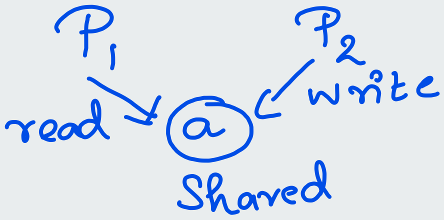
Theory



PROCESS SYNCHRONIZATION

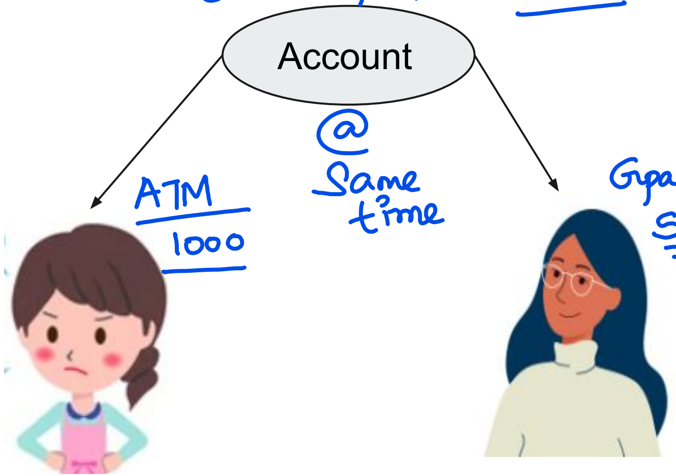
- Introduction- Producer Consumer
- Critical Section Problem
- Soln - Peterson, Synch H/W, Mutex, Semaphore ✓
- Classic problems of Synch ✓
- Monitor ✓

NEED

Because cooperating processes may access the shared data at same time which will cause data inconsistency.

O/P

Bal = 5000 ~~0~~ 4000



@ Same time

ATM 1000

Gpay 5000

⇒ ①

ATM(P_1)	Gpay(P_2)
Bal = 5000	Bal = 5000 ✓
txn → 1000	txn → 5000
Bal = 4000	Bal = 0

✗

Race condition

$P_1 P_2$

$P_2 P_1$

Idly Patti - Sapturaman

X 1 Kundan full $\Rightarrow$ patti idle

X 2 Kundan empty $\Rightarrow$ Sapturam idle

+2

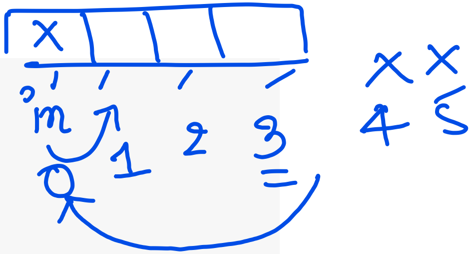
counter+2

counter

4

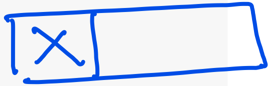
counter-4

array




```
#Idly producer ✓
while(true)
{
 1 /*Produce an idly*/
2 while(counter==kundan_size)
 ;/*do nothing/*
          Location
3 kundan[in]=produced_idly;
 in=(in+1)%kundan_size
4 counter++;
}
```

```
#Idly consumer
while(true)
{
① while(counter==0)
   ;/*Do nothing */
② consumed_idly = kundan[out];
   out = (out+1)% kundan_size;
③ counter--;
}
```



kundan

location

Producer - Consumer problem

idly => item

kundan => Buffer

```
#Idly producer
while(true)
{
    /*Produce an idly*/
 while(counter==kundan_size)
 ;/*do nothing/*

 kundan[in]=produced_idly
 in=(in+1)%kundan_size
 counter++;
}
```

Producer

```
while (true) {
      /* produce an item in produced_item*/


      while (counter == BUFFER_SIZE) ;
            /* do nothing */
      buffer[in] = produced_item;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

```
#Idly consumer
while(true)
{
    while(counter==0)
    ;/*Do nothing */
    consumed_idly = kundan[out];
    out = (out+1)% kundan_size;
    counter--;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        consumed_item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in consumed_item*/
}
```

Problem In Order of Execution

Buffer

10	20	30		

(++)

P

counter = 3

+1

counter = 4

(--) counter = 3

C

counter = 3

-1

counter = 2

1 PC ⇒ 2

2 CP ⇒ 4

X

Producer add

3 ⇒ 10, 20, 30, 40

Consumer

3 ⇒ 20, 30, 40

Race condition

Producer Consumer Problem

Problem: Given the common fixed-size buffer, the task is to make sure that the producer can't add data into the buffer when it is full and the consumer can't remove data from an empty buffer.

Solution: The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same manner, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Race Condition

- A race condition is a condition when there are many processes and every process shares the data with each other and accessing the data concurrently, and the output of execution depends on a particular sequence in which they share the data and access.

Shared Resources/data

Critical Section ✓

accnt

son — amma

P_1 → wait

$P_2 \rightarrow CS$

Checkpost ⟹ entry section — synch variables

Shared variables modify → critical section

reverse → exit section

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

CS ✓ P4

accnt

Solution to CS Problem

P1 → RS

P2 P3 P4

P1 X

P2 CS

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections ✓
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

S P1 X

P2 P3

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

→ Assume that each process executes at a nonzero speed

→ No assumption concerning **relative speed** of the *n* processes