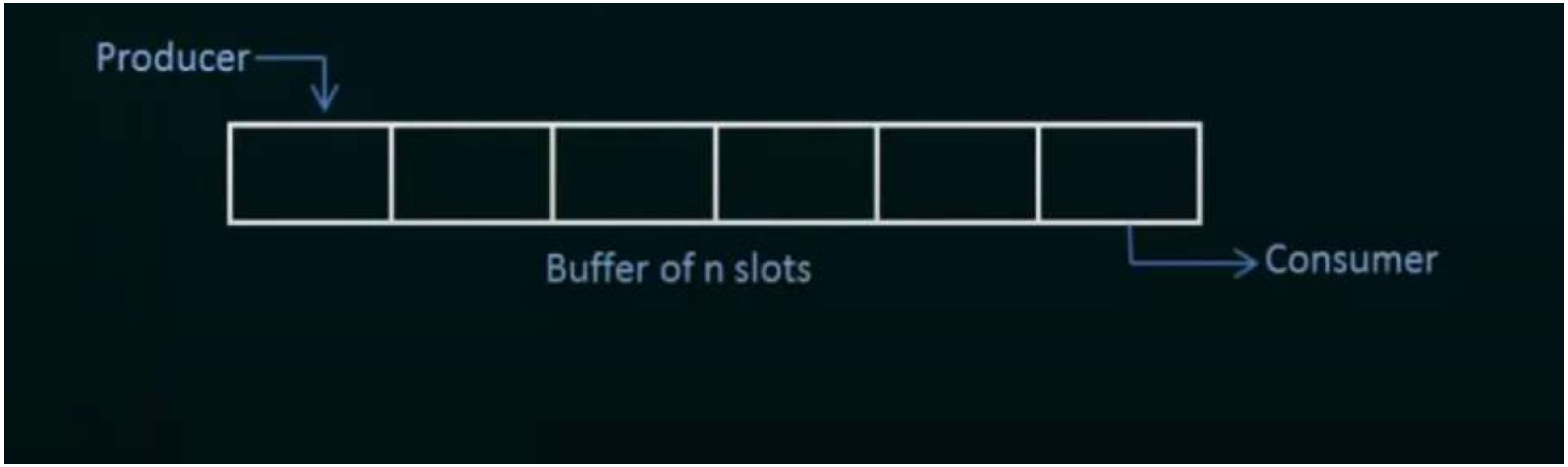
Classical Problems in Synchronization

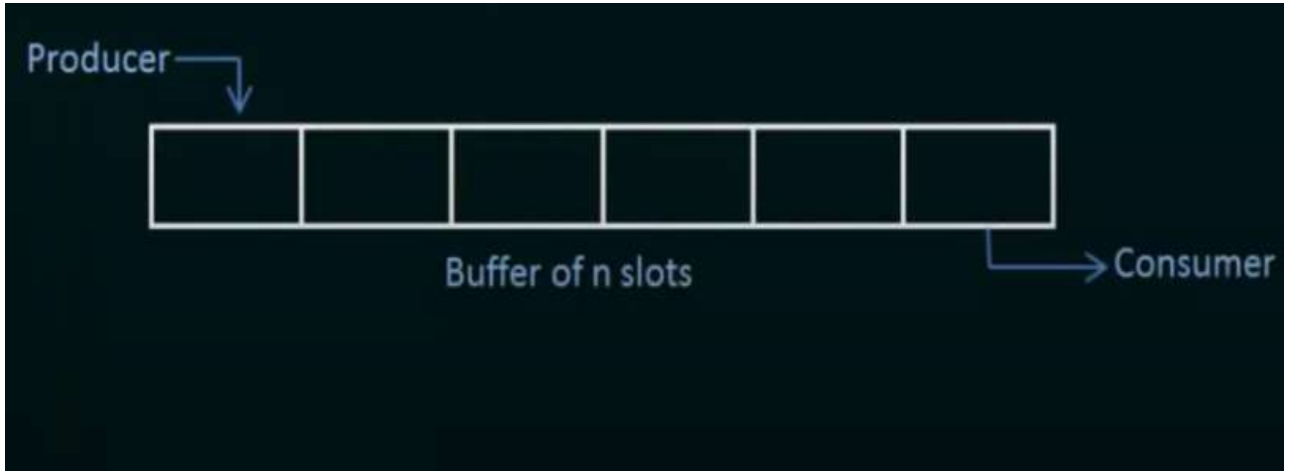
The bounded buffer problem

- **Producer Consumer Problem**
  - There is a buffer of n slots and each slot is capable of storing one unit of data.
  - There are two processes running
    - Producer & Consumer – operating on the buffer



