phead or pstart

```
phead=NULL;
pnew=malloc(...)
pnew->data=10;
phead->next=pnew;
pnew->next=phead;

p=phead;
while(p->next!=phead)
    //print p->data
------------------
pmid //middle node
while(p->next!=pmid)
   //print p->data
```

Inter Process Communication:-

Data Exchange -- sharing of data
Synchronization -- mutual exclusion
-- dependency/sequencing

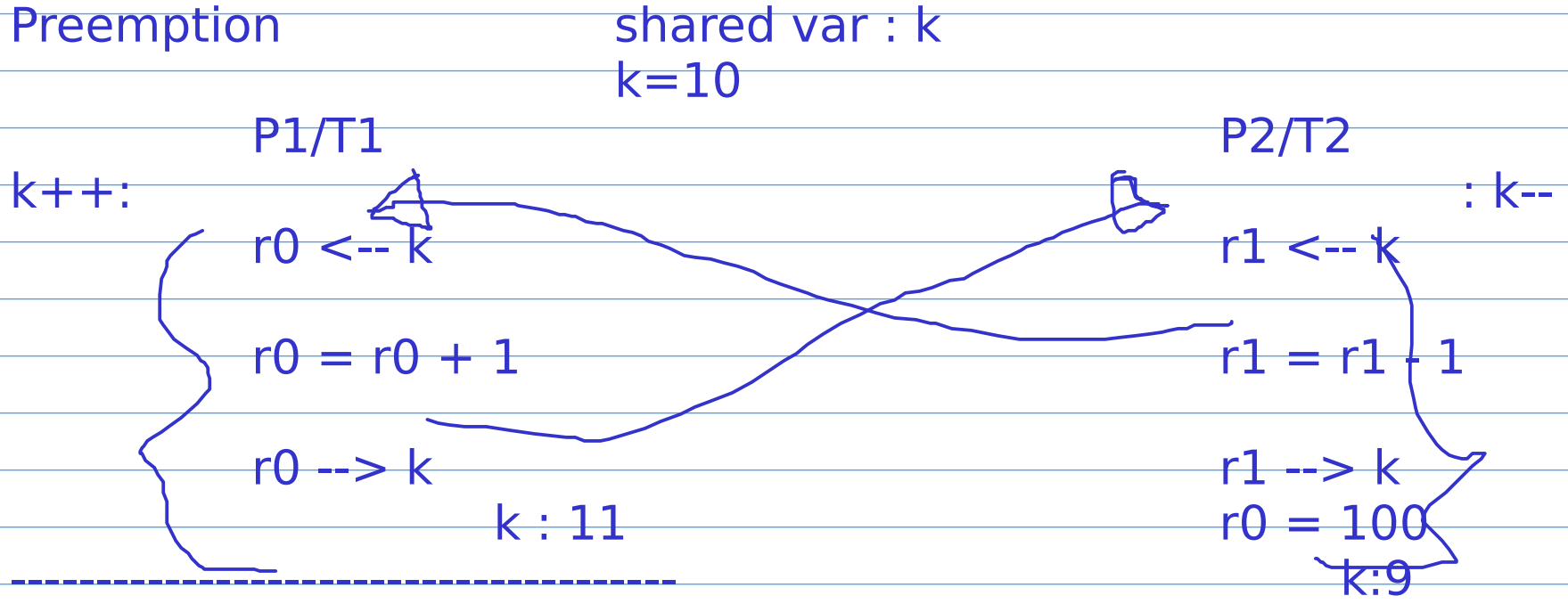
---

producer - consumer scenario
* a process/thread will add data -- producer
* a process/thread will remove data -- consumer
* common buffer / data source

* either producer or consumer, only can access common data at a time (shared resource) -- mutual exclusion
* consumer should block if buffer empty
* producer should block if buffer full

