Recall: Locks: Loads/Stores

- This implementation does not ensure mutual exclusion, since both threads may grab the lock:
- After Thread 1 reads flag==0 and exits the while loop, it is preempted/interrupted by Thread 2, which also reads flag==0 and exits the while loop. Then both threads set flag=1 and enter the critical section.
- Root cause: Lock is not an atomic operation!

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1)  // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1;          // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

flag = 0

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Mutual Exclusion I

```
Boolean flag[2];
flag[0]=false, flag[1]=false;
```

```
//Thread T0
while (true) {
    while (flag[0]==flag[1]);
    //Critical section
    flag[0]=flag[1];
}
```

```
//Thread T1
while (true) {
    while (flag[0]!=flag[1]);
    //Critical section
    flag[0]=!flag[1];
}
```

- Does it achieve one of more of the correctness properties of a concurrent program:
  - Mutual exclusion: Only one thread in critical section at a time
  - Progress (deadlock-free): If several simultaneous requests, must allow one to proceed
  - Bounded waiting (starvation-free): Must eventually allow each waiting thread to enter
- Does it need the TestAndSet() instruction for atomic execution like the previous slide “Locks: Loads/Stores”?
- What is its major flaw?
- ANS:

Mutual Exclusion II

```
Boolean flag[2];
flag[0]=false, flag[1]=false;
```

```
//Thread T0
while (true) {
    flag[0] = true;
    while (flag[1]==true);
    /* Critical Section */
    flag[0] = false;
}
```

```
//Thread T1
while (true) {
    flag[1] = true;
    while (flag[0]==true);
    /* Critical Section */
    flag[1] = false;
}
```

- Does it achieve one of more of the correctness properties of a concurrent program:
  - Mutual exclusion: Only one thread in critical section at a time
  - Progress (deadlock-free): If several simultaneous requests, must allow one to proceed
  - Bounded waiting (starvation-free): Must eventually allow each waiting thread to enter
- ANS:

Mutual Exclusion III (Peterson's Solution)

```
Boolean flag[2];
flag[0]=false, flag[1]=false;
int turn = 0;
```

```
//Thread T0
while (true) {
    flag[0] = true;
    turn = 1;
    while (flag[1]==true && turn==1);
    /* Critical Section */
    flag[0] = false;
}
```

```
//Thread T1
while (true) {
    flag[1] = true;
    turn = 0;
    while (flag[0]==true && turn==0);
    /* Critical Section */
    flag[1] = false;
}
```

- Does it achieve one of more of the correctness properties of a concurrent program:
  - Mutual exclusion: Only one thread in critical section at a time
  - Progress (deadlock-free): If several simultaneous requests, must allow one to proceed
  - Bounded waiting (starvation-free): Must eventually allow each waiting thread to enter
- ANS: