TOPIC : Classical problems of Synchronization

PRESENTATION BY

Mrs.D.Beulah
Assistant Professor
Aditya Degree College, Kkd.

Solutions to Classical problems of Synchronization

1. The Bounded-Buffer Problem

This is a generalization of the producer-consumer problem wherein access is controlled to a shared group of buffers of a limited size.

In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively (and initialized to 0 and N respectively.)

The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.

```
do {
    . . .
    // produce an item in nextp
    . . .
    wait(empty);
    wait(mutex);
    . . .
    // add nextp to buffer
    . . .
    signal(mutex);
    signal(full);
}while (TRUE);
```

Figure 5.9 The structure of the producer process.

```
do {
    wait(full);
    wait(mutex);
    . . .
    // remove an item from buffer to nextc
    . . .
    signal(mutex);
    signal(empty);
    . . .
    // consume the item in nextc
    . . .
}while (TRUE);
```

Figure 5.10 The structure of the consumer process.

2. The Readers-Writers Problem

- Readers - who only read the shared data, and never change it.
- Writers - who may change the data in addition to or instead of reading it.

There is no limit to how many readers can access the data simultaneously,

but when a writer accesses the data, it needs exclusive access.

Variations to the readers-writers problem,

- 1. readers-writers problem which gives priority to readers
- 2. readers-writers problem which. gives priority to the writers.

The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:

readcount : It is used by the reader processes, to count the number of readers currently accessing the data.

Mutex : It is a semaphore used only by the readers for controlled access to readcount.

rw_mutex : is a semaphore used to block and release the writers.

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

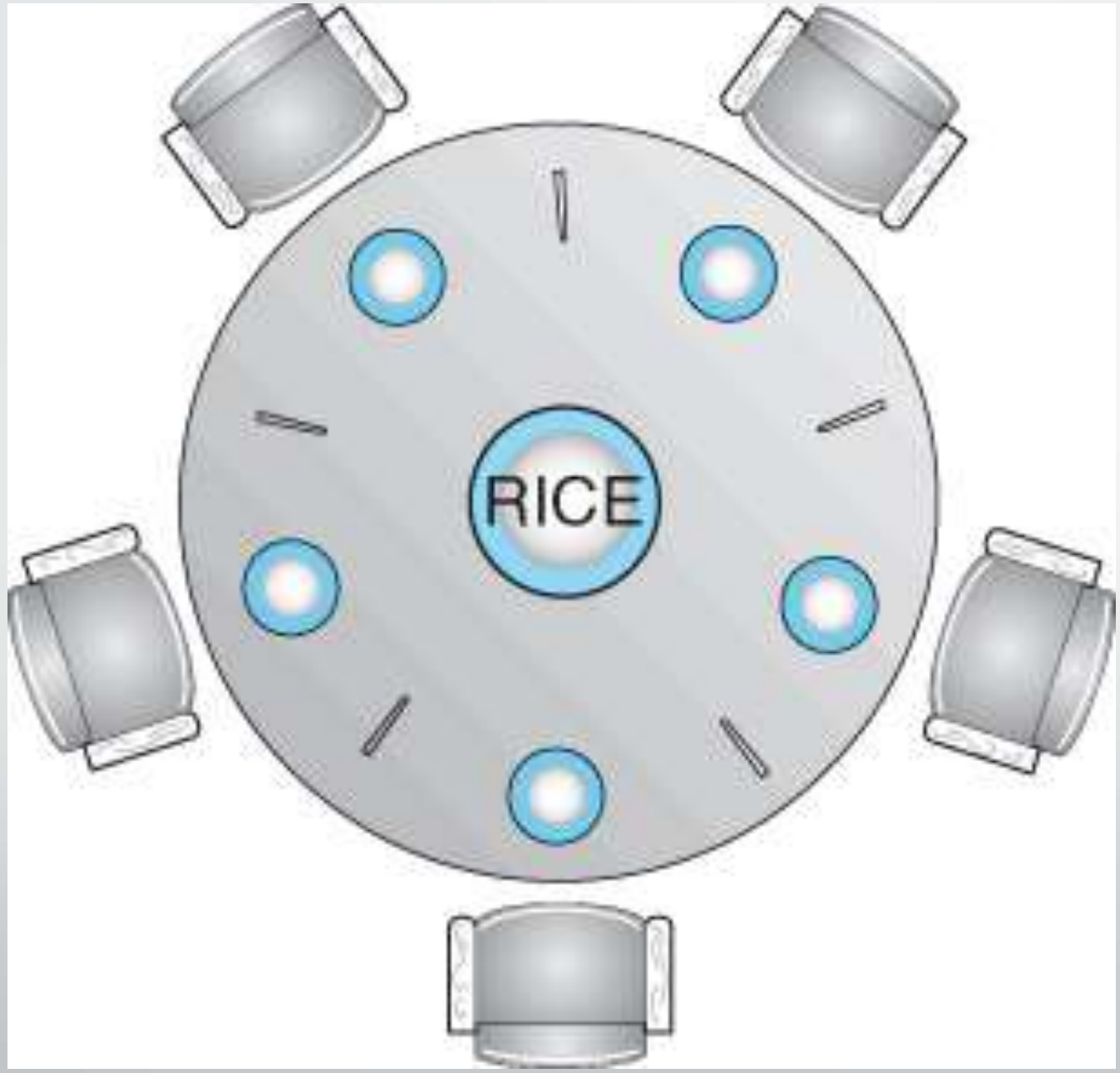
Figure 5.12 The structure of a reader process.

3. The Dining-Philosophers Problem

The problem involves the allocation of limited resources among a group of processes in a deadlock-free and starvation-free manner:

- Consider five philosophers sitting around a table, in which there are five chopsticks evenly distributed and an endless bowl of rice in the center. (There is exactly one chopstick between each pair of dining philosophers.)
- These philosophers spend their lives alternating between two activities: eating and thinking.
- When it is time for a philosopher to eat, it must first acquire two chopsticks - one from their left and one from their right.

When a philosopher thinks, it puts down both chopsticks in their original locations.


RICE

- One possible solution, as shown in the following code section, is to use a set of five semaphores (chopsticks[5]), and to have each hungry philosopher first wait on their left chopstick (chopsticks[i]), and then wait on their right chopstick (chopsticks[(i + 1) % 5])

- But suppose that all five philosophers get hungry at the same time, and each starts by picking up their left chopstick. They then look for their right chopstick, but because it is unavailable, they wait for it, forever, and eventually all the philosophers starve due to the resulting deadlock.

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