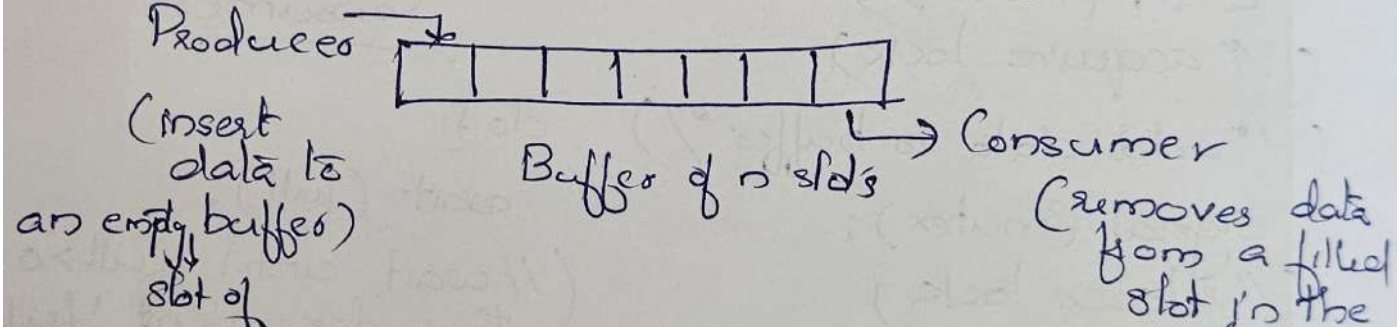
# Classic Problems of Synchronization

Used to determine how the various solutions that we find to synchronization problems perform in solving these problems.

## 1. Producer Consumer Pblm (Bounded-Buffer Problem)

- Buffer of n slots & each slot is capable of storing one unit of data.
- There are 2 processes running: Producer & Consumer, which are operating on buffer.




→ Producer must not insert data when buffer is full.

→ Consumer must not remove data when buffer is empty.

→ Both should not insert & remove data simultaneously.

### Solution

3 semaphores are used.

1. m (mutex), a binary semaphore which is used to acquire & release lock.

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

3. full, a counting semaphore whose initial value is 0:
   ↳ buffer is empty.

## Producer

```
do {                                   → will not produce since buffer is full
  wait (empty);                        if any empty slots only producer produces data.
(//wait until empty > 0
  & 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 &
  then decrement 'full')
  wait (mutex);
(// acquire lock)
(/* remove data from buffer
  signal (mutex);
(// release lock)
  signal (empty);
(// increment 'empty')
} while (TRUE)
```

↳ next one can consume.

## 2. Readers - Writers Problem

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (ie to read & write) the database.
- Readers & Writers.
- If 2 readers access the shared data simultaneously, no adverse effects will result.
- But, if a writer & some other thread (either a reader / writer) access the db simultaneously chaos may ensue.

∴ to ensure that these difficulties don't arise, we require that the writers have exclusive access to shared db.

Solution

2 semaphores & an integer variable:

1. mutex, a semaphore (initially 1) used to ensure mutual exclusion when readcount is updated. (ie. when any reader enters or exit from the critical section).
2. wrt, a semaphore (initally 1) common to both reader & writer processes.

3. readcount, integer variable (initially 0) that keeps track of how many processes are currently reading the ~~object~~ same data.

Writer

```
do{
/* Writer requests for
Critical Section */
wait (wrt);
/* performs write */
// leaves the critical section
signal (wrt);
}While (true);
```

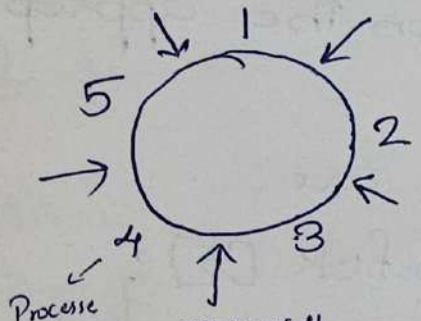
Reader

(Will not allow writer to write)

```
do {
  wait (mutex);
  readcnt++; // no. of readers has now increased by 1
  if (readcnt==1)  // atleast one reader is present
     wait (wrt); //ensures no writer can enter
                 if there is even one reader
  signal (mutex); // other reader can enter
                  ~~wrt~~ 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 critical
                      section.
    signal (wrt);  // writers can enter
    signal (mutex); // reader leaves.
} while (true);
```

3. Dining- Philosophers Problem




2 states :→ Thinking & Eating states

Philosopher either

Thinks ↓

When a philosopher thinks, he does not interact with his colleagues

Eats ↓

When a philosopher gets hungry, he tries to pick up the 2 forks that are closest to him (left & right).

A philosopher may pick up only one fork at a time. Also, one cannot pick up a fork that is already in the hand of a neighbour.

When a hungry philosopher has both his forks at the same time, he eats without releasing his forks. When he has finished eating, he puts down both of his forks & starts thinking again.

One simple solution is to represent each fork/ chopstick with a semaphore.

A philosopher tries to grab a fork / chopstick by executing a wait() operation on that semaphore.
↳ to get hold. so no other process can use.

He releases his fork / 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. {Using Binary Semaphore}.

Solution

Philosopher i

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

Although this solution guarantees that no 2 neighbours are eating simultaneously, it could still create a deadlock.

Suppose that all five philosophers become hungry simultaneously & each grabs their 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.

Possible remedies to avoid deadlocks:

→ Allow atmost 4 philosopher to be sitting simultaneously at the table.

→ 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)

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