Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- Bounded Buffer problem is also called producer consumer problem where a **finite** buffer pool is used to exchange messages between producer and consumer processes.
- The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
- The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.

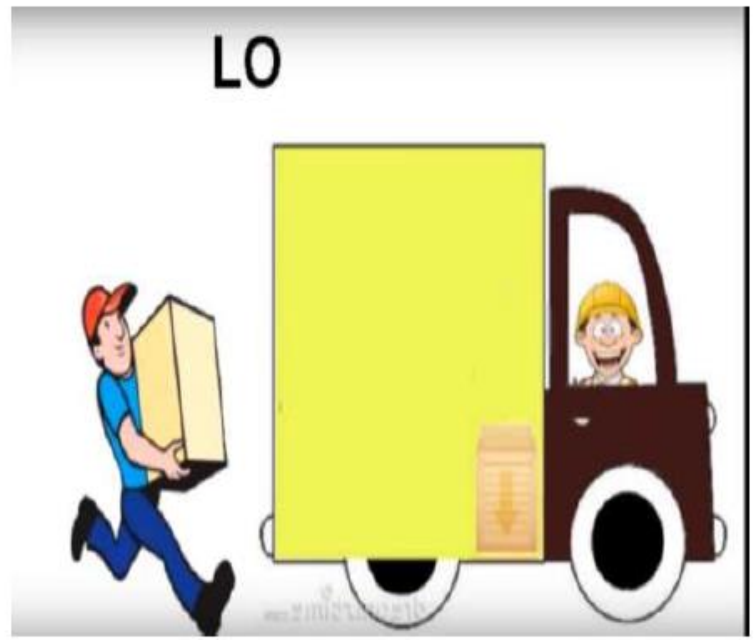
❑ Real life Example

Picture 1: Full of items

Picture 2: Lorry is empty

❑ Real life Example

Picture 3 : Working for load



Picture 4 : Loading

Picture 5 : Truck is full

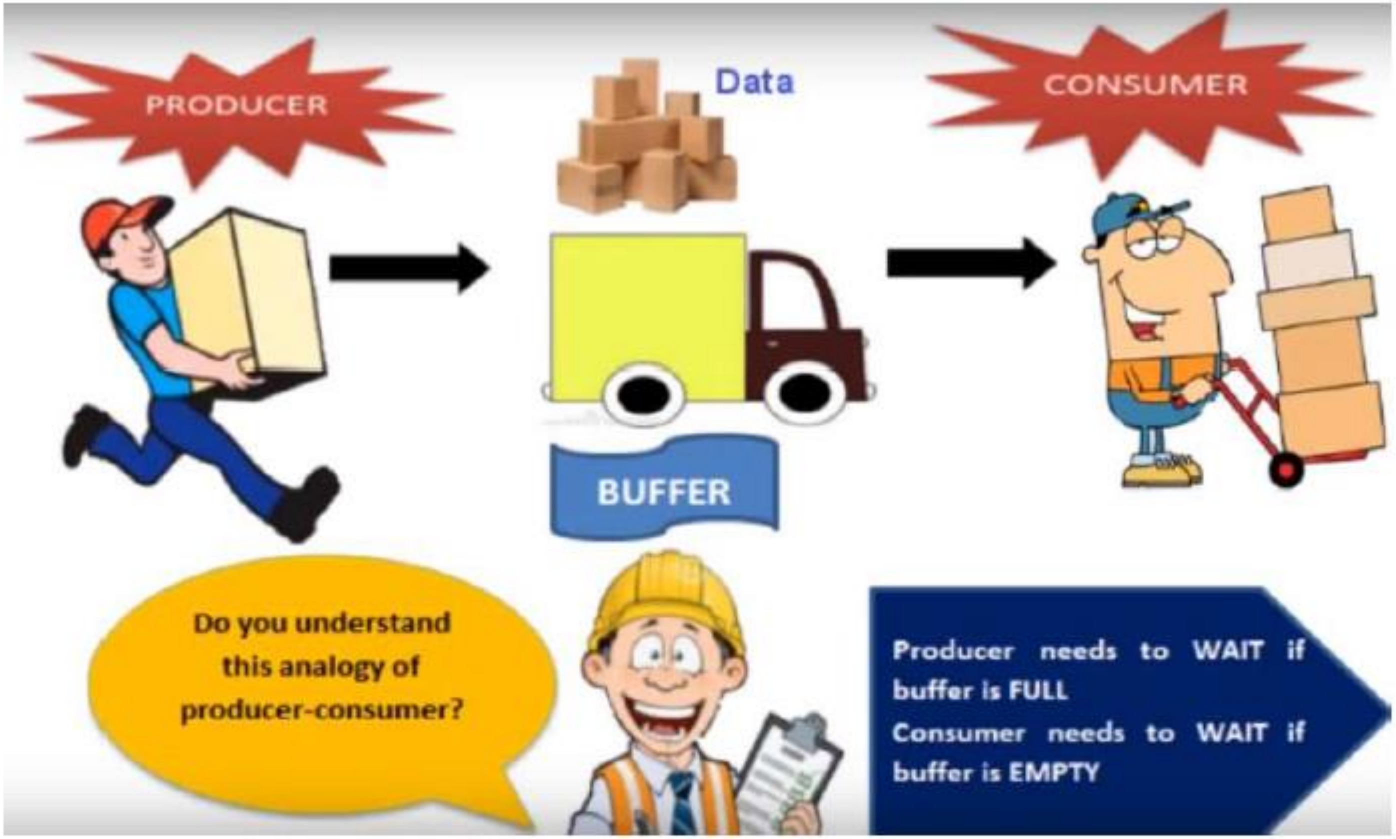
Picture 6 : Still many items

Picture 7 : Truck is full & unloaded

Picture 8 : Uloading

Picture 9 : Truck is empty

Picture 10 : Waiting



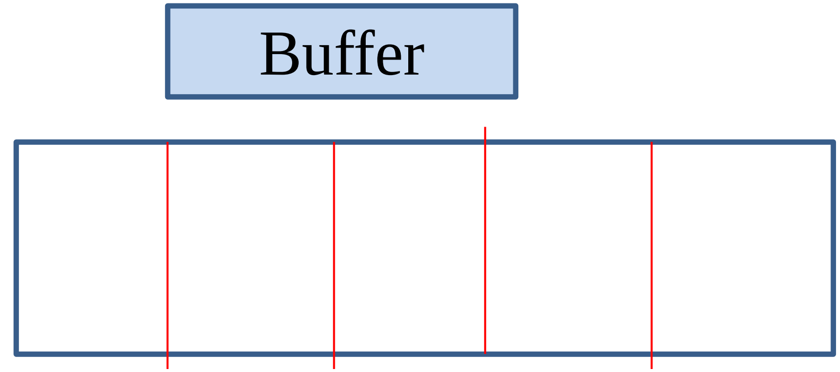
Picture: All Process

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