The bounded buffer problem

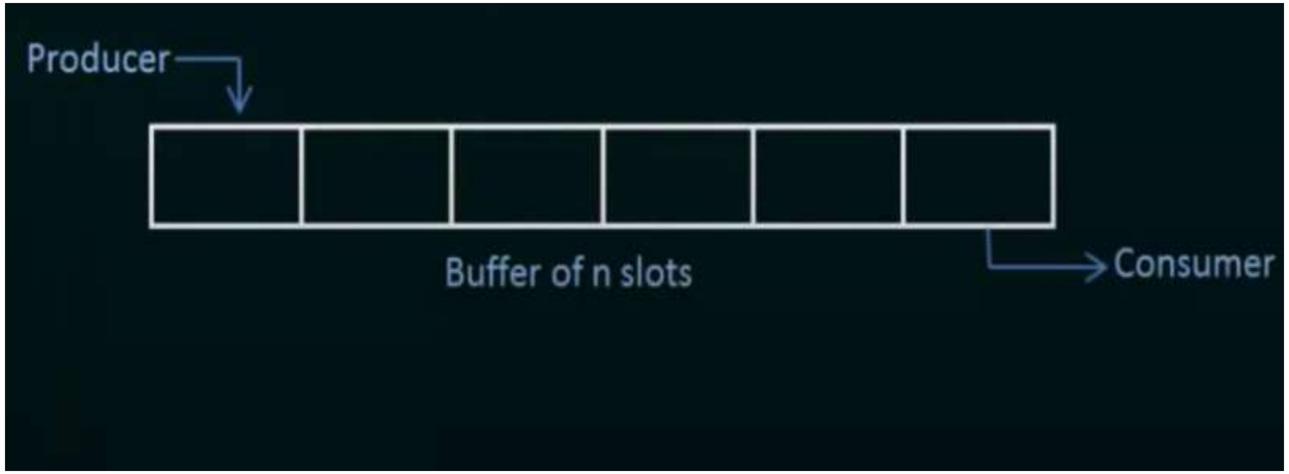
- ✓ A producer tries to insert data into an empty slot of the buffer.



- ✓ A consumer tries to remove data from a filled slot in the buffer.

The bounded buffer problem

- ✓ A producer should not insert data when the buffer is full
- ✓ Consumer should not remove data when the buffer is empty.
- ✓ The producer and consumer should not insert and remove data simultaneously.



The bounded buffer problem

- 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, since, initially all slots are empty.

- Full - a **counting semaphore** whose initial value is 0. Number of filled slots.

