# İşletim Sistemleri
# Process Senkronizasyonu 2.Kısım

Dr. Öğr. Üyesi Ertan Bütün

Bu dersin içeriği hazırlanırken Operating System Concepts (Silberschatz, Galvin and Gagne) kitabı ve
Prof. Dr. M. Ali Akcayol'un (Gazi Üniversitesi Bilgisayar Mühendisliği Bölümü) ders sunumlarından faydalanılmıştır.

https://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/index.html

http://w3.gazi.edu.tr/~akcayol/BMOS.htm

## Konular

- Process Synchronization
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Monitors
- Alternative Approaches

# Synchronization Hardware

- The Critical-Section (CS) problemi için yazılım tabanlı çözümlerin modern bilgisayar mimarilerde doğru çalışması garanti edilmez.
- Bu bölümde pek çok sistemde bulunan bazı basit donanım komutları sunularak bunların critical section probleminin çözümünde nasıl etkili bir şekilde kullanılabileceği gösterilecektir.
- Donanım özellikleri, herhangi bir programlama görevini kolaylaştırabilir, sistem verimliliğini artırabilir, doğrudan senkronizasyon araçları olarak kullanılabilir veya daha soyut senkronizasyon mekanizmalarının temelini oluşturmak için kullanılabilir.
- CS problemi için verilecek donanım ve yazılım tabanlı çözümler temel olarak kilitleme (locking) tabanlı yaklaşımlardır, locking kullanımıyla kritik bölgelerin korunması sağlanır.

# Synchronization Hardware

- Single processor sistemlerde critical section (CS) problemi şöyle basit bir şekilde çözülebilir:
  - bir process critical secitona girmeden önce interrupt'ları disable eder yani CPUnun preempted olmasına izin verilmez.
  - böylece bir process CS'de iken context switch gerçekleşmez.
- Bu şekilde processin yürütüceği komutların CPU'da sırasıyla (**preemption olmaksızın sırayla, araya başka procesin komutları girmeden)** yürütüleceğinden emin olabiliriz. Böylece paylaşılan değişkende beklenmedik değişiklikler söz konusu değildir.
- Bu yaklaşım nonpreemtive kernel sistemler tarafından benimsenen yaklaşımdır.

# Synchronization Hardware

- **Single processor** sistemler için önerilen critical seciton problemi çözümü **multiprocessor sistemler** için uygun çözüm değildir.
- **multiprocessor sistemlerde** interrupt'ların disable/enable yapılması için tüm işlemcilere mesaj göndermek zaman alıcı olabilir. Bu mesajın iletilmesi, her bir critical sectiona girişi geciktirir ve sistemin verimliliği düşer.
- Bu bölümde, CS probleminin çözmek için destek sağlayan üç donanım komutu sunulacaktır:
  - Memory barriers
  - Hardware instructions
  - Atomic variables

# Synchronization Hardware

- **Memory barriers**
- Hardware instructions
- Atomic variables

# Memory Barrier

- **Memory model**, bir bilgisayar mimarisinin uygulama programlarına yaptığı bellek garantileridir.
- Memory model türleri:
  - **Strongly ordered** – bir işlemcideki bellek değişikliği diğer tüm işlemciler tarafından hemen görülebilir.
  - **Weakly ordered** – bir işlemcide bellekte yapılan değişiklikler diğer işlemciler tarafından hemen görülmeyebilir.
- **Memory barrier'ler** bellekteki herhangi bir değişikliğin diğer tüm işlemcilere yayılmasını (görünür kılınmasını) zorlayan komutlardır.

# Memory Barrier

- Bir **memory barrier** komutu gerçekleştirildiğinde, sistem **sonraki** herhangi bir load veya store işlemi gerçekleştirilmeden önce tüm load ve store işlemlerinin tamamlandığından emin olur.
- Bu nedenle, komutlar yeniden sıralansa bile memory barrier, store işlemlerinin bellekte tamamlanmasını ve gelecekteki load veya store işlemleri gerçekleştirilmeden önce diğer işlemciler tarafından görülmesini sağlar.

# Memory Barrier

- **Örnek:**

**İki thread tarafından paylaşılan veriler**

```
boolean flag = false;
int x = 0;
```

**Thread1**

```
while (!flag)
   ;
print x;
```

**Thread2**

```
x = 100;
flag = true;
```

# Memory Barrier

- Örnekte thread 1 çıkışının 100 olmasını sağlamak için aşağıdaki memory barrier komutları ekleyebiliriz.
- Thread 1

```
while (!flag)
  memory_barrier();
print x
```

- Thread 2

```
x = 100;
memory_barrier();
flag = true
```

- Böylece thread 1 için flag değerinin x değerinden önce load edilmesi ve thread 2 için x = 100 komutunun flag = true komutundan önce gerçekleşmesi garanti edilir.

# Memory Barrier

- Peterson'ın çözümüyle ilgili olarak daha önce gösterilen turn ve flag atamalarının yeniden sıralanması problemini önlemek için giriş bölümündeki flag ve turn atamaları arasına bir memory barrier yerleştirebilir.
- Memory barrier’ler çok düşük seviyeli işlemlerdir ve genellikle yalnızca kernel geliştiriciler tarafından mutual exclusion’ı sağlamak için yazılan özel kodlarda kullanılmaktadır.

