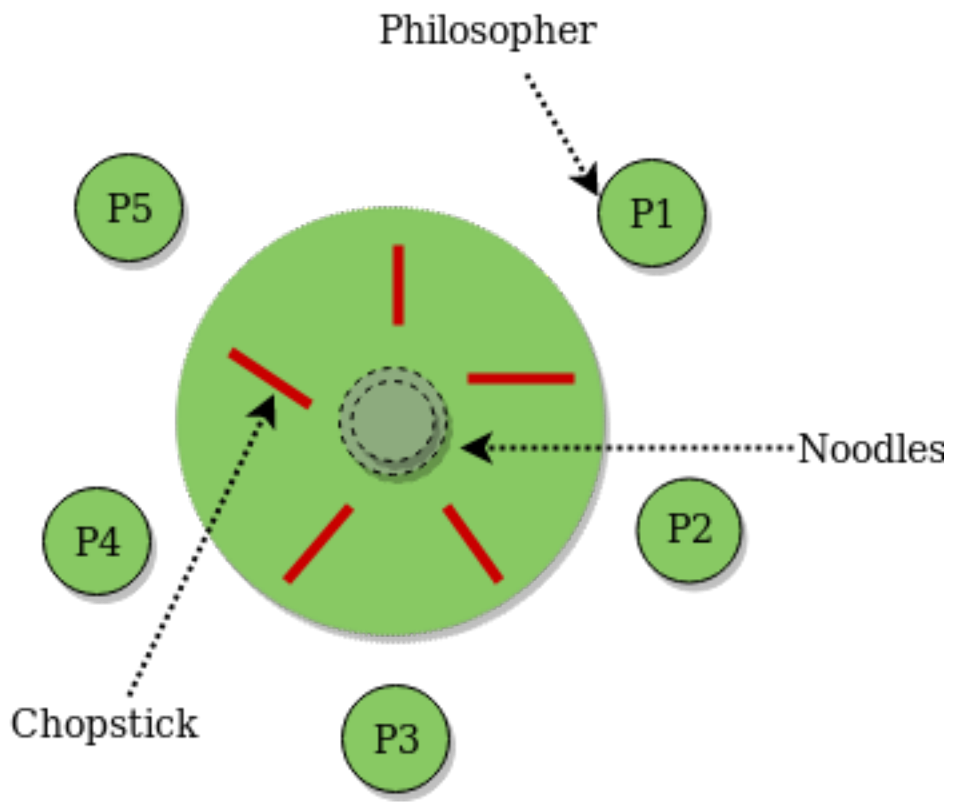
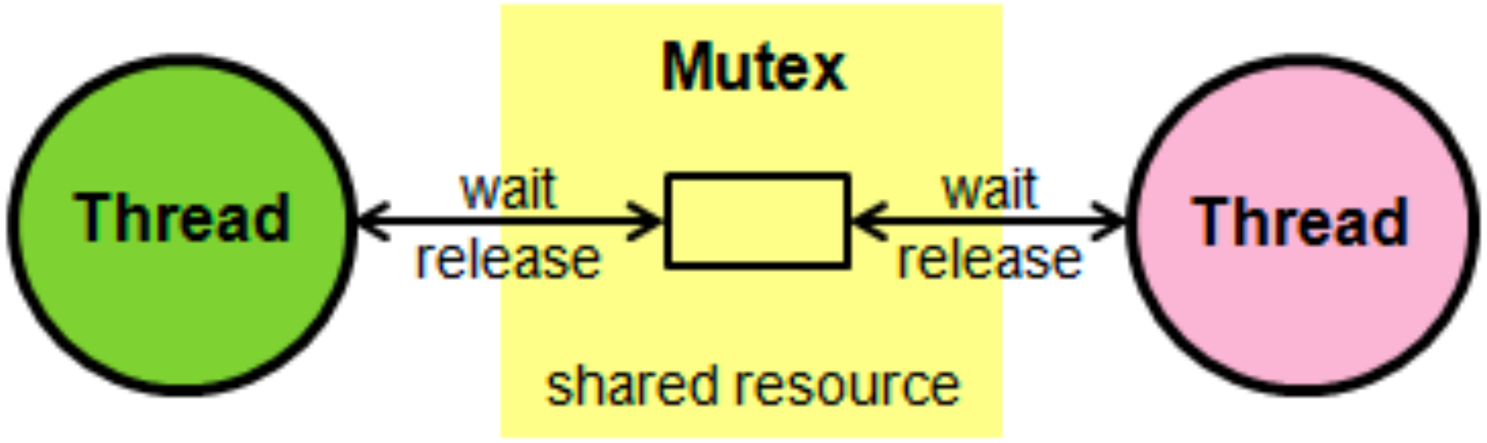
Dining Philosophers



