mutual exclusion

MEP(mutual exclusion problem)

1. Mutex: At any point in time, there is at most one thread in the critical section
2. Absence of livelock: If various threads try to enter the critical section, at least one of them will succeed
3. Free from starvation: A thread trying to enter its critical section will eventually be able to do so

await

```
while(cond){} == await (!cond)
```

await: false to loop, true to continue next excution

attempt 1 take truns

```
int turn = 1;
Thread . start { // P
    // non - critical section
    await (turn ==1);
    // CRITICAL SECTION
    turn = 2;
    // non - critical section
}
Thread . start { // Q
    // non - critical section
    await (turn ==2);
    // CRITICAL SECTION
    turn = 1;
    // non - critical section
}
```

mutex: yes, absence live lock: yes, Free from startvation:no(a process could remain indefinitely in its non-critical section)

Semaphore

Dining Philiosophores

```
Semaphore[ ] forks = [1,...,1]; //N
Semaphore chairs = new Semaphore(N-1);
```

Man/Woman Restroom

```
import java.util.concurrent.Semaphore;

int man = 0, woman=0;
Semaphore manmutex = new Semaphore(1);
Semaphore womanmutex = new Semaphore(1);
Semaphore toilets = new Semaphore(n);
Semaphore restroom = new Semaphore(1);

Thread.start {
    // man
    manmutex.acquire();
    if(man == 0){
        restroom.acquire();
    }
    man++;
    manmutex.release();

    toilets.acquire();
    // man in restroom
    toilets.release();

    manmutex.acquire();
    man--;
    if(man==0){
        restroom.release();
    }
    manmutex.release();
}

Thread.start {
    // woman
    womanmutex.acquire();
    if(woman == 0){
        restroom.acquire();
    }
    woman++;
    womanmutex.release();

    toilets.acquire();
    // man in restroom
    toilets.release();

    womanmutex.acquire();
    woman--;
    if(woman==0){
        restroom.release();
    }
    womanmutex.release();
}
```

Ferry Move

```
import java.util.concurrent.Semaphore;

int max=?,cur=0,coast=0;
Semaphore permitOn = new Semaphore(1);
Semaphore permitOff = new Semaphore(0);
Semaphore ferryMove = new Semaphore(0);

Thread.start {
// ferry
while(true){
ferryMove.acquire();
coast=1-coast;
// move
permitOff.release();
}
}

Thread.start {
// passengers
permitOn.acquire();
cur++;
if(cur==max){
ferryMove.release();
}else{
```

permitOn.release();
}

```
permitOff.acquire();
cur--;
if(cur==0){
    permitOn.release();
}else{
    permitOff.release();
}
```

}

three stations to clean car

```
import java.util.concurrent.Semaphore;

Semaphore blast = new Semaphore(1);
Semaphore rinse = new Semaphore(1);
Semaphore dry = new Semaphore(1);

Semaphore waitSet[3] = {new Semaphore(0),new
Semaphore(0),new Semaphore(0)};

Thread.start { // car
    waitSet[0].release();
    blast.acquire();
    waitSet[1].release();
    rinse.acquire();
    exwaitSetist[2].release();
    dry.acquire();
}

Thread.start { // blast
    while(true){
        waitSet[0].acquire();
        // do blast
        blast.release();
    }
}

Thread.start { // rinse
    while(true){
        waitSet[1].acquire();
        // do rinse
        rinse.release();
    }
}

Thread.start { //dry
    while(true){
        waitSet[2].acquire();
        // do dry
        dry.release();
    }
}
```

Monitor

synchronized,Condition,monitor
notify(),notifyAll(),wait()

Buffer

```
class Buffer {
    Object buffer = null; // shared buffer
    synchronized Object consume() {
        while (buffer == null)
            wait()
        Object aux = buffer;
        buffer = null;
        notifyAll();

        return aux;
    }

    synchronized void produce(Object o) {
        while (buffer != null)
            wait();
        buffer = o;
        notifyAll();
    }
}
```

Reader/Writers

```
monitor RW {
  int readers = 0;
  int writers = 0;
  condition OKtoRead , OKtoWrite ;

  public void StartRead () {
    while ( writers != 0 or !OKtoWrite . empty
()) {
      OKtoRead . wait ();
    }
    readers = readers + 1;
  }

  public void EndRead {
    readers = readers - 1;
    if ( readers ==0) {
      OKtoWrite . notify ();
    }
  }
  public void StartWrite () {
    while ( writers != 0 or readers != 0) {
      OKtoWrite . wait ();
    }
    writers = writers + 1;
  }

  public void EndWrite () {
    writers = writers - 1;
    OKtoWrite . signal ();
    OKtoRead . signalAll ();
  }
}
```

It gives priority to readers over writers.