# Synchronization Hardware

- Memory barriers
- **Hardware instructions**
- Atomic variables

# Hardware Instructions

- Process senkronizasyonu için diğer bir donanım mekanizması: yürütülmesi sırasında interrupt olmayan **atomic** olarak yürütülen özel donanım komutları (**hardware instructions**) kullanmaktır.
- Multiprocess sistemlerde bu özel komutlar aynı andan yürütülmeye çalışıldığında bunlar sırasıyla yürütülür, birinin işi tamamlanmadan araya başkası giremez.

# Hardware Instructions

- Bir word (2 byte -16bitlik veri) içeriğini test edip değiştirme veya iki ayrı word içeriğini yer değiştirme (swap) işlemlerini **atomik** **(interrupta müsaade edilmeden tek bir seferde yürütülen birim)** olarak yapan özel donanım komutları, **critical section problemini basit bir şekilde çözmek için kullanılabilir.**
- Belirli bir makine için belirli donanımsal komut isimleri yerine **test_and_set()** ve **compare_and_swap()** komutları soyut komut isimleri olarak bu bölümde kullanılmıştır.
  - Bu iki soyut komut zaten çoğu mimaride varolan mimariden mimariye isimleri değişebilen ancak yaptıkları iş itibariyle mantıksal olarak bu soyut komutlarla aynı işi yapan komutları temsil etmektedir.
  - Bu iki soyut komut, critical section probleminin donanım desteği ile nasıl çözülebileceğini göstermek için kullanılacaktır.

# Critical-section Probleminin Locks Kullanımıyla Genel Çözümü

- Pi processi acquire lock ile kilidi aldığında critical sectionına girebilecek,
- kilit Pi processinde olduğu için Pj processi critical secitona giremeyecek,
- acquire lock işlemi atomik olarak yürütüldüğü için aynı anda Pj processi de bu komutu çalıştırmak istediğinde Pi ya da Pj den biri önce yürütülecek diğeri sonra yürütülecek.
- bu atomic donanım komutları sayesinde belli bir anda sadece bir process kilidi alabilecek.

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

# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

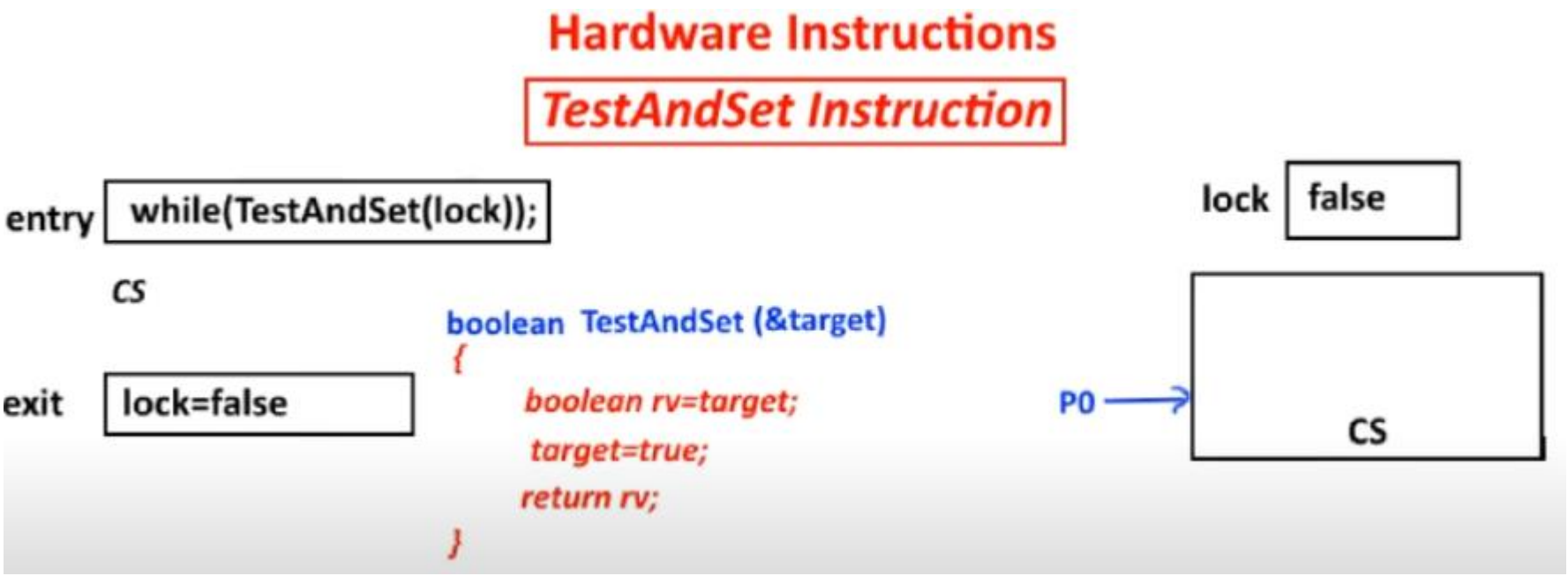
- Boolean türündeki **lock** değişkeni processler arasında paylaşılır, başlangıç değeri FALSE olarak atanır.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

