# Classic Problems of Synchronization

So far, we have discussed three approaches to solve synchronization problems. To verify and test a proposed synchronization scheme, several classical synchronization problems are used. We will describe a few of these problems (producer-consumer, reader-writer, and dining philosophers) and how semaphores are used to resolve issues of mutual exclusion and synchronization in concurrent processing environments.

## Dining-Philosophers Problem

The dining-philosophers problem is considered a classic synchronization problem because it represents a large class of concurrency-control problems. When resources are shared among a group of processes, with a specific order among processes and resources as follows: 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 as shown in Figure 1. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, he does not interact with his colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him (the chopsticks that are between his and his left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, he cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing the chopsticks. When he is finished eating, he puts down both chopsticks and starts thinking again.

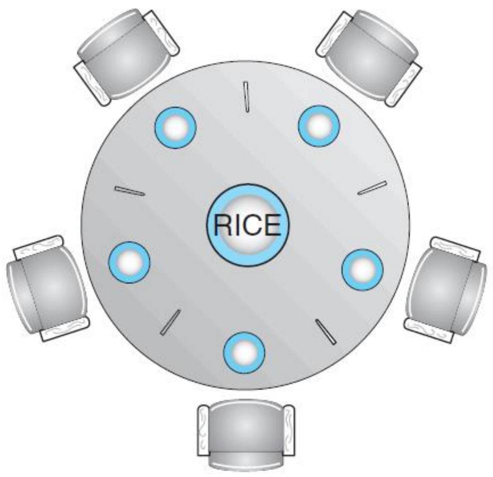

Figure 1: Dining Philosophers

It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
The 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. He releases his chopsticks by executing the signal() operation on the appropriate semaphores. We can use an array of shared semaphore chopstick[5], and initialize values of all elements to ‘1’.

The code of the philosopher process is listed below:

```
Semaphore chopstick[0..4]; int i;
Philosopher P(i)
while (true) {
          . . .
        /* think  */
         . . .
        wait(chopstick[i]);     /* pick left chopstick   */
        wait(chopstick[(i+1) % 5]); /* pick right chopstick   */
           . . .
        /* eat for a while */
         . . .
        signal(chopstick[i]);               /* release left chopstick   */
        signal(chopstick[(i+1) % 5]);  /* pick right chopstick   */    . . .
}
```

Although this solution guarantees that no two neighbors are eating simultaneously (basic requirement implicitly stated in the problem) it is not feasible because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs his left chopstick. All the elements of chopstick will now be equal to '0'. When each philosopher tries to grab his right chopstick, he will be delayed forever.
Several possible remedies to the deadlock problem are discussed below:

Allow four philosophers to be sitting simultaneously at the table. If you look carefully, here we are reducing competing processes. By introducing a room semaphore, initialized to '4' in ensures that at any given time at most two philosophers will eat. The code of the dining philosophers process with four sitting around table is listed below:

```
Semaphore: room (= 4), chopstick[0..4]; int i;
Philosopher P(i)
while (true) {
          . . .
        /* think  */
         . . .
        wait(room);
        wait(chopstick[i]);     /* pick left chopstick   */
        wait(chopstick[(i+1) % 5]); /* pick right chopstick   */
           . . .
        /* eat for a while */
         . . .
        signal(chopstick[i]);               /* release left chopstick   */
        signal(chopstick[(i+1) % 5]);  /* pick right chopstick   */
        signal(room);
        . . .
}
```

Using an asymmetric solution: an odd-numbered philosopher picks up first his left chopstick and then his right chopstick, whereas an even numbered philosopher picks up his right chopstick and then his left chopstick. The code for the dining philosophers processes for odd and even number philosophers is listed below:

```
$P_i$ () /* odd */
 While(condition) {
   think( );
   wait(chopstic[i]);
   wait(chopstick[i + 1 mod 5]);
    Eat();
      ...
    signal(chopstic[i]);
    signal(chopstick[i + 1 mod 5]);
}
```

```
$P_i$ () /* even */
 While(condition) {
   think( );
   wait(chopstick[i + 1 mod 5]);
   wait(chopstic[i]);
    Eat();
      ...
    signal(chopstick[i + 1 mod 5]);
    signal(chopstick[i]);
}
```

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