Operating System

Unit – 3

(Part-C)

Process Synchronization

Mayank Mishra
School of Electronics Engineering
KIIT-Deemed to be University

Reader-Writer Problem

- The readers-writer problem in operating systems is about managing access to shared data. It allows multiple readers to read data at the same time without issues but ensures that only one writer can write at a time, and no one can read while writing is happening.

- This helps prevent data corruption and ensures smooth operation in multi-user systems. Probably the most fundamental problem in concurrent programming is to provide safe access to shared resources. A classic problem used to illustrate this issue is **Readers-Writers**.

- It is a significant problem for showing how data structures might be synchronized such that consistency is guaranteed and efficiency is ensured. The Readers-Writers problem refers specifically to situations where a number of processes or threads may possibly have access to some common resource, like a database or a file.

Reader-Writer Problem (Contd.)

- The Readers-Writers Problem is a classic synchronization issue in operating systems that involves managing access to shared data by multiple threads or processes. The problem addresses the scenario where:
  - **Readers**: Multiple readers can access the shared data simultaneously without causing any issues because they are only reading and not modifying the data.
  - **Writers**: Only one writer can access the shared data at a time to ensure data integrity, as writers modify the data, and concurrent modifications could lead to data corruption or inconsistencies.

Reader-Writer Problem (Contd.)

The challenge now becomes how to create a synchronization scheme such that the following is supported:

- **Multiple Readers**: A number of readers may access simultaneously if no writer is presently writing.
- **Exclusion for Writers**: If one writer is writing, no other reader or writer may access the common resource.

Reader-Writer Problem (Contd.)

There are two fundamental solutions to the Readers-Writers problem:

- ❑ **Readers Preference:** In this solution, readers are given preference over writers. That means that till readers are reading, writers will have to wait. The Writers can access the resource only when no reader is accessing it.

- ❑ **Writer's Preference:** Preference is given to the writers. It simply means that, after arrival, the writers can go ahead with their operations; though perhaps there are readers currently accessing the resource.

Reader-Writer Problem (Contd.)

There are two fundamental solutions to the Readers-Writers problem:

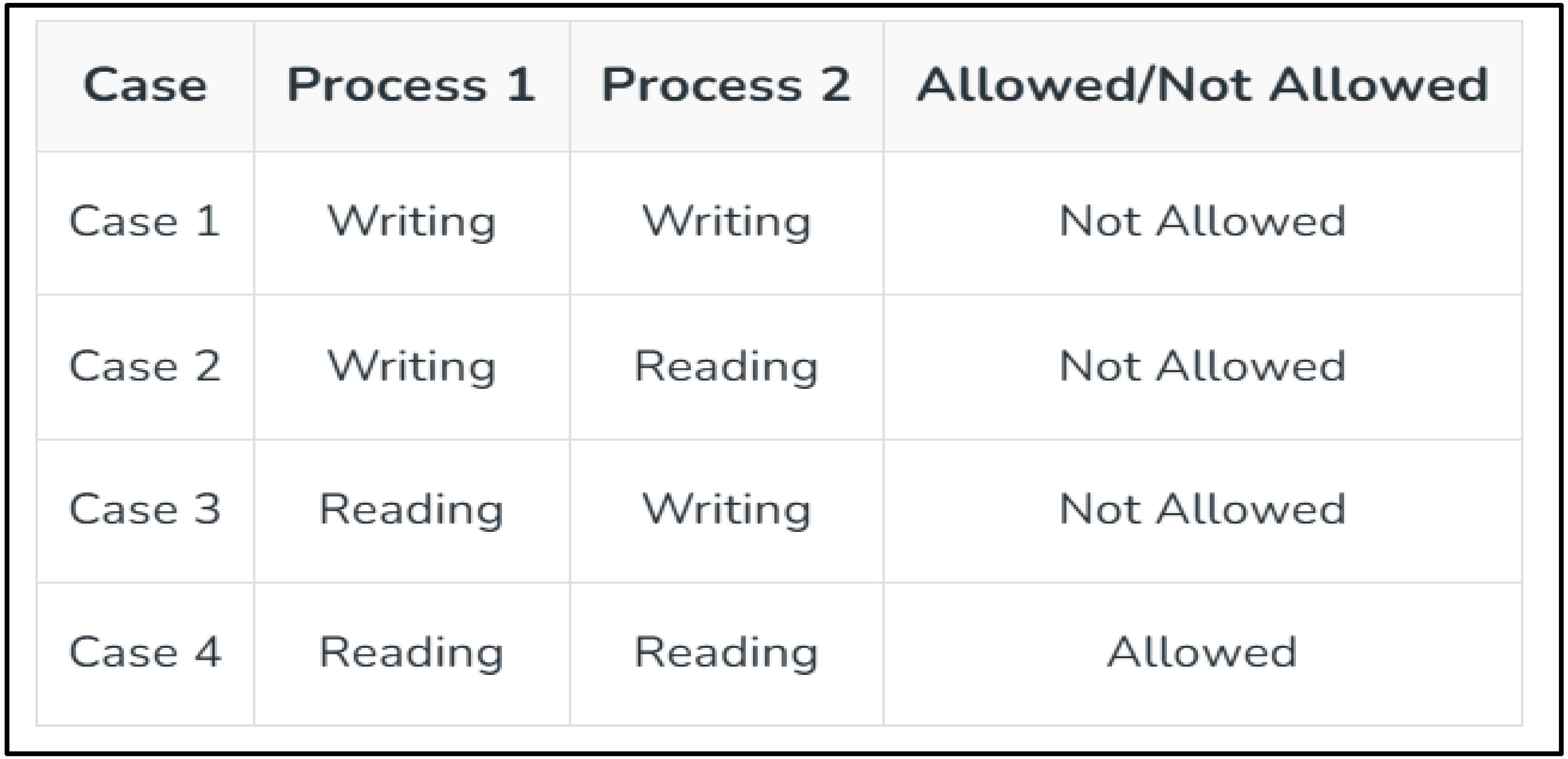
- ❑ **Readers Preference:** In this solution, readers are given preference over writers. That means that till readers are reading, writers will have to wait. The Writers can access the resource only when no reader is accessing it.