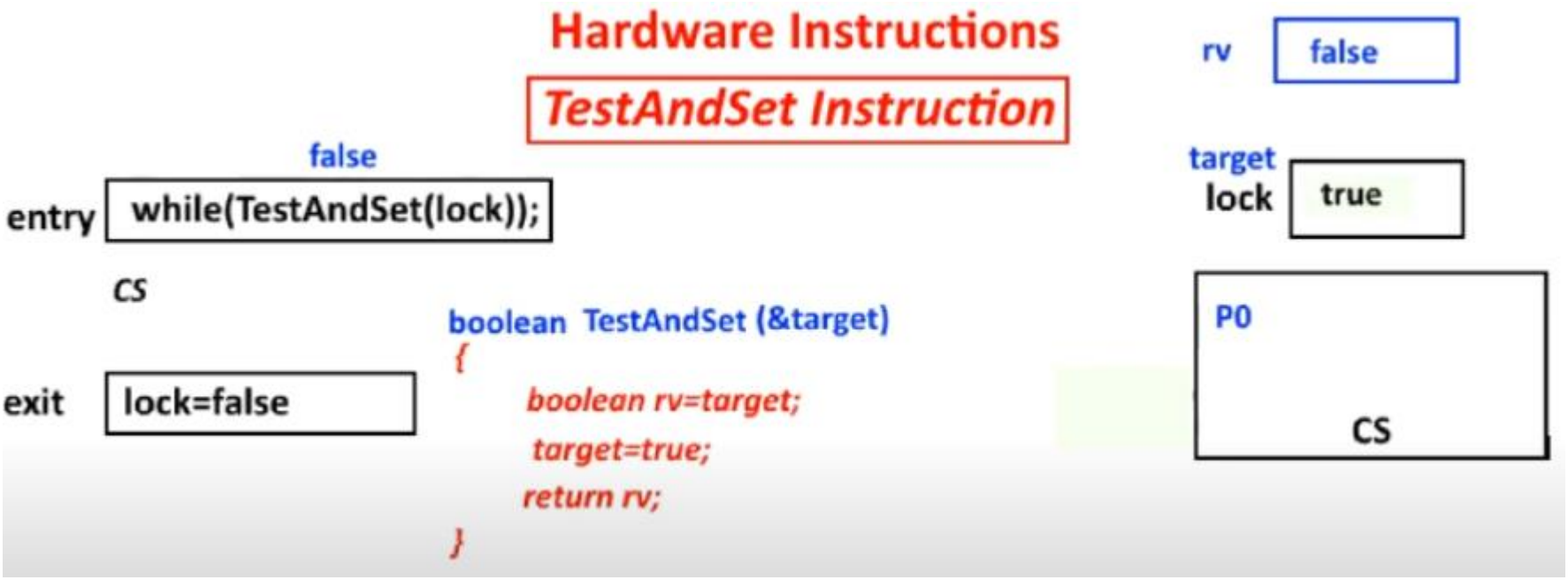
# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Mutual exclusion sağlanıyor mu?**
- P0 CS'ye girmek istesin



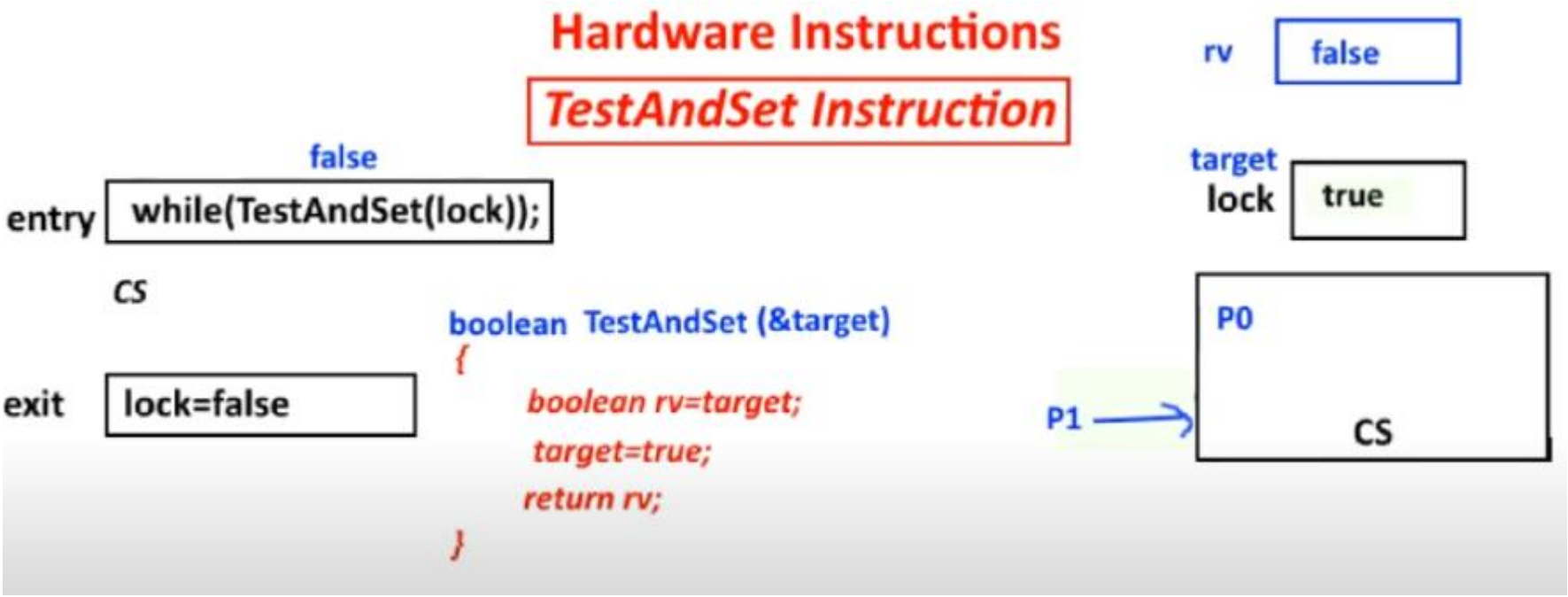
# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Mutual exclusion sağlanıyor mu?**



