What is a `lock`?

What we get without hardware support – Peterson’s Algorithm

```c
shared int interested[2];
shared int turn;
void init()
{
    interested[0] = 0;
    interested[1] = 0;  // 1 -> thread wants lock
    turn = 0;       // whose turn? (thread 0 or 1?)
}

void lock(int me) // my number is 0 or 1
{
    int other = 1 - me; // works because ‘me’ equals 0 or 1
    interested[me] = 1;  // I’m interested in doing critical stuff
    turn = other;       // but I’m patient … make it other thread’s turn
    while ((interested[other] == 1) && (turn == other)) ;  // spin-lock ick!
}

void unlock(int me)
{
    interested[me] = 0; // no longer interested
}
```

How is this code used?
How do you actually code with locks?

```c
#include "pthread.h"
void mycode ( int balance )
{
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  // init is crucial, can also call an init function
    Pthread_mutex_lock(&lock);  // wrapper for pthread_mutex_lock()
    balance = balance + 1;  // our critical section
    Pthread_mutex_unlock(&lock);
}
```

// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex)
{
    int rc = pthread_mutex_lock(mutex);  // use pthread library to ask for a lock
    assert(rc == 0);
}

If you’re an OS designer implementing locks, what do we need to think about?

- **Mutual exclusion**
  - do we ensure one at a time

- **Fairness**
  - every one gets a turn

- **Performance**
  - lock / unlock 

Hardware Solution #1:

```c
void lock()
{
    DisableInterrupts();
}
void unlock()
{
    EnableInterrupts();
}
```
Hardware Solution #2

```c
int TestAndSet(int *old_ptr, int new)
{
    int old = *old_ptr;    //fetch old value
    *old_ptr = new;        //store new into old_ptr
    return old;            //return old value
}
```

```c
void lock(lock_t *lock)
{
    while(TestAndSet(&lock->flag, 1) == 1) // sit and spin
}
```

//unlock function just sets lock to 0

For example, x86's xchg instruction **atomically** implements test and set.

```c
xchg           
	1      
```

How does it do on our criteria?

1. Correctness:

2. Fairness:

3. Performance:

Hardware Solution #3:

```c
int CompareAndSwap(int *ptr, int expected, int new)
{
    int actual = *ptr;
    if (actual == expected)
    {
        *ptr = new;  // set
        return actual;
    }
}
```

```c
void lock(lock_t *lock)
{
    while(CompareAndSwap(&lock->flag, 0, 1) == 1) // spin
}
```
Hardware Solution #4:

```c
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```c
void lock(lock_t *lock) // lock struct has turn and ticket as ints {
    int myturn = FetchAndAdd(&lock->ticket);
    while(lock->turn != myturn) // sit n spin
}
```

When a thread wants a lock, it

1. Atomic fetches and adds on ticket value to get “myturn” value
2. Globally shared lock->turn determines whose turn it is
3. When myturn == turn, that thread enters its critical section
4. When done, unlock increments turn

What issue does this fix compared to the previous solutions?

So Much Spinning!!! ... getting dizzy?

What’s a simple solution to a thread spinning when it can’t get the lock it needs?

A better way:

```
sleep
```
Figure 28.9: Lock With Queues, Test-and-set, Yield, And Wakeup

typedef struct
    { int inuse;
      int guard;
      queue_t *q;
    } lock_t;

void lock_init(lock_t *m)
{
    m->inuse = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m)
{
    while (TestAndSet(&m->guard, 1) == 1) ; // acquire guard lock by spinning
    if (m->inuse == 0)
    {
        m->inuse = 1; // lock is mine
        m->guard = 0;
    }
    else
    {
        queue_add(m->q, gettid());
        m->guard = 0;
        // do you see why? HW: fix it 😎
    }
}

void unlock(lock_t *m)
{
    while (TestAndSet(&m->guard, 1) == 1) ; //acquire guard lock by spinning
    if (queue_empty(m->q))
    {
        m->inuse = 0; // let go of lock; no one wants it
    }
    else
    {
        unpark(queue_remove(m->q)); // pass lock (inuse=1) to next thread!
        m->guard = 0;
    }
}