Producer-Consumer:

```
int count = 0
void Consumer (void)
{   int itemc;
    while (true) {
→   while (count == 0);
    itemc = Buffer[out];
    out = (out + 1) mod n;
    count = count - 1;
→   process_item (itemc);
    }
}
```

```
load Rc, m[count];
DECR Rc;
Store m[count], Rc;
```

```
int count = 0
void Producer (void)
{   int itemp;
    while (true).
→   Produce_item(itemp);
→   while (count == n);
    Buffer[in] = itemp;
    in = (in + 1) mod n;
    count = count + 1;
}
}
```

```
load Rp, m[count]
INCR Rp;
Store m[count], Rp;
```

Critical Section

It is part of the program where shared resources are accessed by various processes.

```
main(){
    non critical section
    entry section
    Critical section
    exit Section
}
```

Synchronization mechanism of section,

Primary
1. Mutual Section → (CS single process execute 2nd, not multiple.
2. Progress

secondary
3. Bounded wait
4. No assumption related to H/w speed.

Mutual section →

P_1 process execute হওয়ার পর P_2 process execute হবে,

Progress: P_1 wants to execute the critical section কিন্তু P_2 stop the P_1. তাই progress হচ্ছে না,

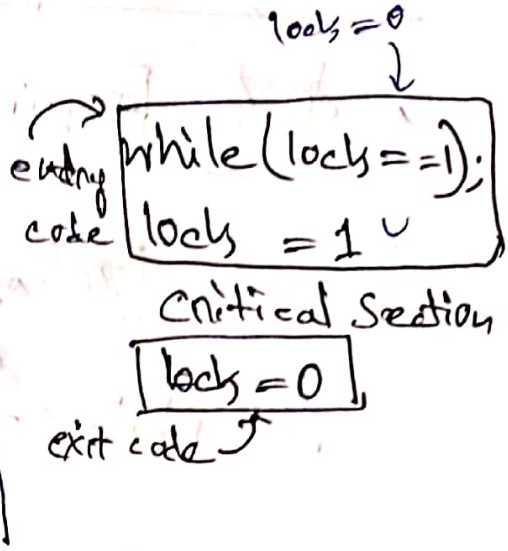
Bounded wait: P_1 process execute C.S. P_2 waits. After few times P_2 can process.

কিন্তু অর্থাৎ P_2 বা P_1 কে সুনির্দিষ্ট process করতে দেওয়া যাবে না,

No assumption: Hardware ভিত্তিতে আলো Bounded থাকতে পারতে না,

Critical section solution using 'Locks'

```
do {
        acquire lock
            CS
        release lock
}
```



→ multiple process can execute

Lock = 0 → vacant
1 → full

→ execute in user mode
→ Multiprocess solution
→ No mutual exclusion guarantee. (यदि preemption या entry section में होता तो mutual exclusion हो नहीं।

Synchronization H/W

→ All solution based on idea of locking.
Protecting critical regions via locks.
→ Provide special atomic Hardware instructions
Atomic = non-interruptible

CS solution using lock

```
do {
    acquire lock
        CS
    release lock
        remainder section
} while (TRUE);
```

Mutex lock

→ Protect cs by acquire() a lock then release() the lock.

→ acquire() and release() must be atomic.

→ This solution requires busy waiting.

→ lock called a spinlock.

```
acquire() {
    while (!available);
    available = false;
}
```

```
release() {
    available = True;
}
```

Binary Semaphone

S = 1 → depend on

```
wait(s){
while (s <=0);
 s = s-1;
}
```

```
signal (s){
   s = s+1;
}
```

Code:

```
do {
    entry section;
        critical section;
    exit section;
        reminder section;
}
while (True);
```

Counting Semaphore

S : $-\infty$ to $+\infty$ (No Fraction value)

```
Down(Semaphore S)
{  S value = S value - 1;
   if (S value < 0){
      put process(PCB) in
   suspended list sleep();
   }
else { return; }
}
```

```
Up (semaphore S)
{  S value = S value + 1;
   if (S value <= 0){
      Select a process
   from suspended list
   wake up();
   }
}
```

Solution of P-C using Semaphore: (Producer-Consumer)

producer (item p)

1) down (empty);
2) down (s);

```
Buffer [In] = item p;
In = (In+1) mod n;
```

3) up (s)
4) up (full)

consumer (item c)

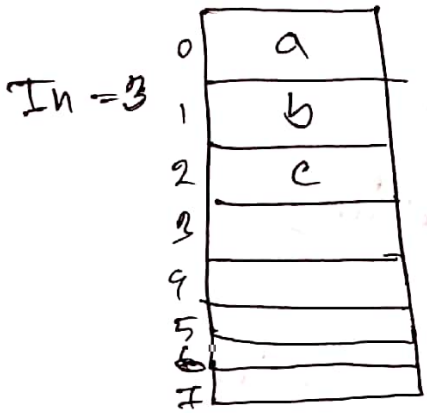
down (full);
down (s);

```
item c = Buffer [out];
out = (out+1) mod n;
```

up (s);
up (empty);

---

n=8



In = 3

Out = 0

full, empty, S, n, In, out
↓
variables

counting semaphore → full → no of filled slots.
→ empty → " " empty ".

