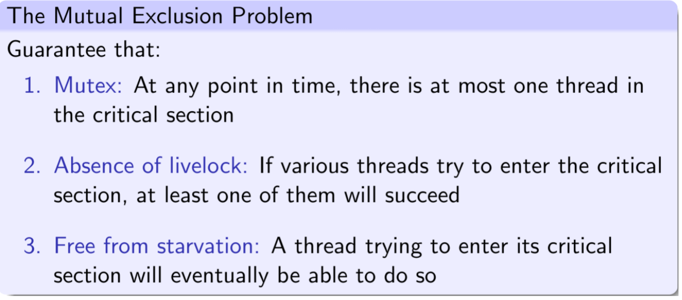
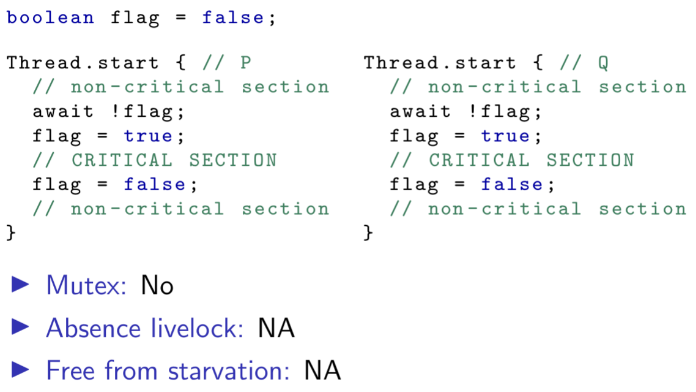


The Mutual Exclusion Problem

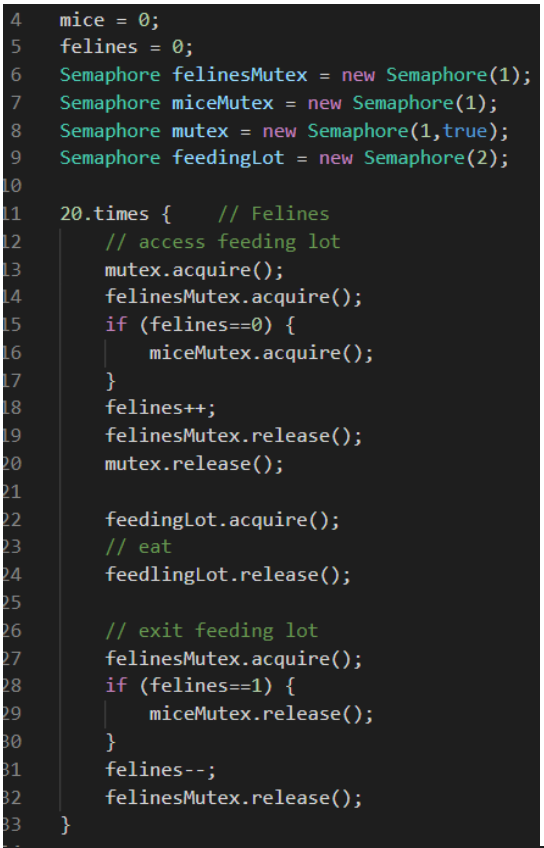
Guarantee that:

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

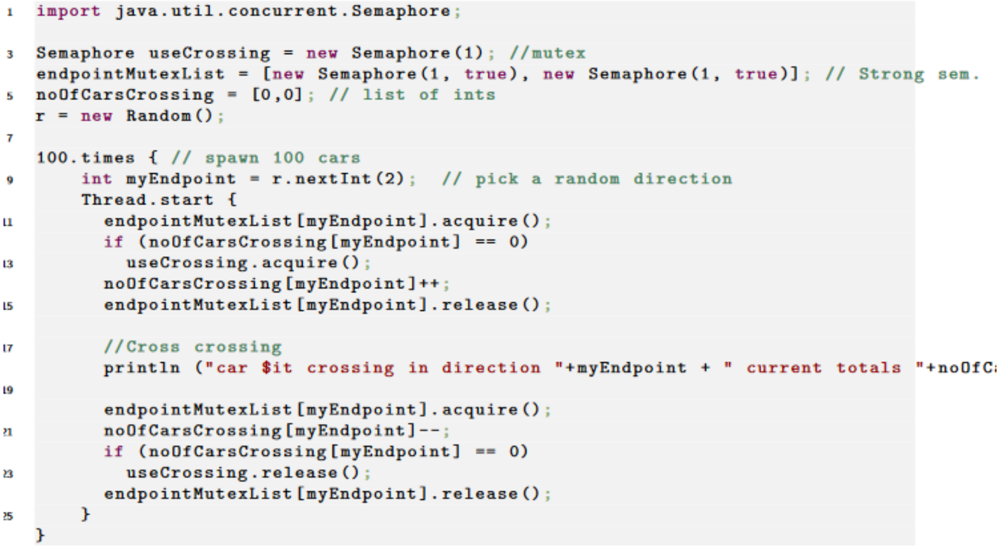


```
boolean flag = false;

Thread.start { // P              Thread.start { // Q
  // non-critical section          // non-critical section
  await !flag;                     await !flag;
  flag = true;                     flag = true;
  // CRITICAL SECTION              // CRITICAL SECTION
  flag = false;                    flag = false;
  // non-critical section          // non-critical section
}                                }
```