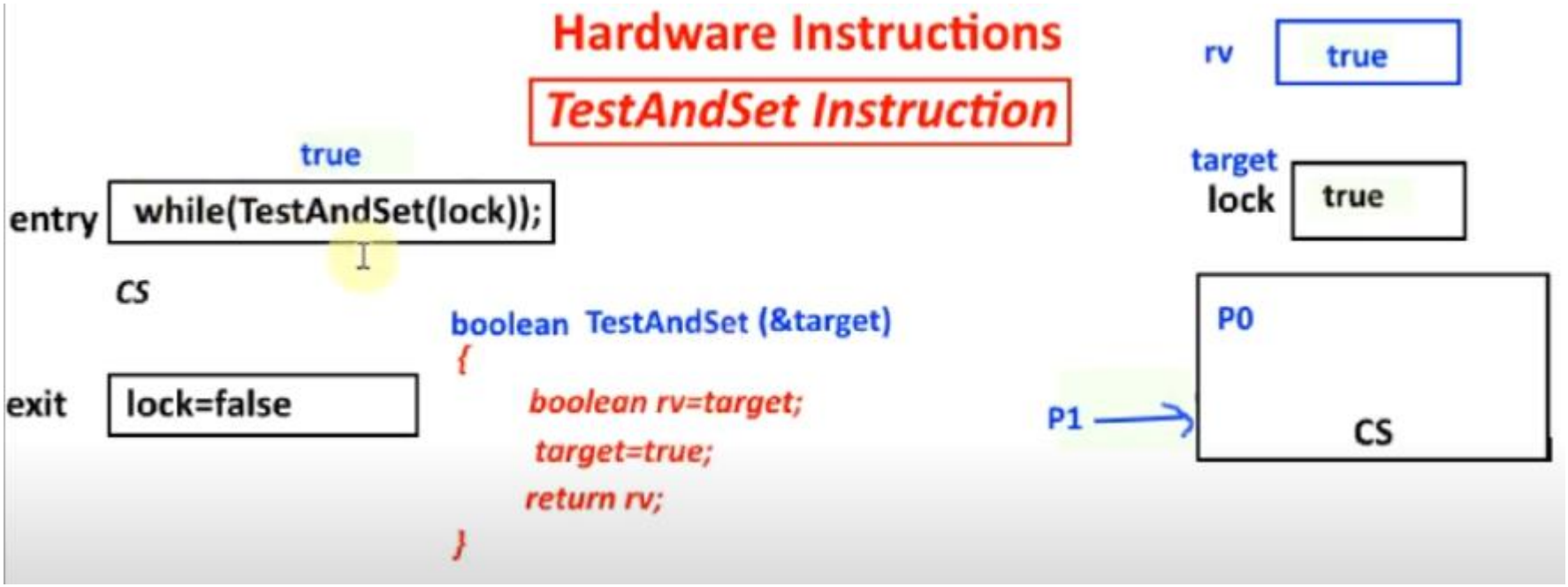
# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Mutual exclusion sağlanıyor mu?**



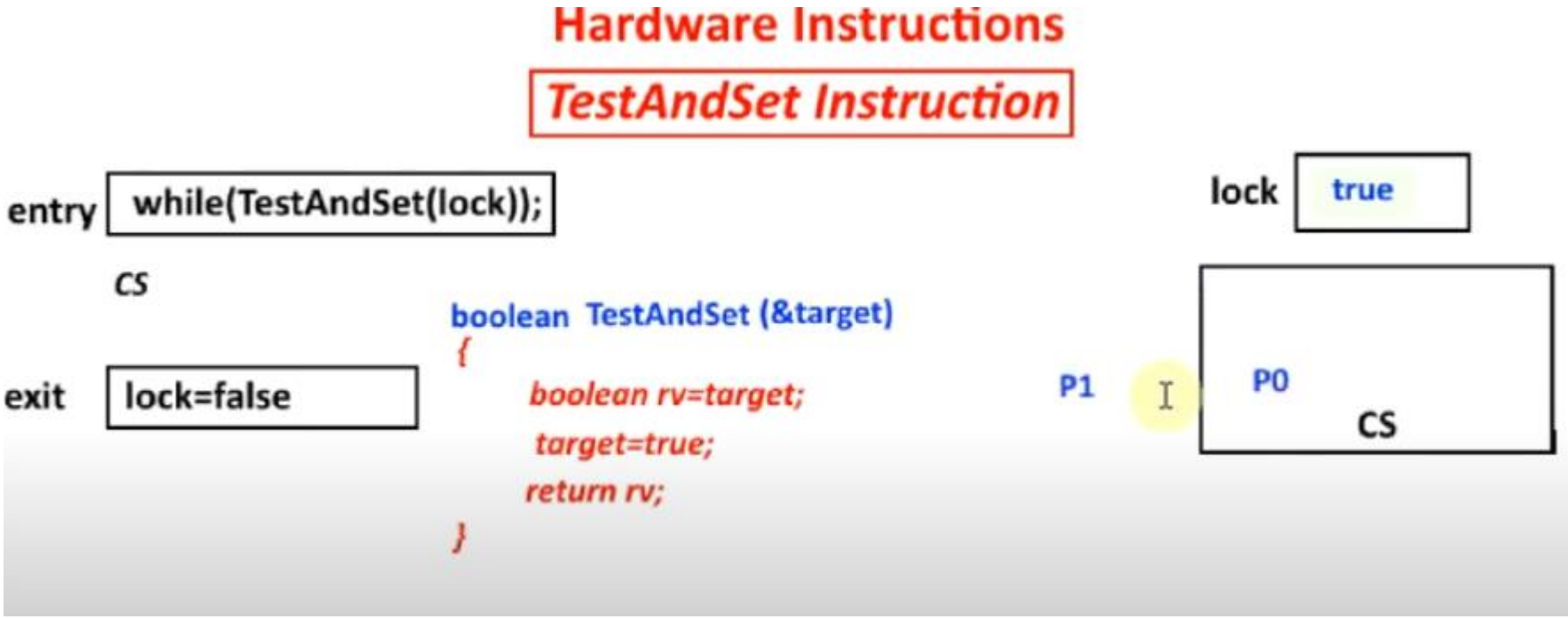
# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Mutual exclusion sağlanıyor mu?**