- ❑ **Writer's Preference:** Preference is given to the writers. It simply means that, after arrival, the writers can go ahead with their operations; though perhaps there are readers currently accessing the resource.

✓ ***We will be covering the "Reader's Preference".***

Reader-Writer Problem (Contd.)

- Here priority means, no reader should wait if the share is currently open for reading. There are four types of cases that could happen here.



Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Reader-Writer Problem (Contd.)

Three variables are used: **mutex, D/B, rc** to implement a solution.

- **semaphore** mutex, D/B; // semaphore **mutex** is used to ensure mutual exclusion when **rc** is updated i.e. when any reader enters or exits from the critical section, and semaphore **D/B** used by both readers and writers (As mutex and D/B is binary semaphore, it is initialized to 1).

- **int** rc; //**rc** tells the number of processes performing read in the critical section, initially 0.

```
int rc=0
semaphore mutex=1;
semaphore D/B=1;
```

```
void Reader(void)
  {
          while(true)
      {
          down(mutex);
          rc=rc+1;
          if(rc==1) then down(D/B);
          up(mutex);
         Critical Section/ DB
          down(mutex)
          rc=rc-1;
          if(rc==0) then up(D/B);
          up(mutex)
          process data
       }
  }
```

Reader

```
void write(void)
  {
        while(true)
      {
         down(D/B);
       Critical Section/ DB
         up(D/B);
      }
}
```

Writer

- **DB**: Database
- A **database** is an electronically stored, systematic collection of data. It can contain any type of data, including words, numbers, images, videos, and files.

Dining Philosophers Problem

- The Dining Philosophers Problem is a classic synchronization and concurrency problem in computer science and is used to illustrate issues related to resource sharing and process synchronization. It was formulated by **Edsger Dijkstra** in 1965.
- The Dining Philosopher Problem involves ‘n’ philosophers sitting around a circular table.
- Each philosopher alternates between two states: **thinking** and **eating**.

