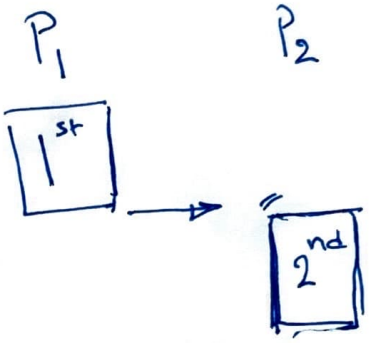
Process Synchronization

Race Condition

* Protecting shared data / resource.
* Coordination



Critical Section (CS)

```
do {
    entry section
        C.S
    exit section
        R.S
} while (1)
```

Conditions

1. Mutual Exclusion. ✓
2. Bounded waiting
3. Progress.

1. Disable / Enable interrupts
2. Simple hardware instruction (atomic).
   → TestAndSet (&lock).

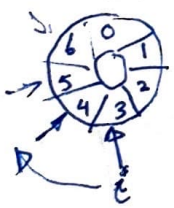
atomic

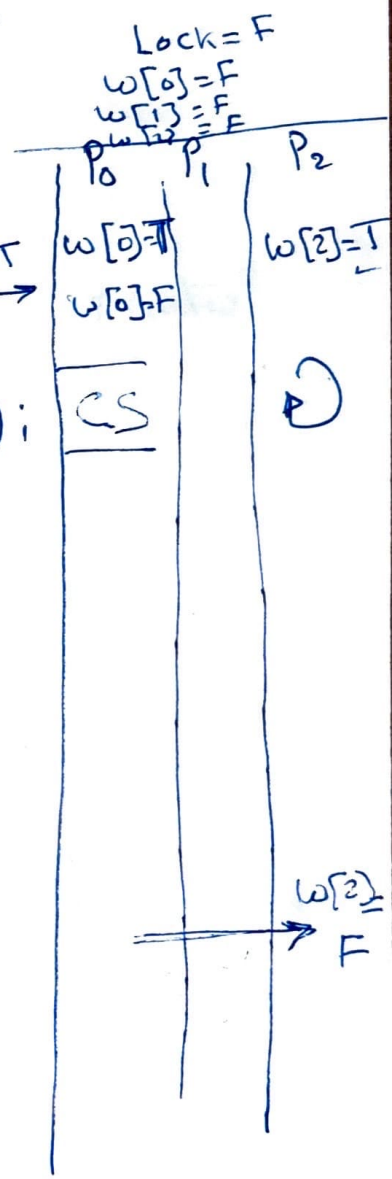
```
boolean TestAndSet (Boolean * target){
    boolean rv = *target;
    *target = TRUE;
    returns rv;
}
```

n processes.

boolean waiting[n]; } Initialized to False.
boolean lock;

```
Pi: do{  waiting[i] = TRUE;
         Key = TRUE;
         while (waiting[i] && Key)
             Key = TestAndSet(&lock);
         waiting[i] = FALSE;
         C.S
         j = (i+1) % n;
         while (j != i && !waiting[j])
             j = (j+1) % n
         If (j == i) ---> did full circle
             lock = False;
         else
             waiting[j] = False;
                  R.S
} while(TRUE);
```





3] Semaphores

An integer value that can be only accessed through 2 atomic operations: wait( ) and Signal( ),

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

```
Signal(S){
    S++;
}
```

Binary Semaphores — take values 0/1 — mutex

Counting Semaphores — unrestricted domain

Initialize S → 1

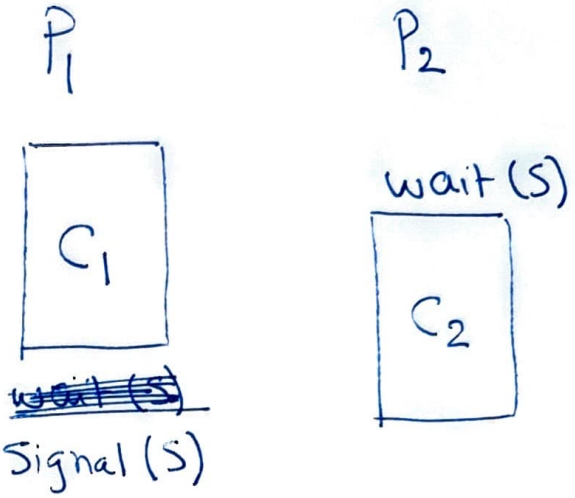
```
P: do{ wait(S);
       C.S
       Signal(S);
       R.S
     }while(TRUE);
```

[Solve n-processes Critical Section problem]

Initialize $S \rightarrow 0$?!

Solve coordination problems.

C_1 followed by C_2.



Counting Semaphores

Initialize $S \rightarrow n$ allows up to n processes to be in the Critical Section at a time.