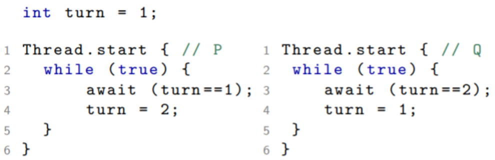
- Mutex: No
- Absence livelock: NA
- Free from starvation: NA



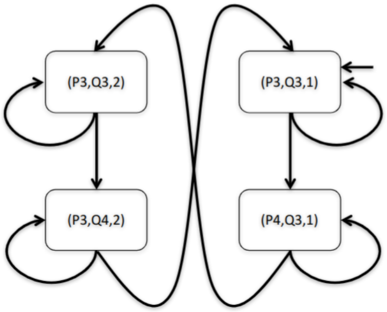
```
mice = 0;
felines = 0;
Semaphore felinesMutex = new Semaphore(1);
Semaphore miceMutex = new Semaphore(1);
Semaphore mutex = new Semaphore(1,true);
Semaphore feedingLot = new Semaphore(2);

20.times {    // Felines
    // access feeding lot
    mutex.acquire();
    felinesMutex.acquire();
    if (felines==0) {
        miceMutex.acquire();
    }
    felines++;
    felinesMutex.release();
    mutex.release();

    feedingLot.acquire();
    // eat
    feedlingLot.release();

    // exit feeding lot
    felinesMutex.acquire();
    if (felines==1) {
        miceMutex.release();
    }
    felines--;
    felinesMutex.release();
}
```



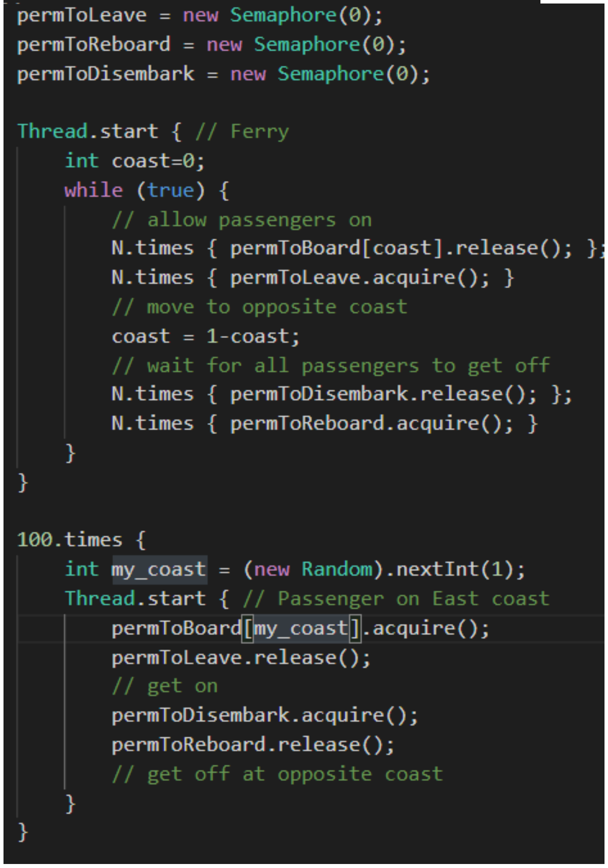
```
import java.util.concurrent.Semaphore;

Semaphore useCrossing = new Semaphore(1); //mutex
endpointMutexList = [new Semaphore(1, true), new Semaphore(1, true)]; // Strong sem.
noOfCarsCrossing = [0,0]; // list of ints
r = new Random();

100.times { // spawn 100 cars
    int myEndpoint = r.nextInt(2);  // pick a random direction
    Thread.start {
      endpointMutexList[myEndpoint].acquire();
      if (noOfCarsCrossing[myEndpoint] == 0)
        useCrossing.acquire();
      noOfCarsCrossing[myEndpoint]++;
      endpointMutexList[myEndpoint].release();

      //Cross crossing
      println ("car $it crossing in direction "+myEndpoint + " current totals "+noOfC

      endpointMutexList[myEndpoint].acquire();
      noOfCarsCrossing[myEndpoint]--;
      if (noOfCarsCrossing[myEndpoint] == 0)
        useCrossing.release();
      endpointMutexList[myEndpoint].release();
    }
}
```



