① Hardware solution to Synchronization problem

⇒

Hardware based solution for critical section problem

However, software based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Instead, we can generally state that any solution to the critical section problem requires a simple tool - a lock. Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section. This is illustrated in the following code.

```
do {
    [acquire lock]
        critical section
    [release lock]
        remainder section
} while (TRUE);
```

Code: Solution to the critical section problem using locks

Hardware instructions for solving critical section problem.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically - that is, as one uninterruptible unit. We can use these special instructions to solve the critical section problem in a relatively simple manner. Rather than discussing one spefic instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the TestAndSet() and swap() instructions.

The TestAndSet() instruction can be defined as in the following code. The important characteristic of this instruction is that it is executed atomically. Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet() instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialised to false. The structure of process P_i is shown in figure below.

```
boolean TestAndSet (boolean * target) {
        boolean rv = * target ;
        * target = TRUE ;
        return rv;
}
```

Code : The definition of the TestAndSet() instruction

```
do {
    while (TestAndSet (& lock));
          // do nothing
          // critical section
     lock = FALSE ;
          // remainder section
} while (TRUE);
```

Code : Mutual-exclusion implementation with TestAndSet().

The swap() instruction, in contrast to the TestAndSet() instruction, operates on the contents of two words; it is defined as the following code. Like the TestAndSet() instruction, it is executed atomically. If the machine supports the swap() instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process P_i is shown in the following codes.

```
void swap (boolean * a , boolean * b) {
    boolean temp = * a;
      * a = * b;
      * b = temp ;
}
```

Code: The definition of the swap() instruction.

```
do {
   key = TRUE ;
   while ( key == TRUE)
       swap ( & lock , & key );

       // critical section

     lock = FALSE ;

       // remainder section .

} while (TRUE);
```

Code : Mutual exclusion implementation with the swap () instruction.

② The Dining - Philosophers Problem.

⇒ Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. when a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

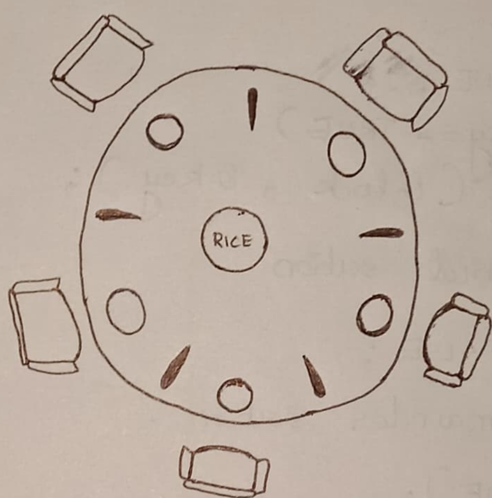

Fig: The situation of the dining philosophers.

The dining-philosophers problem is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore, she releases her chopsticks by executing the signal () operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick [5];

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown in the code below:

```
do {
    wait (chopstick [i]);
    wait (chopstick [(i+1) % 5]);
    ...
    // eat
    ...
    signal (chopstick [i]);
    signal (chopstick [(i+1) % 5]);
    ...
    // think
    ...
} while (TRUE);
```

code : The structure of philosopher i

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed below.

→ Allow at most four philosophers to be sitting simultaneously at the table.

→ Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).

→ Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.