Producer & Consumer Operation

Producer

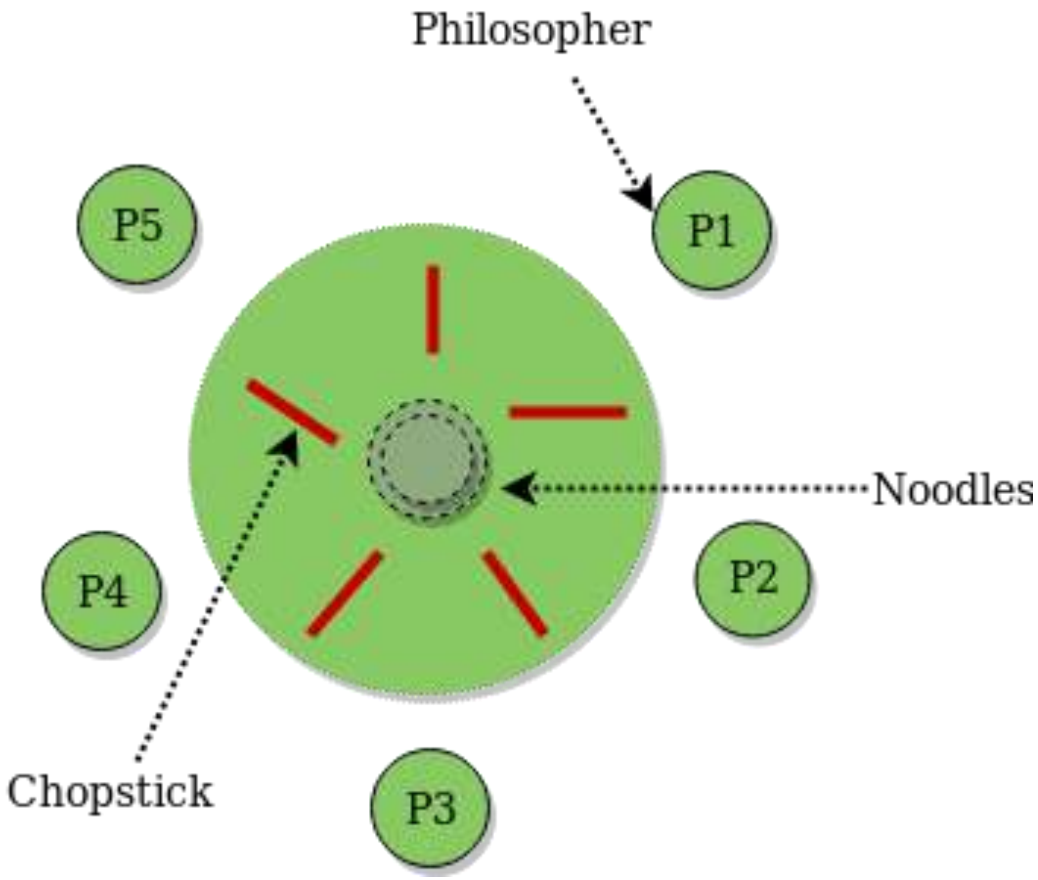
```
do {
  wait (empty); // wait until empty>0
               and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

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)
```