Memory read/write is not a blocking ops
File/Driver acess may be a blocking ops

Race Conditions:-



Atomic execution -- no switching in b'n

Critical Section:-
  * portion of code, accessing shared resources
  * no two critical sections should execute at same time (only
    one critical section should be under execution) -- Mutual Exclusion
  * possible ways
        - atomic execution of critical section
        - not allowing execution of other crticial sections

Solutions:-
* Atomic execution   -- disable interrupts
   challenges - not possible in user space, disabling interrupts
                for long, not applicable SMP
* H/w supported atomic instr --- XCHG in x86, SWP in arm
* busy waiting principle (or) spinning (HOLD)

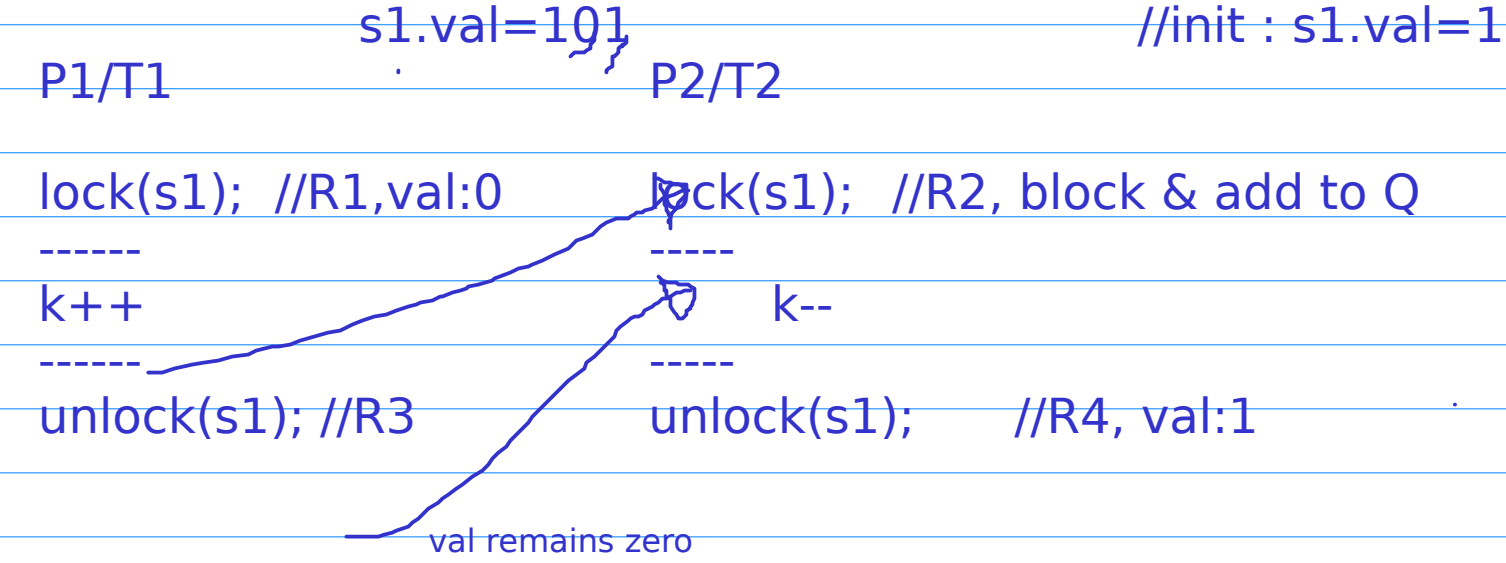
Common feasible solution in both userspace/kernel space
* Semaphores
* Mutex
* Spinlocks

lock operation (down/wait/acquire/take/get)

R1 : if sem.value > 0, --sem.value and go ahead, no blocking (A)
R2 : if sem.value == 0, block current process/thread and add to wait Q (B)

unlock operation (up/post/release/give/put)

R3 : if sem.waitQ is non empty remove only one process/thread
     from Q and resume (C), no change in semaphore value
R4 : if sem.waitQ is empty, ++sem.value (D)