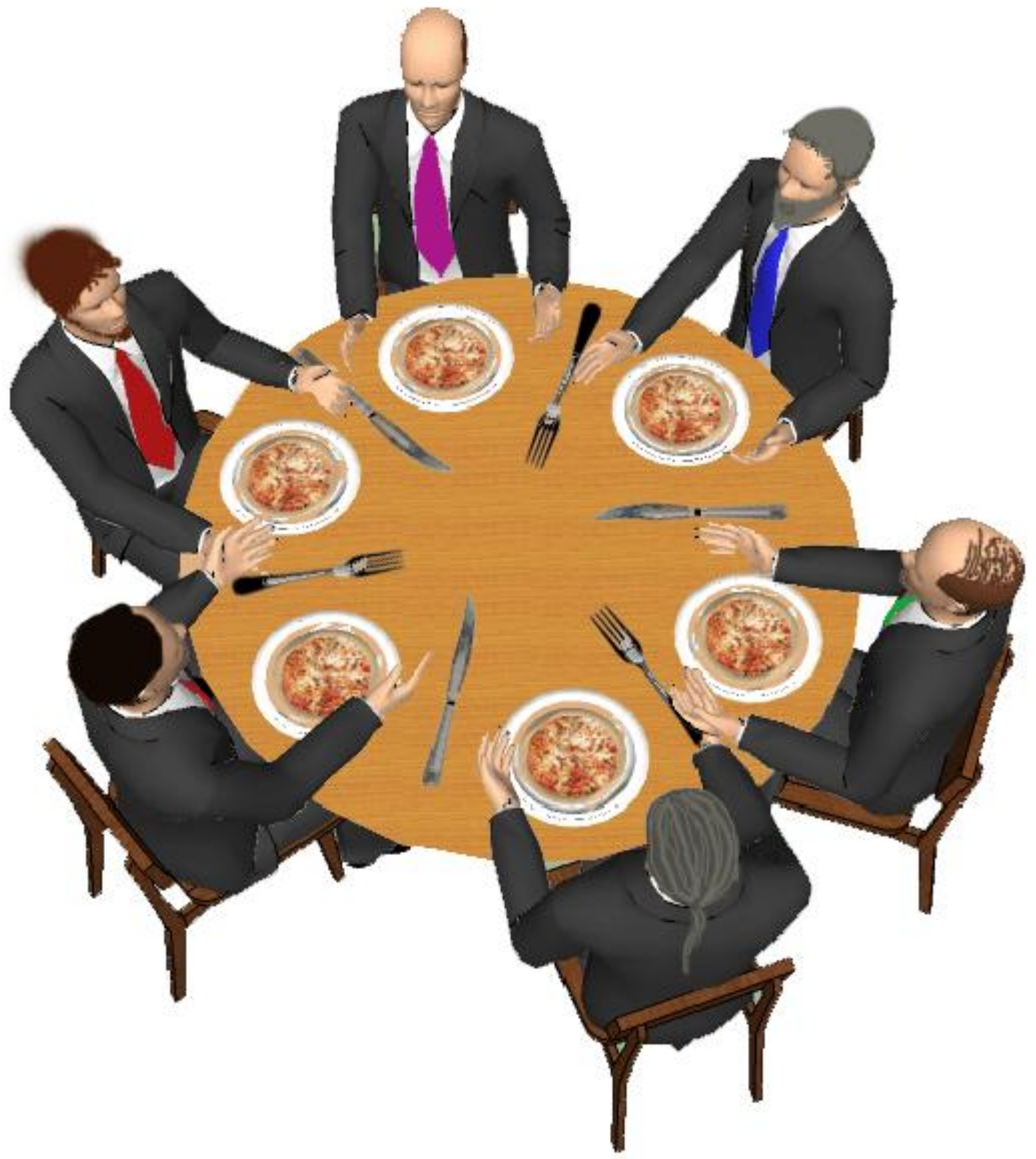


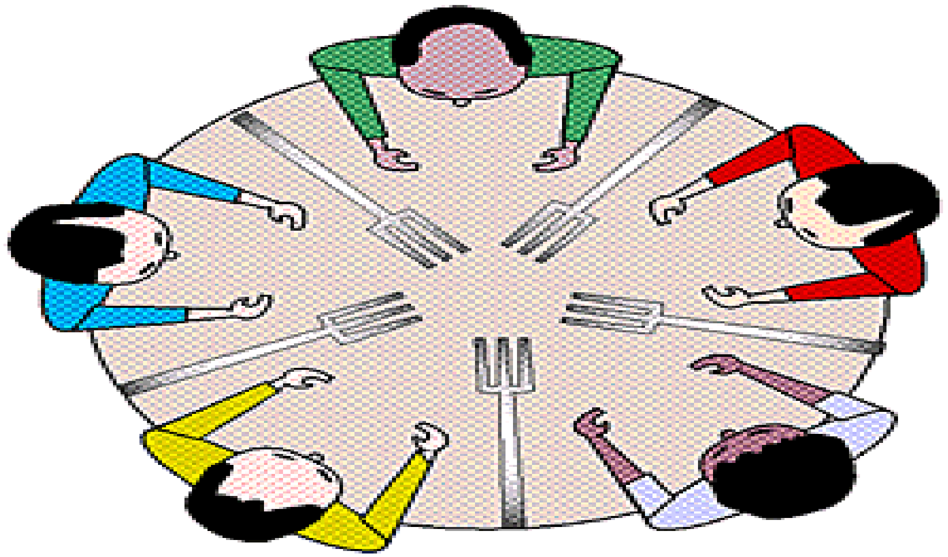
Fig Source: https://www.researchgate.net/profile/Stefano_Schivo/publication/43378479/figure/fig4/AS:669467633791033@1536624930717/An-example-configuration-of-the-dining-philosophers-problem-showing-how-6-philosophers.png

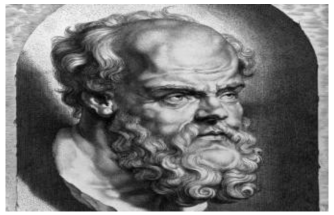
Dining Philosophers Problem

- To eat, a philosopher needs two forks, one on their left and one on their right.

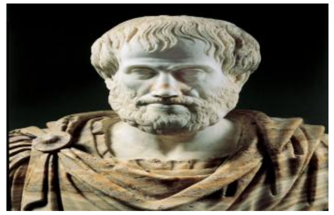
- However, the number of fork is equal to the number of philosophers, and each fork is shared between two neighboring philosophers.

- The standard problem considers the value of ‘n’ as 5 i.e. we deal with 5 Philosophers sitting around a circular table.