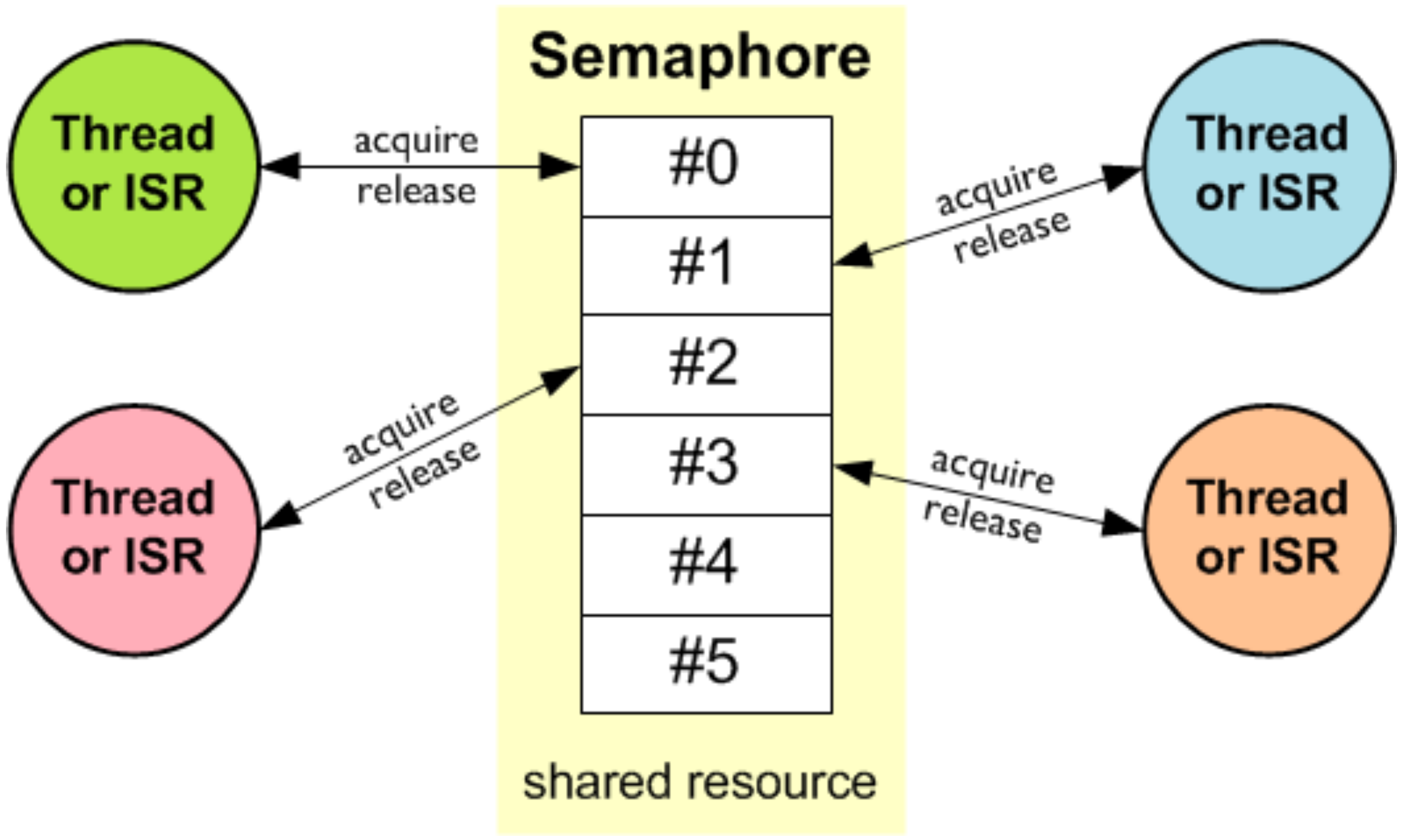
- Five philosophers spend their lives thinking and eating rice.
- There are only 5 chopstick on the table, one between every two philosophers.
- In order to eat, a philosopher needs to get hold of the two chopsticks that are closest to her.
- A philosopher cannot pick up a chopstick that is already taken by a neighbor.

Mutexes



- Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed. Mutexes are used for serializing shared resources.