(depend on)

Binary semaphore → S = 1 (এখানে S value 0 or 1 depend করে preemption হয় না, যদি Mutual exclution হয়).

Readers and writers Problem

(1) writer - writer : clash

(2) writer - reader : clash

(3) reader - reader : No clash

wrt = 1, mutex = 1, readcount = 0

→ depend on

Writer

```
wait (wrt);
    write operation;
signal (wrt);
```

Reader

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

entry

Read Operation

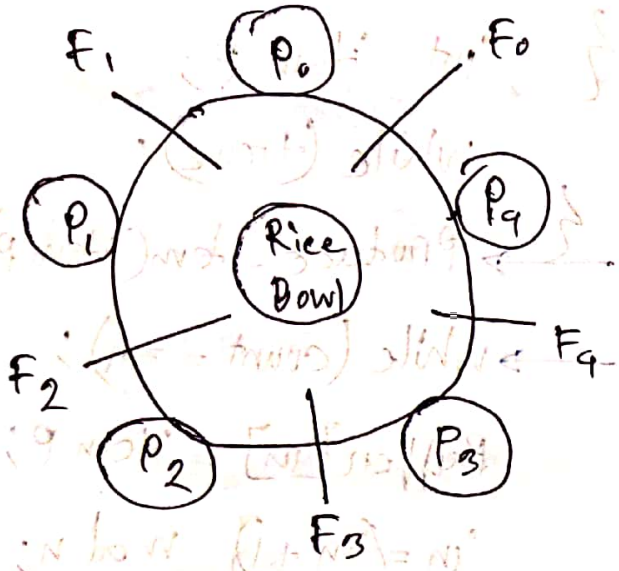
```
wait (mutex);
readcount--;
if (readcount == 0) {
        signal (wrt); }
signal (mutex);
```

exit

Dining - Philosophers

$P_i \longrightarrow (i)$

Standard system.



'5 Philosopens'

'5 Fonks'

eat
think

```
void philosophers (void)
{
  while (true)
  {
    Thinking ();
    take_fonks (i),
    take_fonks ((i+1)%N).
    EAT ();
    put fonks (i);
    put fonks ((i+1)%N);
  }
}
```

```
wait (take_fonks (Si));
wait (take_fonks (S(i+1)%N));
```

```
signal (put_fonks (Si));
signal put_fonks (S(i+1)%N)
```

Guess 5 semaphore s[i],

S_0	S_1	S_2	S_3	S_4
↓	↓	↓	↓	↓
1	1	1	1	1

Pesudocode:

```
void philosophen (void)
{
    while (true){
        Thinking();
        entry [wait(take_forks(Si));
               wait(take_forks(S(i+1) mod n));
        EAT();
        exit [signal(put_forks(Si));
              signal(put_forks(S(i+1) mod n));
    }
}
```

P_0 : S_0 S_1

P_1 : S_1 S_2

P_2 : S_2 S_3

P_3 : S_3 S_4

P_4 : S_4 S_0

* Here same time two philosophen can execute. But those have to independent.
* Deadlocks can occur.

Deadlock occur হয় যখন left Fork তোলা হয়, কিন্তু Right fork take করতে পারে না, যদি অন্য কোনো process preemption ঘটায়, তাহি Deadlock create হয় যখন কেউ EAT() করতে পারে না।

যদি Last philosopher reverse করে দেওয়া হয় তাহলে Deadlock Prevent করা যায়।

Bounded-Buffer Problem

Need three semaphore:

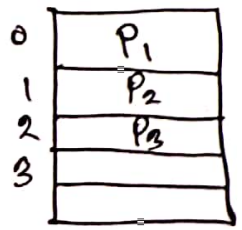
m (mutex), empty, full

Producer

```
do {
    wait(empty);
    wait(mutex);
    // add data to buffer
    signal(mutex);
    signal(full);
} while(True)
```

Consumer

```
do {
    wait(full);
    wait(mutex);
    // remove data from Buffer
    signal(mutex);
    signal(empty);
} while(True)
```



full = 3
mutex = 1
empty = 2

Petenson's Solution

Two processes → P_i and P_j

P_0

```
while (1)
{
    Flag[0] = T;
    Turn = 1;
    while (Turn==1 && Flag[1]==T);
    Critical section;
    Flag[0] = F;
}
```

P_1

```
while(1)
{
    Flag[1] = T;
    Turn = 0;
    while (Turn==0 && Flag[0]==T)
    Critical section;
    Flag[1] = F;
}
```

Turn = boolean variable

Flag = array, value → True, False