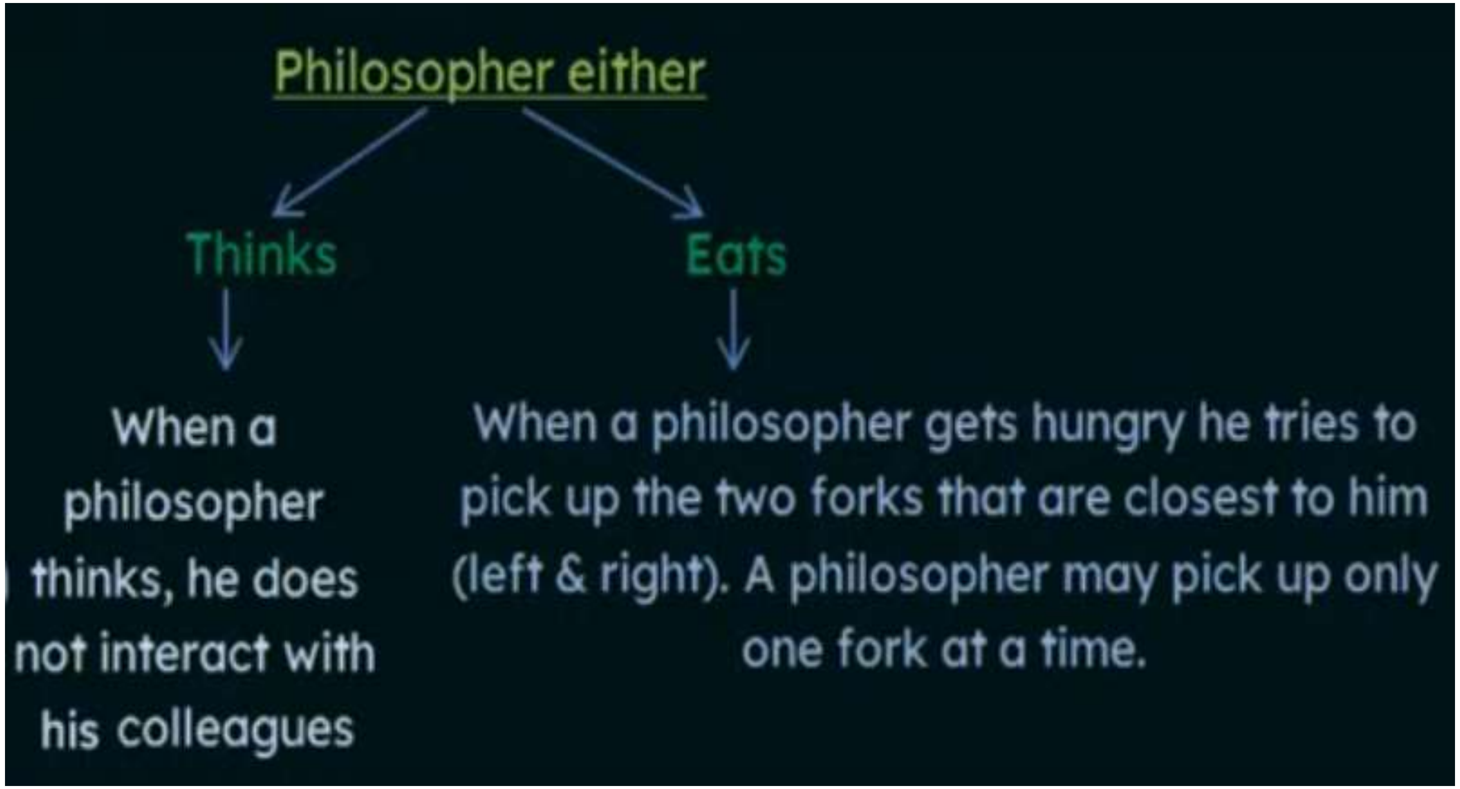
The Dining Philosophers Problem

- Five philosophers sit around a circular table and alternate between thinking and eating.
- A bowl of noodles and five forks for each philosopher are placed at the center of the table.



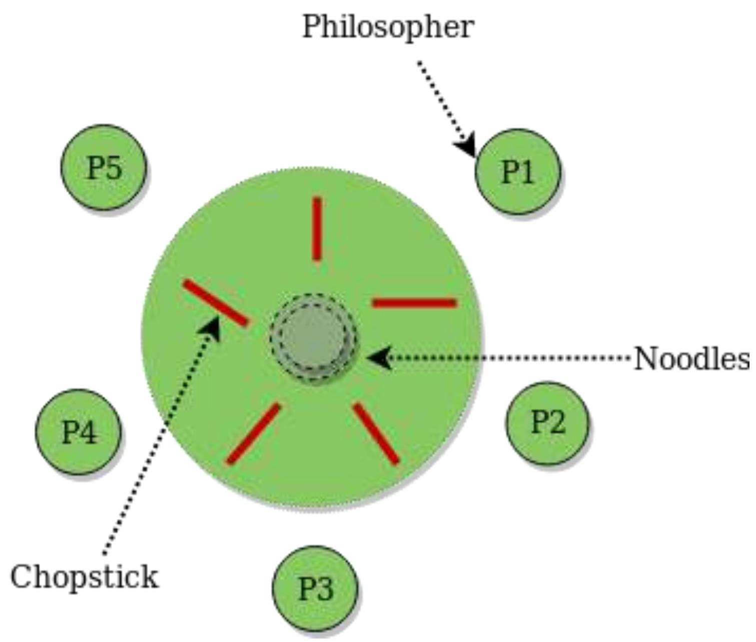
The Dining Philosophers Problem

- ✓ One cannot pick up a fork that is already in the hand of a neighbor.
- ✓ When a hungry philosopher has both his forks at same time, he eats without releasing his fork.
- ✓ When he has finished eating, he puts down both of his forks and starts thinking again.



The Dining Philosophers Problem

- A philosopher must use both their right and left forks to eat.
- A philosopher can only eat if both of his or her immediate left and right forks are available.
- If the philosopher's immediate left and right forks are not available, the philosopher places their (either left or right) forks on the table and resumes thinking.