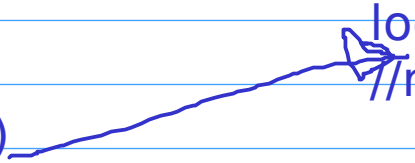
s1.val=0

Producer | Consumer

lock(s1)

//add | //remove

unlock(s1)



consumer schedules before producer??

lock(s1) --> R2 --> block

unlock(s1) --> R3 --> unblock consumer

producer executed before consumer??

unlock(s1) --> R4 --> s1.val=1

consumer can lock without blocking (R1)

sequencing/depency b'n producer & consumer

Issues with semaphore:-

* not unlocking at end
* locking twice / unlocking twice
* unlocking before locking (intention : mutual exclusion)

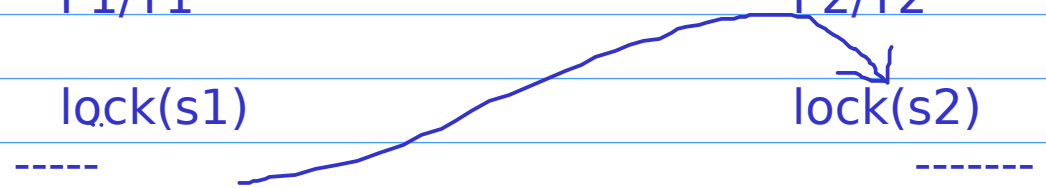
Binary Semaphore : 0 or 1 only

Counting Semaphore : 0 or any +ve any value

Deadlock:-

circular dependency b'n two or more processes/threads
infinitely blocked (forever)

Digital Copy -- Printer(s1), Scanner(s2)



P1/T1	P2/T2
lock(s1)	lock(s2)
lock(s2)	lock(s1)
//copy	//copy
unlock(s2)	unlock(s1)
unlock(s1)	unlock(s2)

Avoid deadlocks:-

* Atomic locking , whenever possible (lock all or none)

* Follow some order
* Don't apply mutual exclusion before resolving dependencies
* Use waitpid towards end (after unlocking resources)

Just Binary Semaphore is not a Mutex

```
ret=fork();
if(ret==0) {
    lock(s1)
    exit(0);
}
else {
    waitpid(....)          //order??
    unlock(s1)
}
```

Mutex:-

- strictly for mutual exclusion [symmetric lock/unlock]
- subset of semaphore operations
- ownership applies
- Unlocking before locking or unlocking more than once not allowed

---

mutex APIs from pthread lib:-

```
	pthread_mutex_t m1;

	pthread_mutex_init(&m1,NULL);
	pthread_mutex_destroy(&m1);

	pthread_mutex_lock(&m1);			//before crtitical
	pthread_mutex_unlock(&m1);			//after critical
```

```
//base:- pconcur.c/psingle.c

printf("A--welcome\n");
for(i=1;i<=max;i++)
{
	printf("A--%d\n",i);
	sleep(1);
}
```

TODO:- apply mutex to avoid intermixed output b'n threads

```
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;   //static init
```

Semaphores:- softprayog.in

POSIX Semaphores (light weight, real time)

* unnamed semaphores , typically used in threads, where vars can be shared
* named semaphores , for independent processes

Sys V Semaphore

POSIX Unnamed Semaphores:-

```
sem_t s1;

sem_init(&s1, 0, 1); //2nd param - 0 - same addrspace (threads)
                     //3rd param - 1 - initial value
sem_destroy(&s1);

sem_wait(&s1);       //lock, before critical section
sem_post(&s1);       //unlock, after critical
```

TODO:- apply semaphore operations to race cond example

```
sem_t *ps,*qs;
ps=sem_open("s1",O_CREAT, 0666, 1);
qs=sem_open("s2",O_CREAT, 0666, 0);
ret=fork();
if(ret==0)
        //child
else
        //parent
```

IPC - Message Queues , Shared Memory
Scheduling
File System
Memory Management