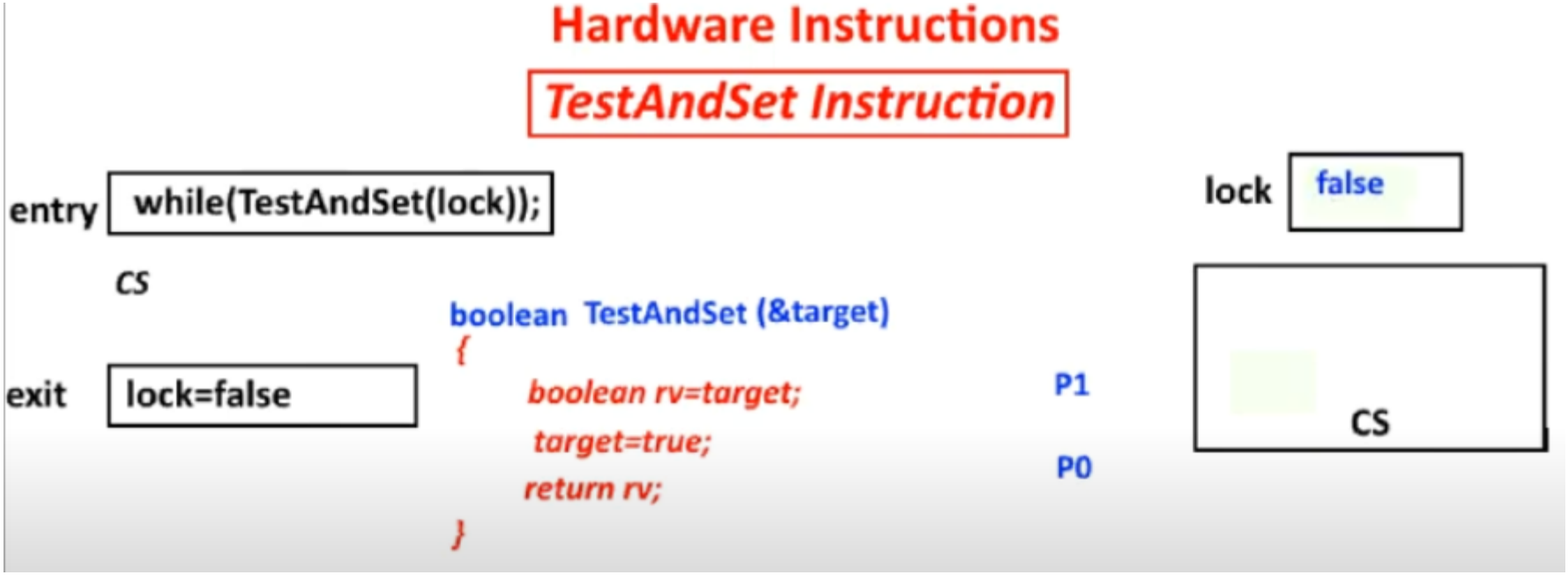
# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Progress sağlanıyor mu?**