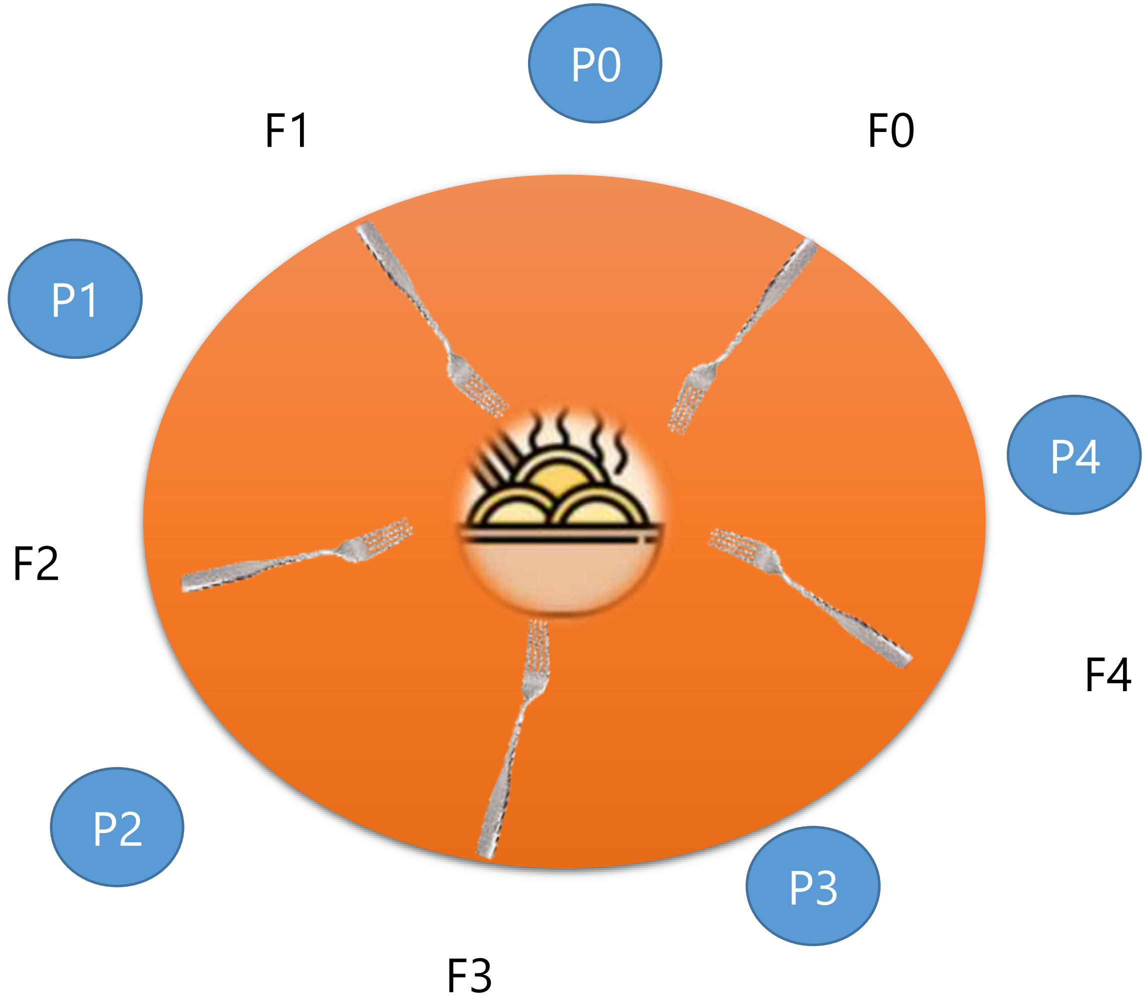
Dining Philosophers – Resource Allocation Problem

- ✓ Forks are limited
- ✓ No two philosophers are allowed to eat simultaneously.
- ✓ Problem of resource allocation in synchronized manner.
- ✓ Philosophers → Processes
- ✓ Fork → Resources required by the processes