Empty=n=
Mutex=1
Full=0




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

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

Bounded Buffer Problem

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

Bounded Buffer Problem

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

 } while (TRUE);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write

    Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

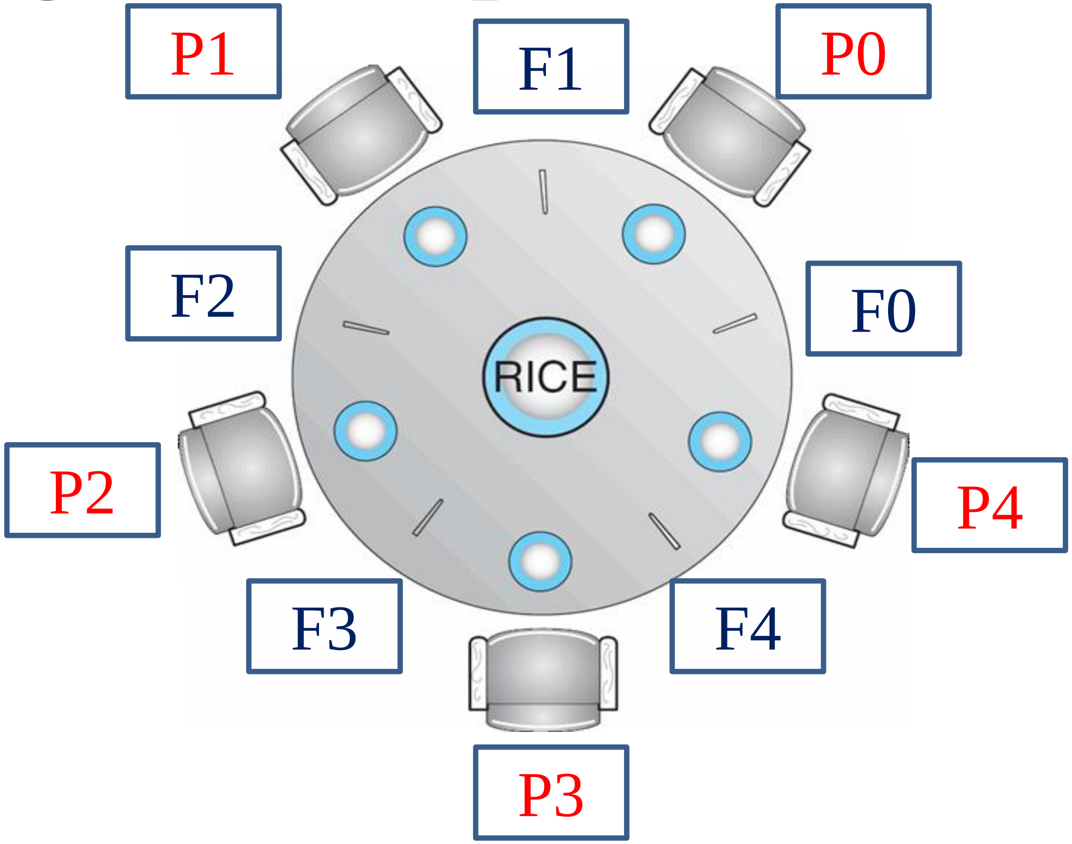
Readers-Writers Problem (

```
do {
        wait (mutex) ;
        readcount ++ ;
       if (readcount == 1)
            wait (wrt) ;

       signal (mutex)

   // reading is performed

       wait (mutex) ;
        readcount  - - ;
      if (readcount  == 0)
            signal (wrt) ;
      signal (mutex) ;
     } while (TRUE);
```