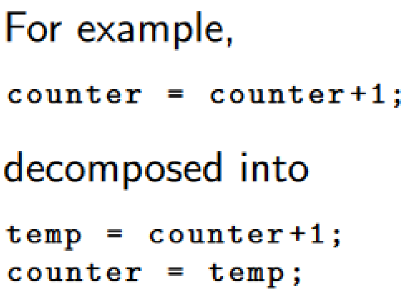
```
int turn = 1;

Thread.start { // P          Thread.start { // Q
  while (true) {               while (true) {
      await (turn==1);            await (turn==2);
      turn = 2;                   turn = 1;
  }                            }
}                            }
```



Mutex: Holds if all accessible states do not contain a state of the form (p4,q4,turn) for some value of turn.



```
permToLeave = new Semaphore(0);
permToReboard = new Semaphore(0);
permToDisembark = new Semaphore(0);

Thread.start { // Ferry
    int coast=0;
    while (true) {
        // allow passengers on
        N.times { permToBoard[coast].release(); };
        N.times { permToLeave.acquire(); }
        // move to opposite coast
        coast = 1-coast;
        // wait for all passengers to get off
        N.times { permToDisembark.release(); };
        N.times { permToReboard.acquire(); }
    }
}

100.times {
    int my_coast = (new Random).nextInt(1);
    Thread.start { // Passenger on East coast
        permToBoard[my_coast].acquire();
        permToLeave.release();
        // get on
        permToDisembark.acquire();
        permToReboard.release();
        // get off at opposite coast
    }
}
```