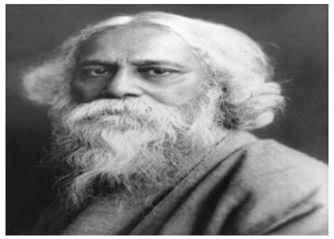




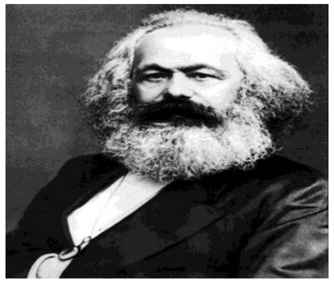
Socrates



Aristotle



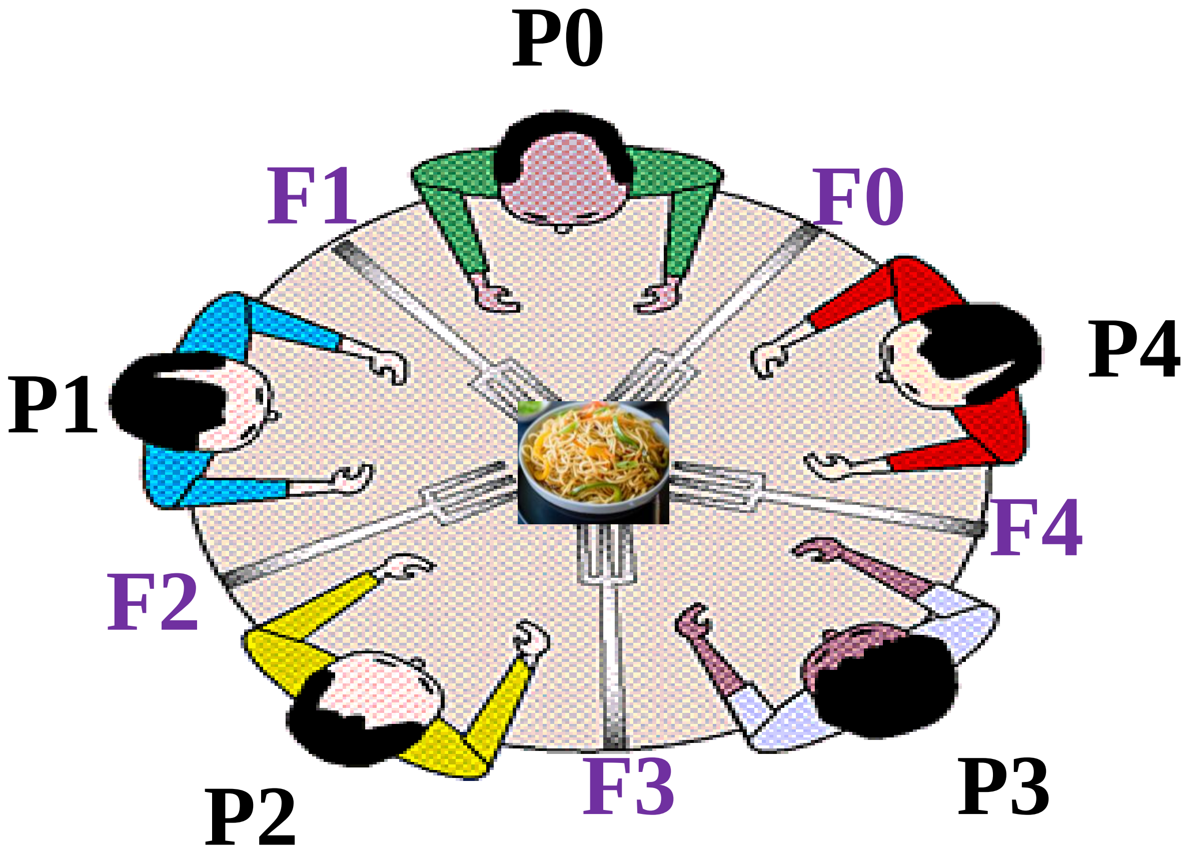
Rabindranath Tagore



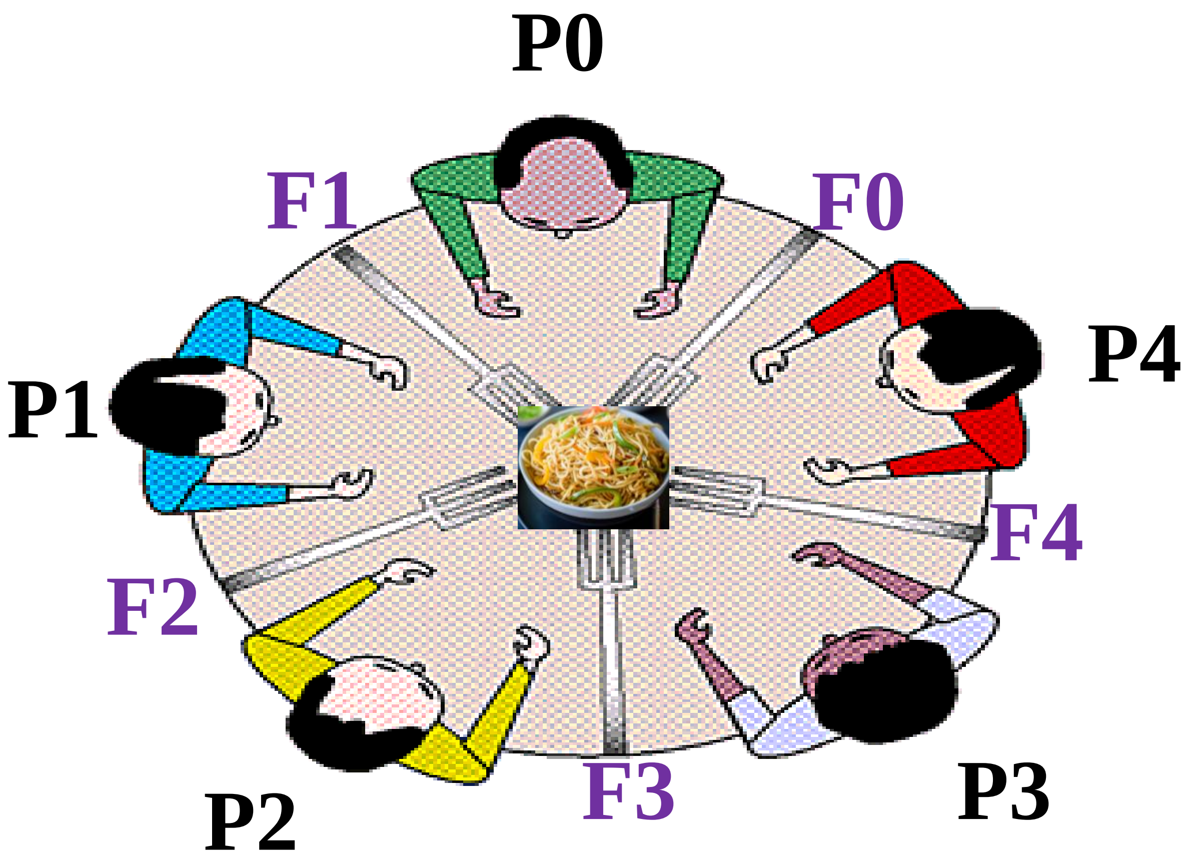
Karl Marx

Simone de Beauvoir

```
void Philosopher(void)
{
While(true)
{
Thinking();
Take_fork(i);