Sharing Limited resources to processes

The Dining Philosophers Problem

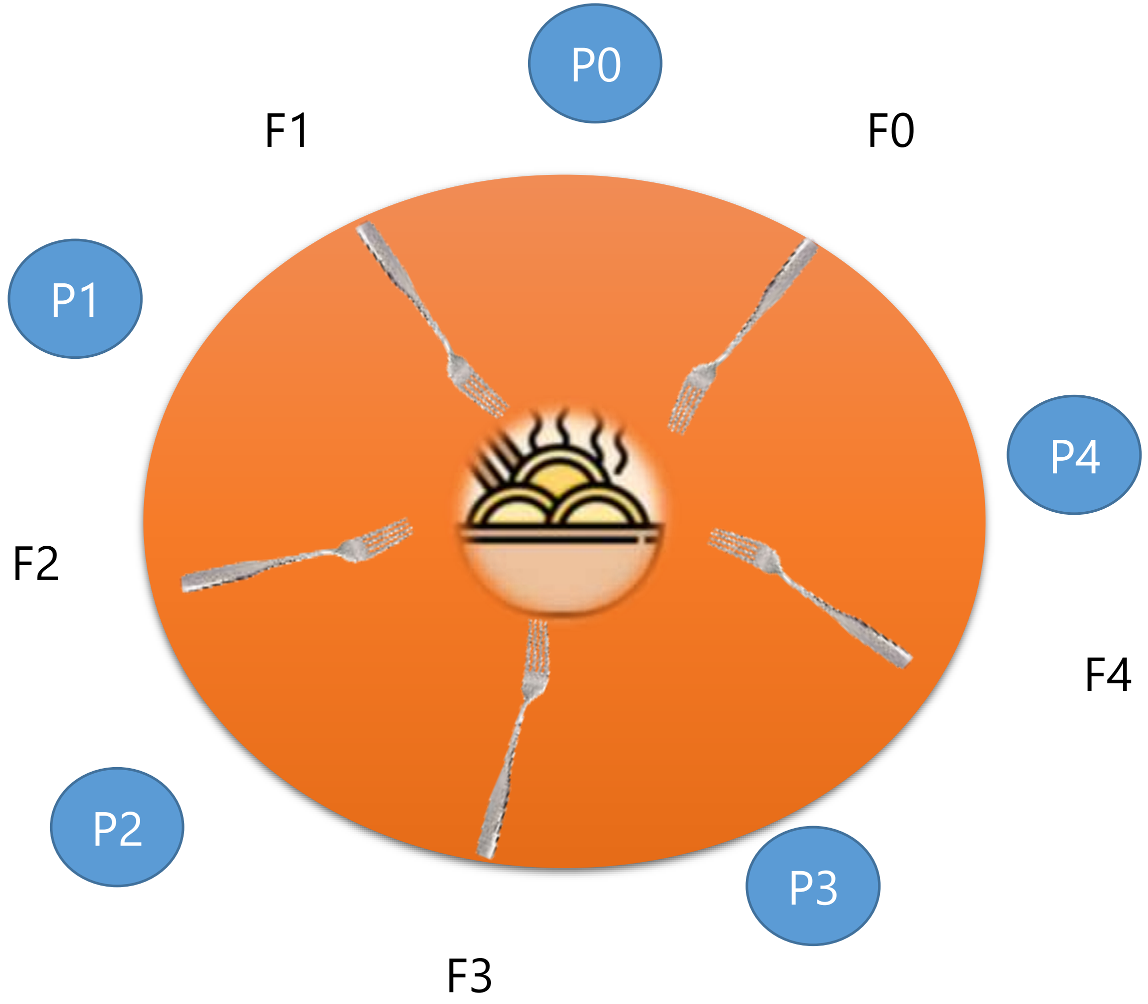
```
void Philosopher
{
 while(1)
  {
  take_fork[i];
  take_fork[ (i+1) % 5] ;

  EATING THE NOODLE

  put_fork[i];
  put_fork[ (i+1) % 5] ;

   THINKING
  }
}
```



The Dining Philosophers Problem

The structure of philosopher i

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



Readers Writers Problem

- Managing access to shared data by multiple threads or processes.
- Two processes
  - 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.

Readers Writers Problem

- **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.

Readers Writers Problem

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Process 1	Process 2	Allowed/Not Allowed
Writing	Writing	Not Allowed
Writing	Reading	Not Allowed
Reading	Writing	Not Allowed
Reading	Reading	Allowed

Readers Writers Problem

We will make use of two semaphores and an integer variable:

1. mutex, a semaphore (initialized to 1) which is used to ensure mutual exclusion when readcount is updated i.e. when any reader enters or exit from the critical section.
2. wrt, a semaphore (initialized to 1) common to both reader and writer processes.
3. readcount, an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object.

Readers Writers Problem

Writer Process

```
do {
/* writer requests for critical
section */
   wait(wrt);
  /* performs the write  */
   // leaves the critical section
   signal(wrt);
} while(true);
```

Reader Process

```
do {
  wait (mutex);
 readcnt++;  // The number of readers has now increased by 1
 if (readcnt==1)
    wait (wrt); // this ensure no writer can enter if there is even one reader
  signal (mutex);   // other readers can enter while this current reader is
                       inside the critical section
 /* current reader performs reading here */
  wait (mutex);
  readcnt--; // a reader wants to leave
  if (readcnt == 0)      //no reader is left in the critical section
     signal (wrt);       // writers can enter
     signal (mutex);  // reader leaves
} while(true);
```