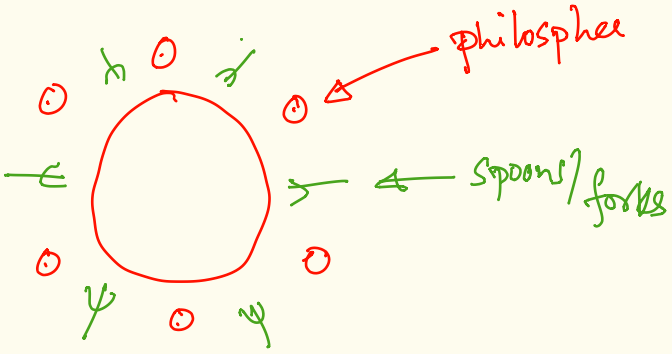
CS347m | CS219 Lecture 19

spinlocks, mutex

deadlock

1 dining philosophers problem




```
philosopher() {
    think()
    grabspoons()   — left & right spoon
    eat()
    returnspoons()
}
```

for potential deadlock!

- solution?
  - force change in order for atleast one philosopher
  - introduce a lock for the table.

```
getlock(table)
    grabspoons;
unlock(table)
```

0 nested interrupts.

```
lock(L) {
    lock(L) {
    }
}
```

single CPUs

deadlock: instances of execution waiting a condition that will never be TRUE.

2

```
T1                  T2
lock(L1)            lock(L2)
lock(L2)            lock(L1)
```

dealock!

bad logic / bug

```
T1                  T2
lock(L1)            lock(L2)
count++;            count++;
unlock(L1)          unlock(L2)
```

(#) spinlock, mutex

(3) semaphore

- init — initialize value of a semaphore
- down — if (s > 0) s--; else sleep(s);
- up — s++; wakeup(s);

atomic

```
S: semaphore
int val;
spinlock L;
```

```
down(semaphore *s) {
    spinlock(s->L)
    while (s->val <= 0)
        sleep(s);
    s->val--;
    spinunlock(s);
}
```

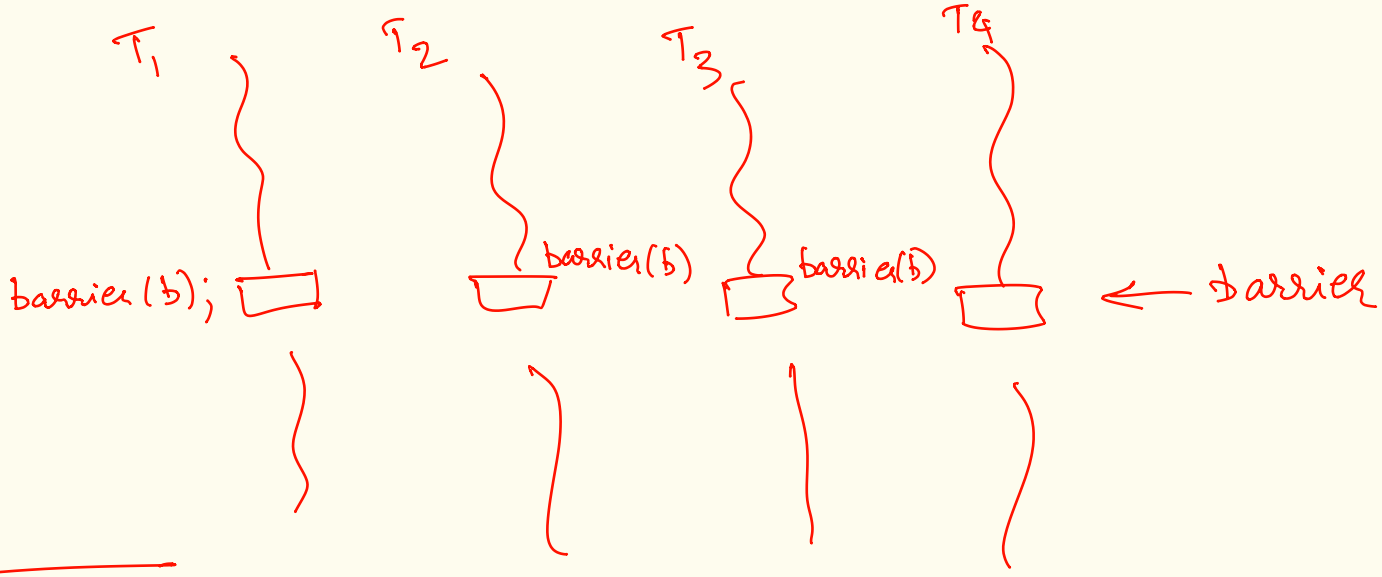
```
up(semaphore *s) {
    spinlock(s);
    s->val++;
    wakeup(s);
    spinunlock(s);
}
```

---

condition variables based mutual exclusion

- generalized condition on which ~~mutex~~ mutual exclusion is guaranteed.
  - queuelength ≤ 90% capacity
    - 10% NCHILD goto ZOMBIE
  - variable based compound general cond'n.

④ **barrier** — wait at a point/ loc^n / instruction till all instances reach that location.



Q ~~who~~ how to implement using semaphores?

~ barrier.init(, K) ← init # of threads waiting for barrier.

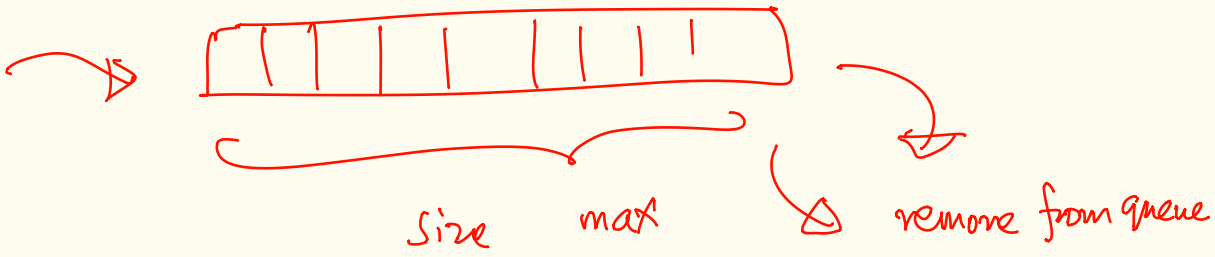
```
barrier (Barrier * b) {

    down (S1);
      if (S1==0)
          S2.init(K);
    down (S2);

}
```

```
barrierinit (Barrier *b, int K){

    b->S1->init (K);

    b->S2->init (0);

}
```

```
struct Barrier {
    semaphore* S1;      ~ init
    semaphore* S2;        down
                          up
}
```

HW1 → re-implement barrier using condition variables.

HW2 → implement producer-consumer shared add/delete to queue using semaphores.



if q_{size} == MAX
cannot produce

if q_{size} == 0
cannot consume

} constraints + synchronization