Process Synchronization

✓ Cooperating process → can be or can affect other Process executing in the system

- (i) directly share a logical address space (both code and data.)
- (ii) be allowed to share data only through file or manages.

✓ Producer Consumer Problem: A producer process Produces info that in consumed by a consumer Process.

- One solution to the producer-consumer problem uses shared memory.
- A producer can produce one item while the consumer in consuming another item.
- The producer and consumer must be synchronized.

- The buffer will reside in region of memory that is shard by the producer and consumer proces. (उत्पादक Producer info produce करे & consumer info consume करे)

2 types of buffer →

① Unbounded buffer → ✓ NO. size limit.

② Bounded buffer → ✓ size limit. fixed

✓ Counter++ : register = counter (reg = 5)

(word) register = register + 1 (reg = 6)

counter = register. (counter = 6)

✓ Counter--:

register$_2$ = counter (reg = 5)

register$_2$ = register$_2$ - 1 (= 4)

counter = register$_2$ (4)

Race condition:

- We need process synchronization for not to interfere with one another.

The Critical-Section problem:

- where the process maybe changing common variables, updating a table, writting a file, and so on.
- Two processes cannot execute same time.
- Rules: (I) Each processes must request permission to enter it's critical section.
  (II) code implementing → entry section.
  (III) The critical sec is followed by an exit sec.
  (IV) The remaining code in the reminder sec.

Koso

Test and set lock

- Hardware solution to the synchronization problem
- There is a shared lock variable which can take either of the two values, 0 or 1.
  (unlock) (lock)

Testing part:

- Before entering into the critical section, a process inquires about the lock.
- if it is locked, it keeps on waiting till it becomes free.
- if it is in un-locked, it ~~keeps on~~ takes the lock and executes the critical section

Atomic operation :-

two values (0, 1)

Code :-

```
boolean TestAndSet (boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Process P1

```
do {
    while (TestAndSet (&lock));
    // do nothing
    // critical section
    lock = FALSE;
    // remainder section
} while (TRUE);
```

passing value at the beginning

Process P2

```
do { while (TestAndSet (&lock));
    // do nothing
    // critical sec
    lock = FALSE;
    // remainder sec; } while (TRUE);
```

now it 1 because

=> output : 1 then will not care

locked

- it's satisfies mutual-exclusion
- it's not " bounded waiting

Semaphores : is a technique to manage concurrent process by using a simple integer value, which is known as a semaphore.

- is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in multiprocessing environment.
- S is an semaphore variable, two standard atomic operations → wait() & signal().

wait() → P ["to test"]

signal() → V ["to increment"]

Definition of wait(): (operation) / down. → ptest (used for)

```
P(Semaphore S){
        while(S<=0);
          S--; // no operation
    }
```

signal(): / up

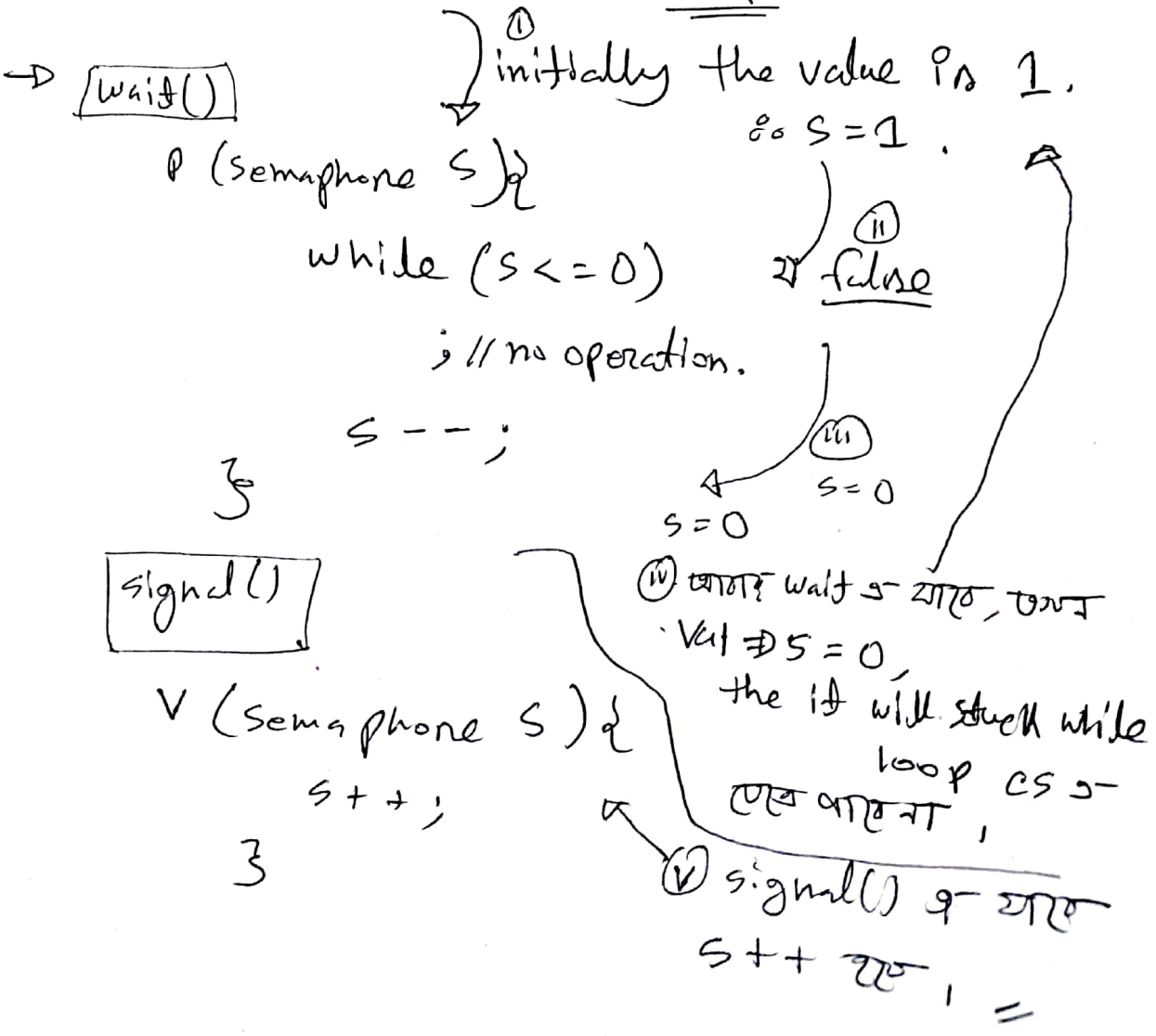
```
V(Semaphore S){
        S++;
    }
```

NOTE: All modifications to the integer value of the semaphore in the wait() and signal() operation must be executed indivisibly.

When one process is modifying in semaphore no other process cannot modify same sema val.

Types of semaphore

① **Binary semaphore:** The value of a binary semaphore can ~~read~~ range only between 0 & 1. On some system, binary semaphores are known as **muten locks**, as they are locks that provide mutual exclusion. **Steps**



(iv) Counting Semaphore: It's value can range over an unrestricted domain. It is used to control access to a resource has multiple instance.

wait()

```
P(Semaphore S){
    while (S<=0)
        ; // no operation
    S--
```

signal()

```
V(Semaphore S){
    S++;
}
```