Take_fork( (i+1) mod N); // N- Number of  forks

EAT();

Put_fork(i);
Put_fork ( (i+1) mod N );
}
}
```

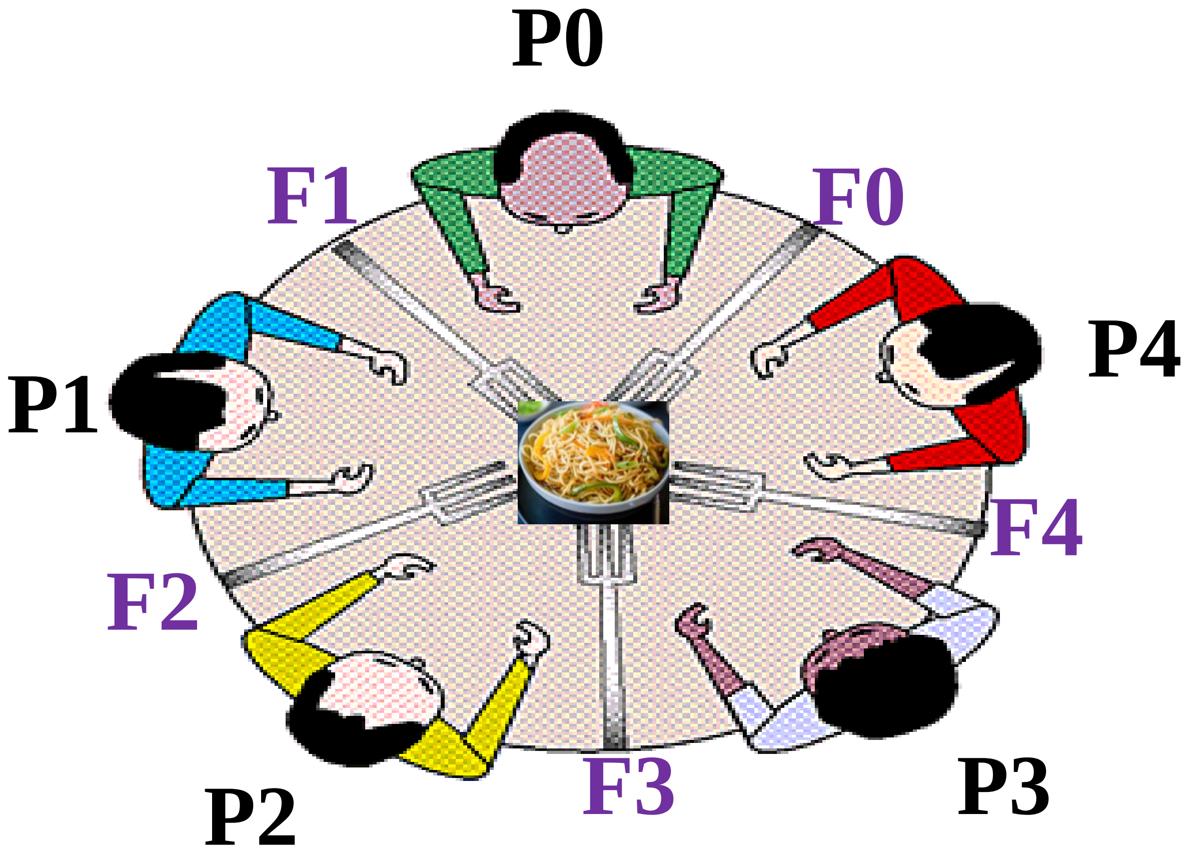


```
void Philosopher(void)
{
While(true)
{
Thinking();
Take_fork(i);
Take_fork( (i+1) mod N); // N- Number of  forks

EAT();

Put_fork(i);
Put_fork ( (i+1) mod N );
}
}
```