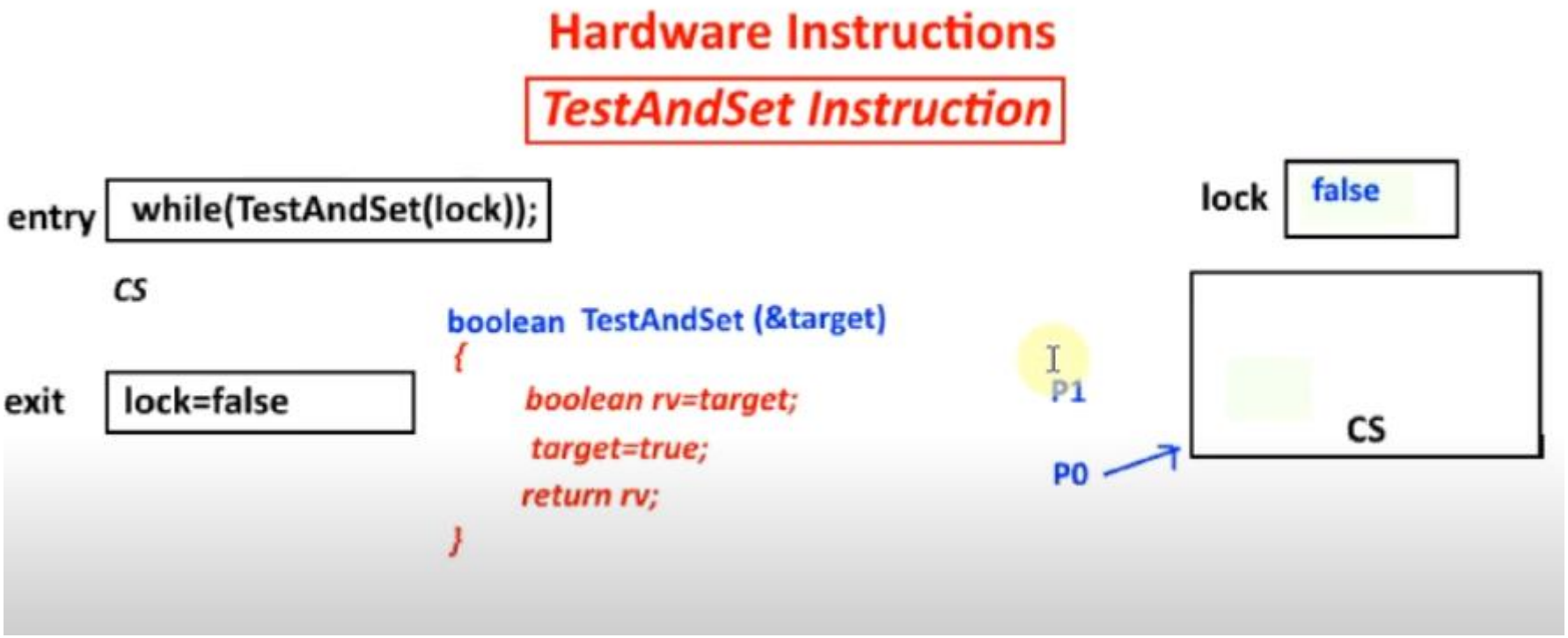
# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Progress sağlanıyor mu?**



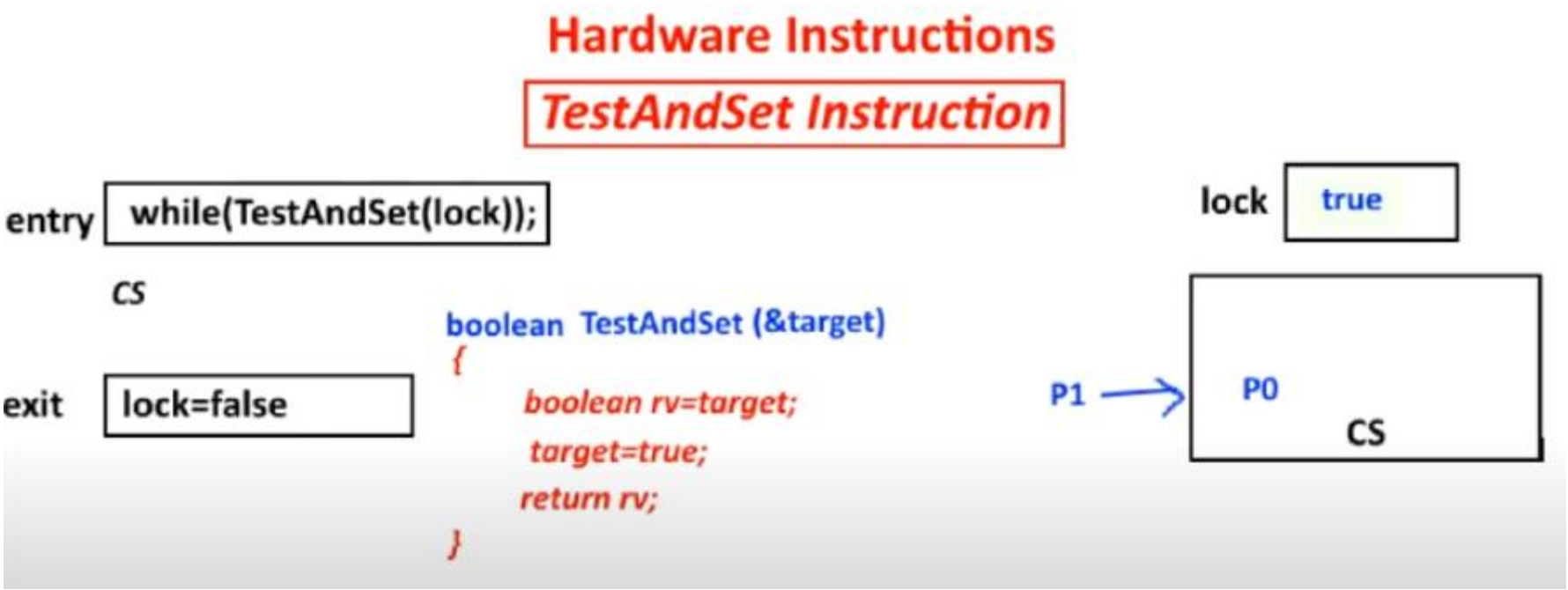
# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Progress sağlanıyor mu?**