mutex=1
Wrt=1
readcount=0

```
do {
      wait (wrt) ;
   //  writing is performed
      signal (wrt) ;
   } while (TRUE);
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- 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);
```

S0	S1	S2	S3	S4
1	1	1	1	1

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

P-1

- The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0, S2 = 0.
- Process P0
- while(true)
- { wait(S0);
- print '0';
- release(S1);
- release(S2); }

Process P1
wait(S1);
release(S0);

Process P2
wait(S2);
release(S0);

- How many times will P0 print ‘0’?
- a) At least twice b) Exactly twice c) Exactly thrice d) Exactly once
- Ans: a
- Minimum no. of time 0 printed is twice when execute in this order (p0 -> p1 -> p2 -> p0)
- Maximum no. of time 0 printed is thrice when execute in this order (p0 -> p1 -> p0 -> p2 -> p0).

P-2(GATE 2016)

- A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution?
- **(A)** -2
  (B) -1
  (C) 1
  (D) 2

S=2
x=0

- W X Y Z

W	X	Y	Z
wait(s) Read(x) x=x+1 Write(x) Signal(s)	Wait(s) Read(x) x=x+1 Write(x) Signal(s)	wait(s) Read(x) x=x-2 Write(x) Signal(s)	Wait(s) Read(x) x=x-2 Write(x) Signal(s)

- Maximum value=2
- Minimum value=-4

- Maximum two process can enter in CS
- for maximum value,
- W enters in cs read x=0 and perform x=x+1 and pre-empted
- Y and Z executed x=-4
- W execute again and write x i.e. 1 and and
- process X executes x=2

- for minimum value,
- Y enters in cs read x=0 and perform x=x-2 and pre-empted
- W and X executed x=2
- Y execute again and write x i.e. -2 and and
- process Z executes x=-4

P-3

- Suppose we want to synchronize two concurrent processes P and Q using binary semaphores S1 and S2. The code for the processes P and Q is shown below-

Process P

```
while(1)
{ P(S1);
P(S2);
Critical Section
V(S1);
V(S2); }
```

Process Q

```
while(1)
{ P(S1);
 P(S2);
Critical Section
 V(S1);
V(S2); }
```

P-4

- Suppose we want to synchronize two concurrent processes P and Q using binary semaphores S1 and S2. The code for the processes P and Q is shown below-

Process P

```
while(1)
{ P(S1);
P(S2);
Critical Section
V(S1);
V(S2); }
```

Process Q

```
while(1)
{ P(S2);
 P(S1);
Critical Section
 V(S1);
V(S2); }
```

P-5

- Note – Consider lower number to have higher priority.

Process	Execution Time (Burst)	Priority	Arrival Time
P1	1	2	0
P2	7	6	1
P3	3	3	2
P4	6	5	3
P5	5	4	4
P6	15	10	5
P7	8	9	15

Average Waiting Time = (0+14+0+7+1+25+7)/7
= 54/7 = 7.71ms

P-6

- RR with TQ=4

Process ID	Arrival Time	Burst Time
1	0	5
2	1	6
3	2	3
4	3	1
5	4	5
6	6	4

Avg Waiting Time = (12+16+6+8+15+11)/6 =
76/6 units