Note:
*If philosophers are coming sequentially, then no problem will be encountered. When two or more philosophers comes at the same time, then there may be problem of **Race Condition**. To overcome this condition, **binary semaphore** will be taken into consideration.*

```
void Philosopher(void)
{
While(true)
{
Thinking();

wait(Take_fork(S[i]);

Wait(Take_fork( S[(i+1) mod N]);

EAT();

Signal(Put_fork(S[i]);

Put_fork ( S[(i+1) mod N] );
}
}
```



- *Array of Semaphore will be used: S[i], i-indexing of fork. Here, i = 0 to 4.*

```
void Philosopher(void)
{
While(true)
{
Thinking();

wait(Take_fork(S[i]);

Wait(Take_fork( S[(i+1) mod N]);

EAT();

Signal(Put_fork(S[i]);

Put_fork ( S[(i+1) mod N] );
}
}
```

Requirements of Semaphores by each Philosopher:

P0	S0	S1
P1	S1	S2
P2	S2	S3
P3	S3	S4
P4	S4	S0

- *Array of Semaphore will be used: S[i], i-indexing of fork. Here, i = 0 to 4.*

Does this solution have any problem?

- What if P0 takes left fork and gets pre-empted before taking right fork ?

- What if same happens with all the four Philosophers?

P0	S0	S1
P1	S1	S2
P2	S2	S3
P3	S3	S4
P4	S4	S0

The situation of **Deadlock** will be occurred.

What should be the solution?

Solution

P0	S0	S1
P1	S1	S2
P2	S2	S3
P3	S3	S4
P4	S0	S4

Several possible remedies to the deadlock problem are the following:

- 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.

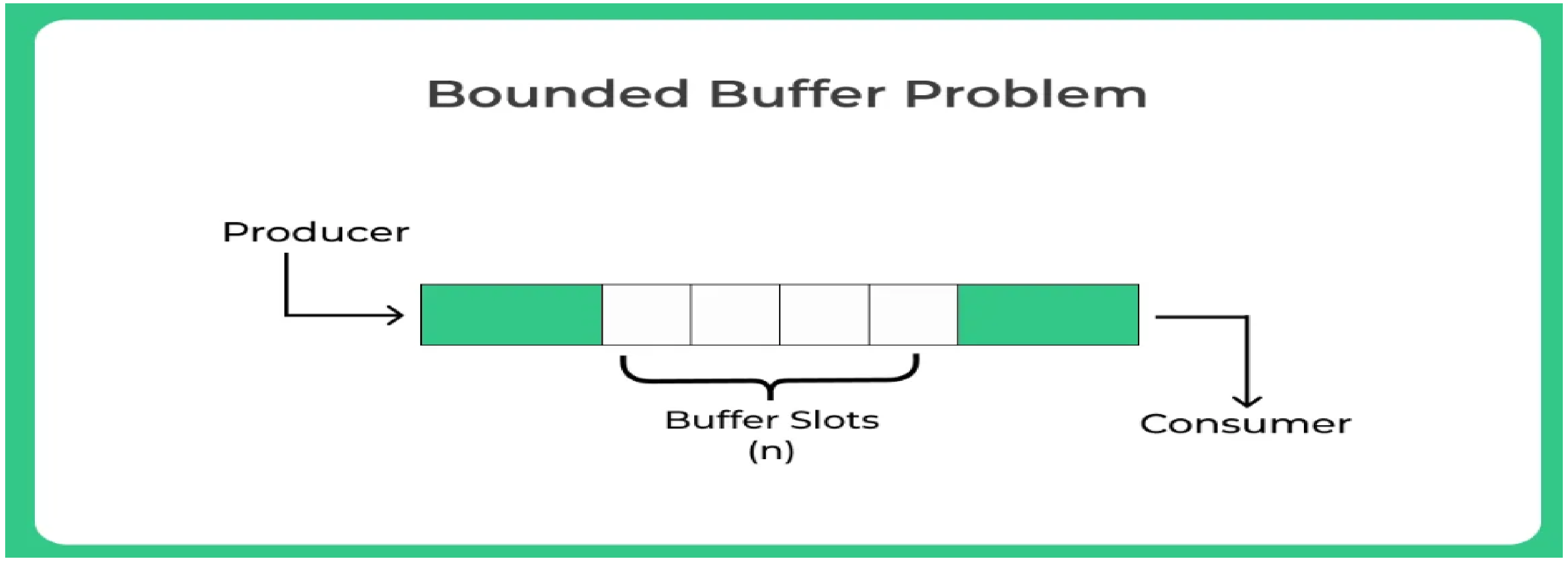
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