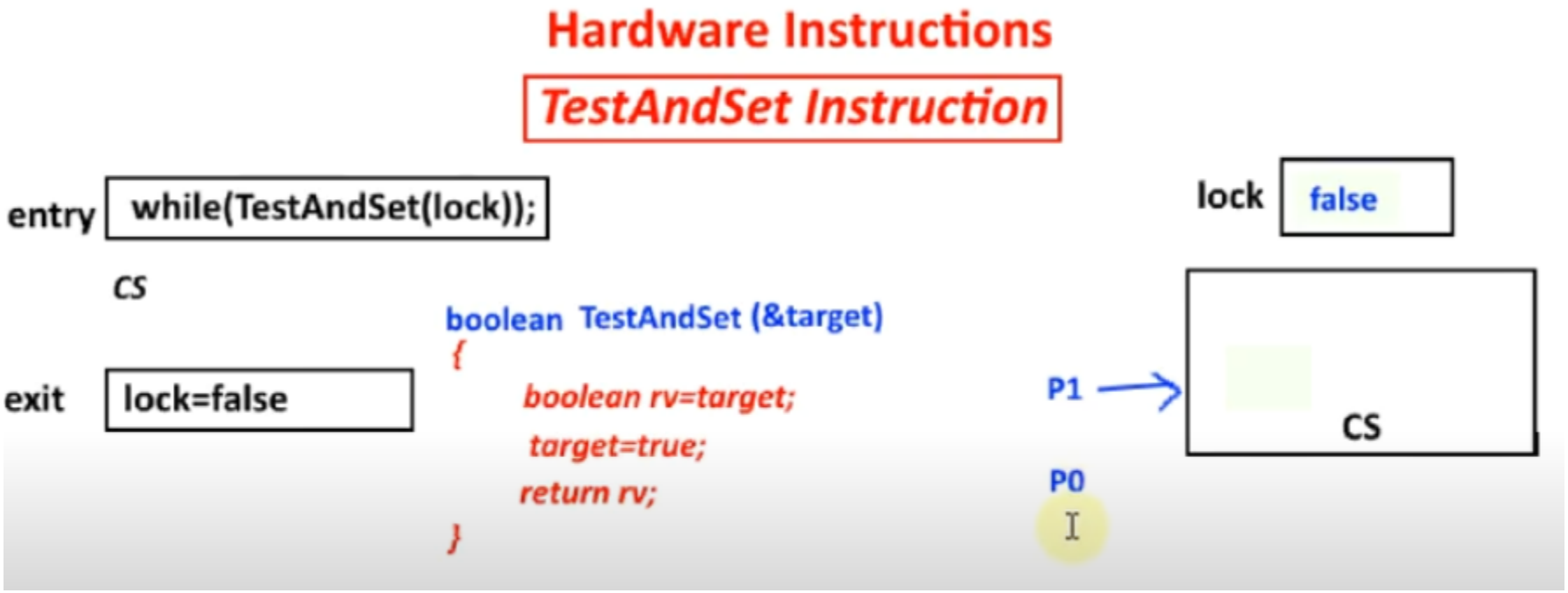
# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Bounded waiting sağlanıyor mu?**



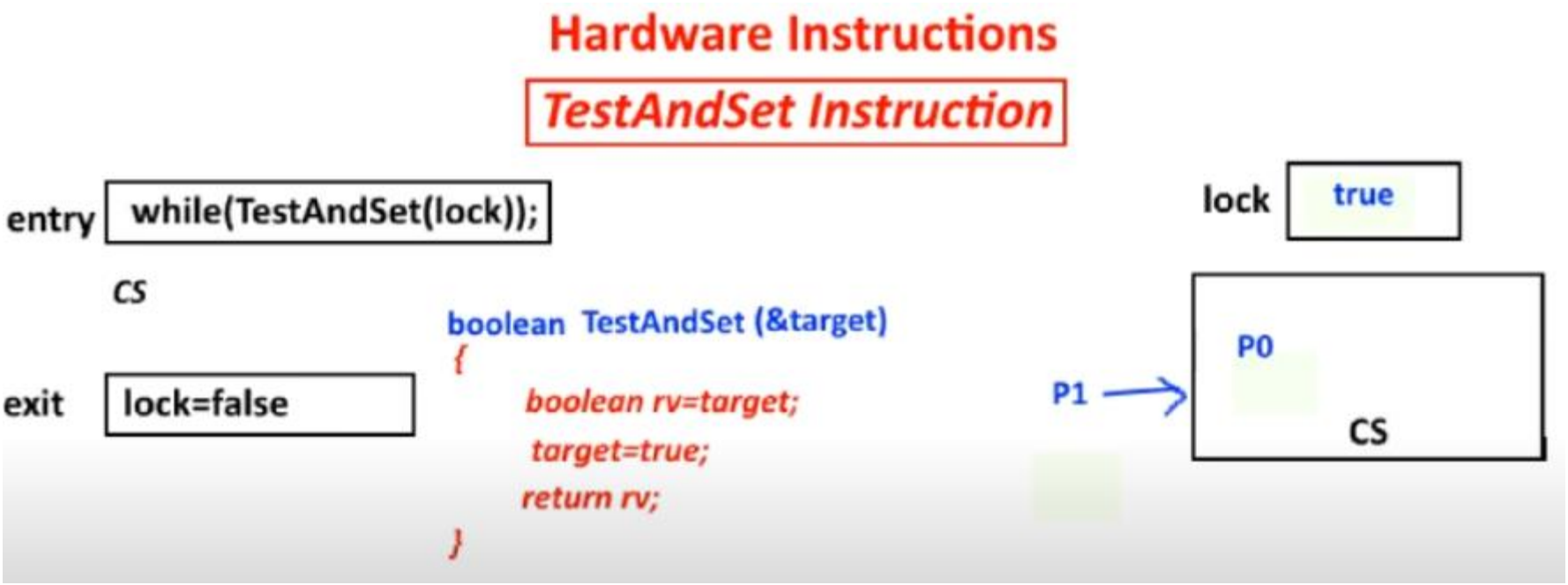
# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Bounded waiting sağlanıyor mu?**