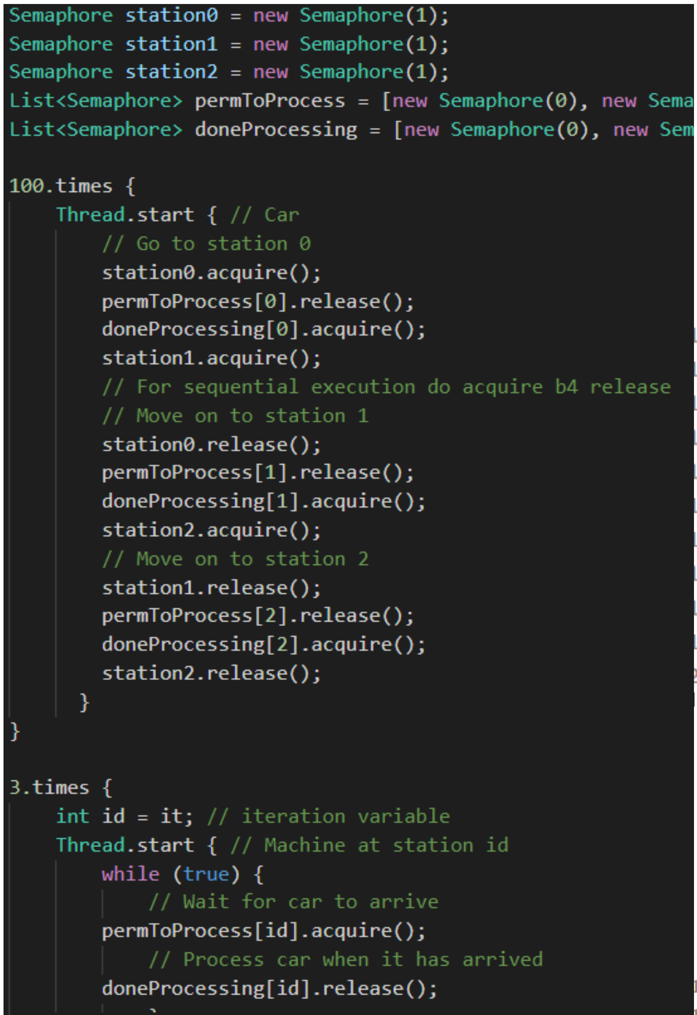


For example,

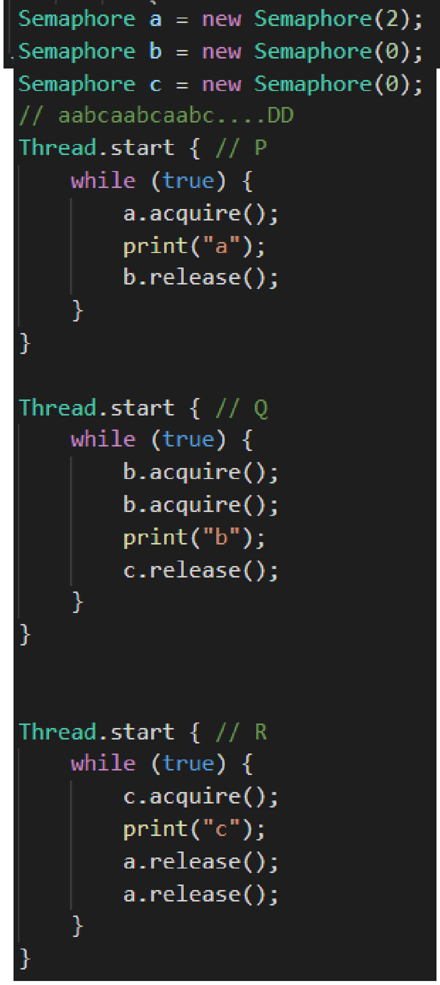
```
counter = counter+1;
```

decomposed into

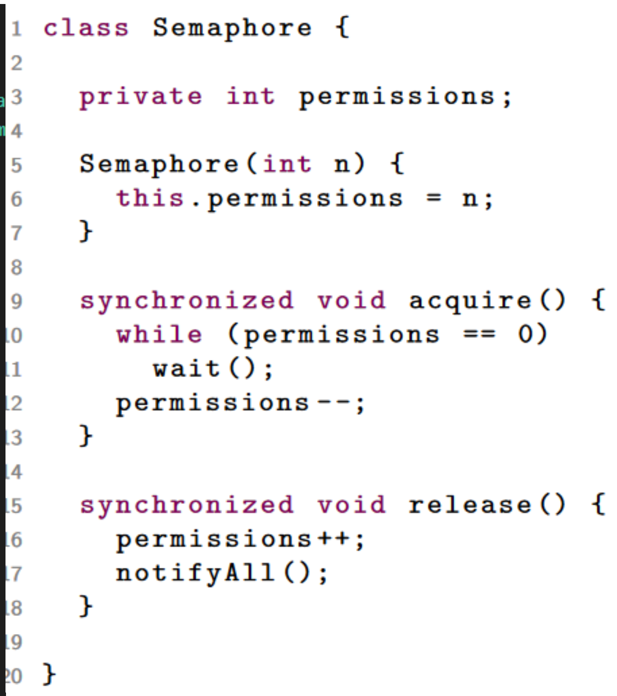
```
temp = counter+1;
counter = temp;
```



