Synchronization Hardware
- Many systems provide hardware support for critical section code
- Uniprocessors
  - could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
- Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
    - Either test memory word and set value
    - Or swap contents of two memory words

- Implementable, but not trivial, in multiprocessor environments
  (atomicity has to be across all CPUs)

In general, solutions to the critical-section problem use the concept of a lock, locks are acquired before entering a critical section, then released on exit. When a process wants to enter a CS, it checks if lock is 0. If so, it sets it to 1 and enters CS. After it is done, it resets it to 0.

Note that this is not a solution, but the same problem. We need mutual exclusion for accesses to the lock variable itself. Three elements of locking:

- Lock Before Using
- Unlock When Done
- Wait(or skip) if locked

Shared boolean variable lock., initialized to false.

```c
    do {
        acquire lock
        critical section
        release lock
        remainder section
    } while (TRUE);
```

Test and Set Instruction (TAS)
Definition:

```c
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

**TAS with shared variable Lock**

do {
    while ( TestAndSet (&lock ))
        ; // do nothing
        // critical section
    lock = FALSE;
        // remainder section
} while (TRUE);

**Swap Instruction**

Definition:

```c
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

Solution:

do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
        // critical section
lock = FALSE;
    // remainder section
} while (TRUE);

**Bounded-waiting Mutual Exclusion with TestAndSet()**

```c

do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // remainder section
} while (TRUE);
```