Semaphore


Semaphore
Thread or ISR
acquire
release
#0
#1
#2
#3
#4
#5
Thread or ISR
acquire
release
Thread or ISR
acquire
release
Thread or ISR
acquire
release
shared resource

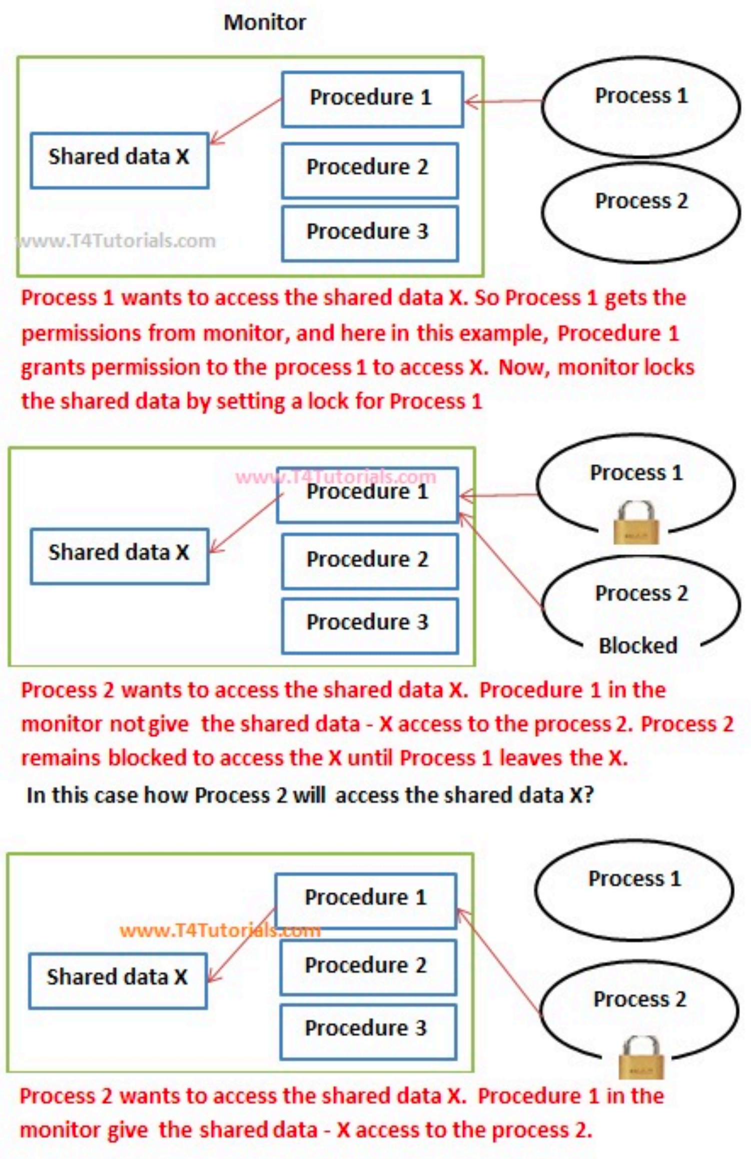
Monitors

```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor



Process 1 wants to access the shared data X. So Process 1 gets the permissions from monitor, and here in this example, Procedure 1 grants permission to the process 1 to access X. Now, monitor locks the shared data by setting a lock for Process 1

Process 2 wants to access the shared data X. Procedure 1 in the monitor not give the shared data - X access to the process 2. Process 2 remains blocked to access the X until Process 1 leaves the X.

In this case how Process 2 will access the shared data X?

Process 2 wants to access the shared data X. Procedure 1 in the monitor give the shared data - X access to the process 2.

Monitors

T1

```
int WaitForPredicate()
{
    // lock mutex (means:lock access to the predicate)
    pthread_mutex_lock(&mtx);

    // we can safely check this, since no one else should be
    // changing it unless they have the mutex, which they don't
    // because we just locked it.
    while (!predicate)
    {
        // predicate not met, so begin waiting for notification
        // it has been changed *and* release access to change it
        // to anyone wanting to by unlatching the mutex, doing
        // both (start waiting and unlatching) atomically
        pthread_cond_wait(&cv,&mtx);

        // upon arriving here, the above returns with the mutex
        // latched (we own it). The predicate *may* be true, and
        // we'll be looping around to see if it is, but we can
        // safely do so because we own the mutex coming out of
        // the cv-wait call.
    }

    // we still own the mutex here. further, we have assessed the
    //  predicate is true (thus how we broke the loop).

    // take whatever action needed.

    // You *must* release the mutex before we leave. Remember, we
    //  still own it even after the code above.
    pthread_mutex_unlock(&mtx);
}
```

T2

```
pthread_mutex_lock(&mtx);
TODO: change predicate state here as needed.
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mtx);
```

- #include <semaphore.h>
- sem_t sem;
- sem_init(&*sem*, 0, count);
- sem_wait(&sem);
- sem_post(&sem);

- pthread_mutex_t mutex;
- pthread_mutex_lock(&mutex);
- pthread_mutex_unlock(&mutex);

- pthread_cond_t wait_for_cons;
- pthread_cond_wait(&wait_for_cons, &mutex);
- pthread_cond_signal(&wait_for_prod);