Bounded Buffer Problem/ Producer Consumer Problem (Solution using Semaphores)

- Bounded Buffer Problem (Producer Consumer Problem), is one of the classic problem of synchronization.

- There is a buffer of ‘n’ slots and each slot is capable of storing one unit of data.

- There are two processes running, namely, Producer and Consumer, which are operating on the buffer.



- ➢ The producer tries to insert data into an empty slot of the buffer.
- ➢ The consumer tries to remove data from a filled slot in the buffer.
- ➢ The producer must not insert data when the buffer is full.
- ➢ The consumer must not remove data when the buffer is empty.
- ➢ The producer and consumer should not insert and remove data simultaneously.

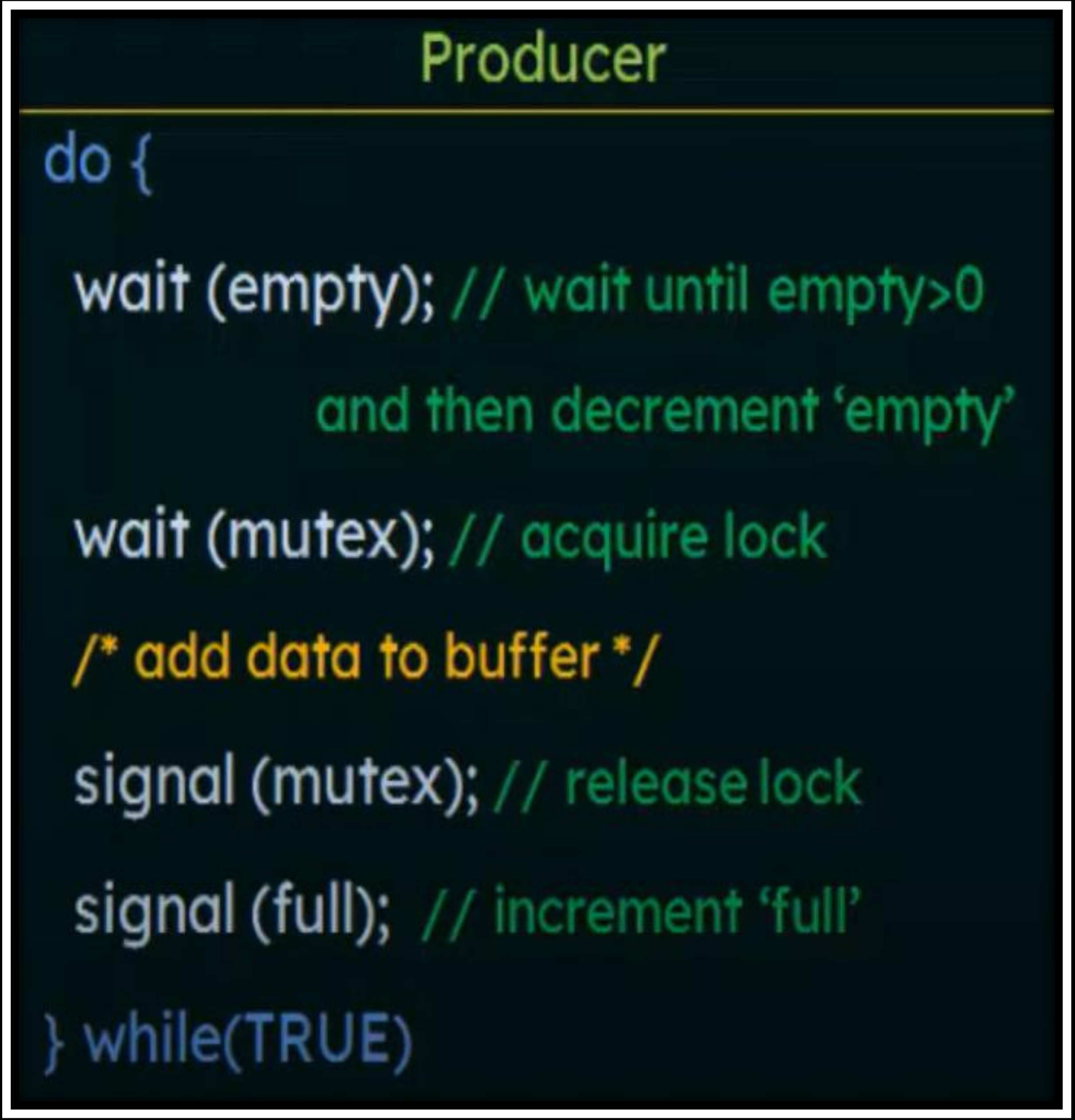
Solution to the Bounded Buffer Problem using Semaphores

Three semaphores will be utilized:

- **m (mutex),** a binary semaphore which is used to acquire and release the lock.

