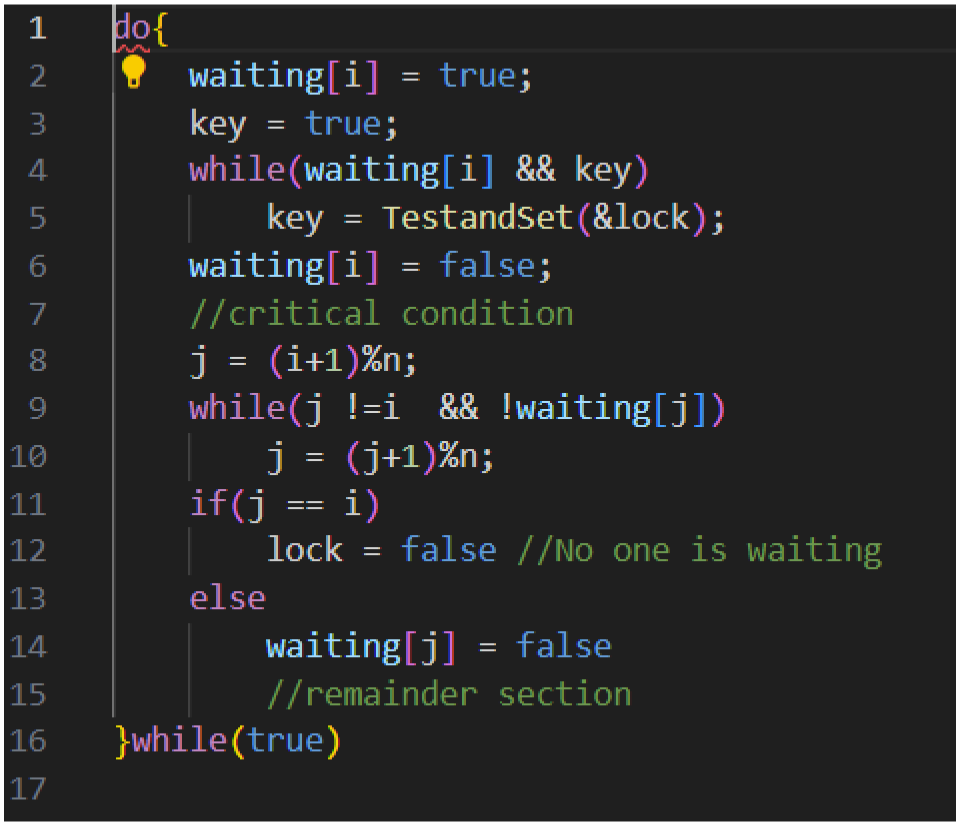
Course : Operating Systems **Instructor:** 黃能富

Student name: Tuguldur Tserenbaljir **Assignment:** Assignment 3

Student ID: 109006271

Bounded-waiting Mutual Exclusion with TestandSet()



```
do{
    waiting[i] = true;
    key = true;
    while(waiting[i] && key)
        key = TestandSet(&lock);
    waiting[i] = false;
    //critical condition
    j = (i+1)%n;
    while(j !=i  && !waiting[j])
        j = (j+1)%n;
    if(j == i)
        lock = false //No one is waiting
    else
        waiting[j] = false
        //remainder section
}while(true)
```

To prove that the algorithm works for ”Bounded-waiting Mutual Exclusion with TestandSet()”, we must see if it satisfies the three properties:

1) Mutual Exception: See if there is only one critical process one at a time.

2) Progress: If no process is in the critical section and there are some processes that wish to enter the critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3) Bounded-Waiting: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Course : Operating Systems **Instructor:** 黃能富

Student name: Tuguldur Tserenbaljir **Assignment:** Assignment 3

Student ID: 109006271

<u>Proving (1):</u>

The key variable is initially set to true, and each process will try to enter the critical section by setting **waiting[i]** to **true**. The **TestAndSet** function ensures that only one process can set the lock variable to **true** and enter the critical section. When a process successfully sets the lock, key is set to **false**, and it enters the **critical section**. Since lock can only be true for one process at a time, no other process can enter the critical section, ensuring mutual exclusion.

<u>Proving (2):</u>

Once a process is in the critical section, it sets **waiting[i]** to **false**. When it exits the critical section, it checks if there is any other process waiting to enter (in the loop **while (j != i && !waiting[j])**). If no process is waiting (**if (j == i)**), it releases the lock (**lock = false**). This ensures that a process in the remainder section cannot indefinitely postpone the processes waiting to enter the critical section, as the lock will be released when it detects that no process is waiting, and another process can acquire the lock.

<u>Proving (3):</u>

After a process has finished its critical section, it cycles through the waiting array to find the next process (j) that is waiting. It sets **waiting[j] = false** for the next waiting process, which will enable that process to enter the critical section on the next evaluation of the **while(waiting[i] && key)** loop. Since processes are checked in a cyclic order starting from **(i+1)%n**, each process will get a chance to acquire the lock. This means there is a bounded number of times that other processes can enter their critical sections between a process signaling its intention to enter the critical section and being allowed to enter, thus ensuring bounded waiting.

In conclusion, the algorithm satisfies the 3 properties. Therefore, the algorithm is proven.