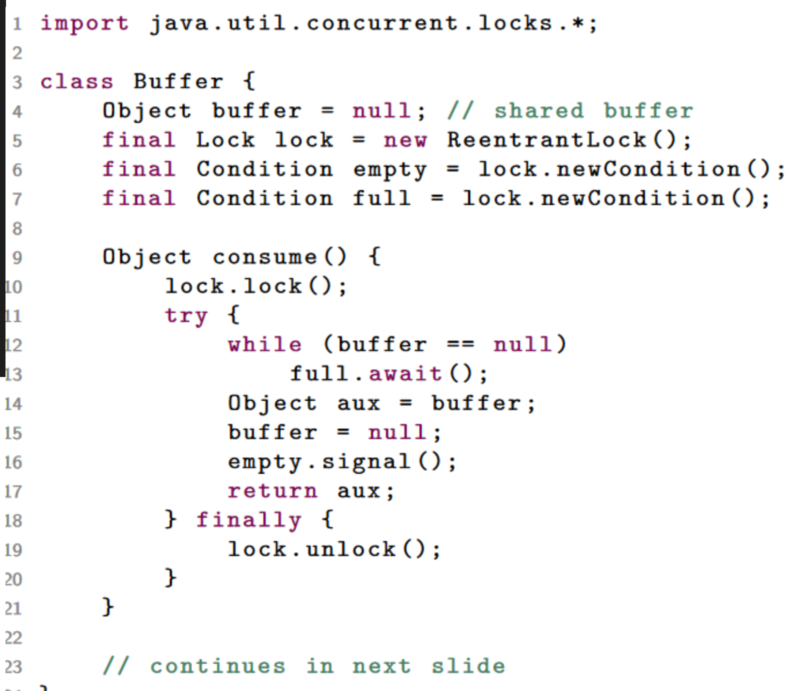
```
Semaphore station0 = new Semaphore(1);
Semaphore station1 = new Semaphore(1);
Semaphore station2 = new Semaphore(1);
List<Semaphore> permToProcess = [new Semaphore(0), new Sema
List<Semaphore> doneProcessing = [new Semaphore(0), new Sem

100.times {
    Thread.start { // Car
        // Go to station 0
        station0.acquire();
        permToProcess[0].release();
        doneProcessing[0].acquire();
        station1.acquire();
        // For sequential execution do acquire b4 release
        // Move on to station 1
        station0.release();
        permToProcess[1].release();
        doneProcessing[1].acquire();
        station2.acquire();
        // Move on to station 2
        station1.release();
        permToProcess[2].release();
        doneProcessing[2].acquire();
        station2.release();
      }
}

3.times {
    int id = it; // iteration variable
    Thread.start { // Machine at station id
        while (true) {
            // Wait for car to arrive
        permToProcess[id].acquire();
            // Process car when it has arrived
        doneProcessing[id].release();
```



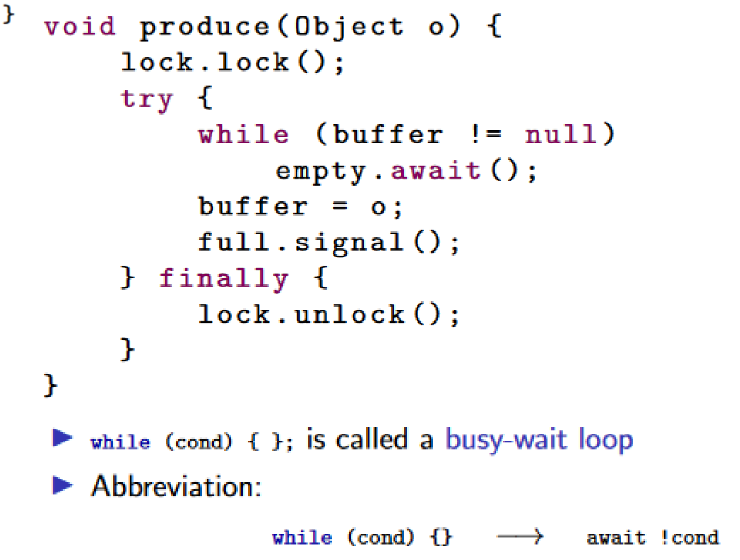
```
Semaphore a = new Semaphore(2);
Semaphore b = new Semaphore(0);
Semaphore c = new Semaphore(0);
// aabcaabcaabc....DD
Thread.start { // P
    while (true) {
        a.acquire();
        print("a");
        b.release();
    }
}

Thread.start { // Q
    while (true) {
        b.acquire();
        b.acquire();
        print("b");
        c.release();
    }
}


Thread.start { // R
    while (true) {
        c.acquire();
        print("c");
        a.release();
        a.release();
    }
}
```



```
class Semaphore {

  private int permissions;

  Semaphore(int n) {
    this.permissions = n;
  }

  synchronized void acquire() {
    while (permissions == 0)
      wait();
    permissions--;
  }

  synchronized void release() {
    permissions++;
    notifyAll();
  }

}
```



```
import java.util.concurrent.locks.*;

class Buffer {
    Object buffer = null; // shared buffer
    final Lock lock = new ReentrantLock();
    final Condition empty = lock.newCondition();
    final Condition full = lock.newCondition();

    Object consume() {
        lock.lock();
        try {
            while (buffer == null)
                full.await();
            Object aux = buffer;
            buffer = null;
            empty.signal();
            return aux;
        } finally {
            lock.unlock();
        }
    }

    // continues in next slide
}
```



```
void produce(Object o) {
    lock.lock();
    try {
        while (buffer != null)
            empty.await();
        buffer = o;
        full.signal();
    } finally {
        lock.unlock();
    }
}
```

- while (cond) { }; is called a busy-wait loop
- Abbreviation:

  while (cond) {} ⟶ await !cond