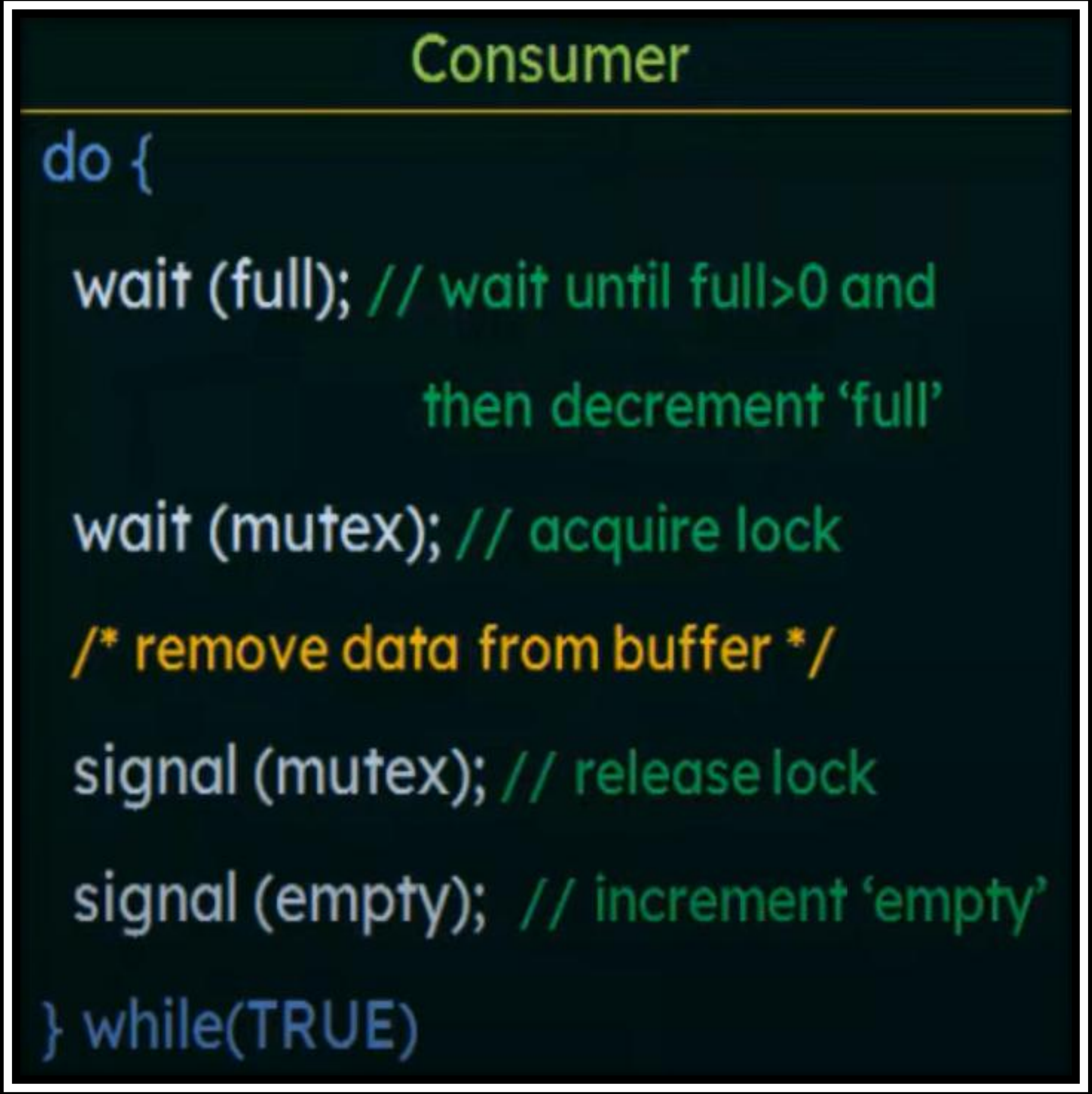
- **empty**, a counting semaphore whose initial value is the number of slots in the buffer, **initially all slots are empty**.

- **full**, a counting semaphore whose **initial value is 0**.



Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```



Consumer

```
do {
  wait (full); // wait until full>0 and
                  then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

References

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, “Operating System Concepts,” Eleventh Edition (Willey).

2. Andrew S. Tanenbaum, “Modern Operating Systems”, Fourth Edition (Pearson Publications), 2014.

3. https://www.geeksforgeeks.org/

4. https://www.javatpoint.com/

5. https://www.tutorialspoint.com/

6. https://www.nesoacademy.org/

7. https://www.baeldung.com/

8. https://www.educative.io/

9. https://prepinsta.com