# test_and_set() Kullanımıyla Mutual Exclusion Gerçekleştirimi

- **Bounded waiting sağlanıyor mu?**



# compare_and_swap()

- **compare_and_swap()** komutu şöyle tanımlanabilir:

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

- test_and_set() komutu gibi **atomik** olarak yürütülür.
- Her zaman **value** değişkeninin orijinal değerini döndürür.
- **value** değişkenin değeri **expected** değerine eşit olunca **value** değerine **new_value** atanarak swap yapılır.

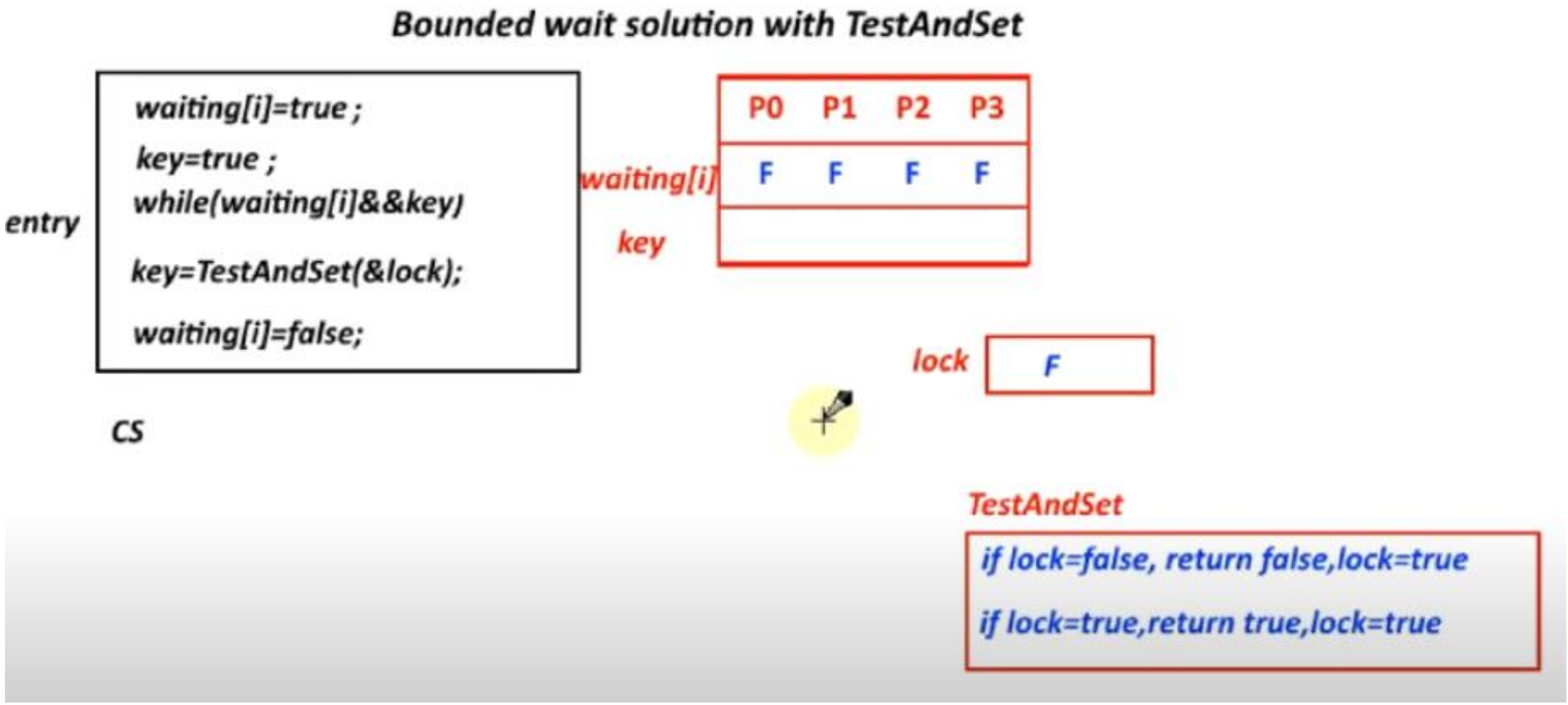
# compare_and_swap() Kullanımıyla Mutual Exclusion Gerçekleştirimi

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

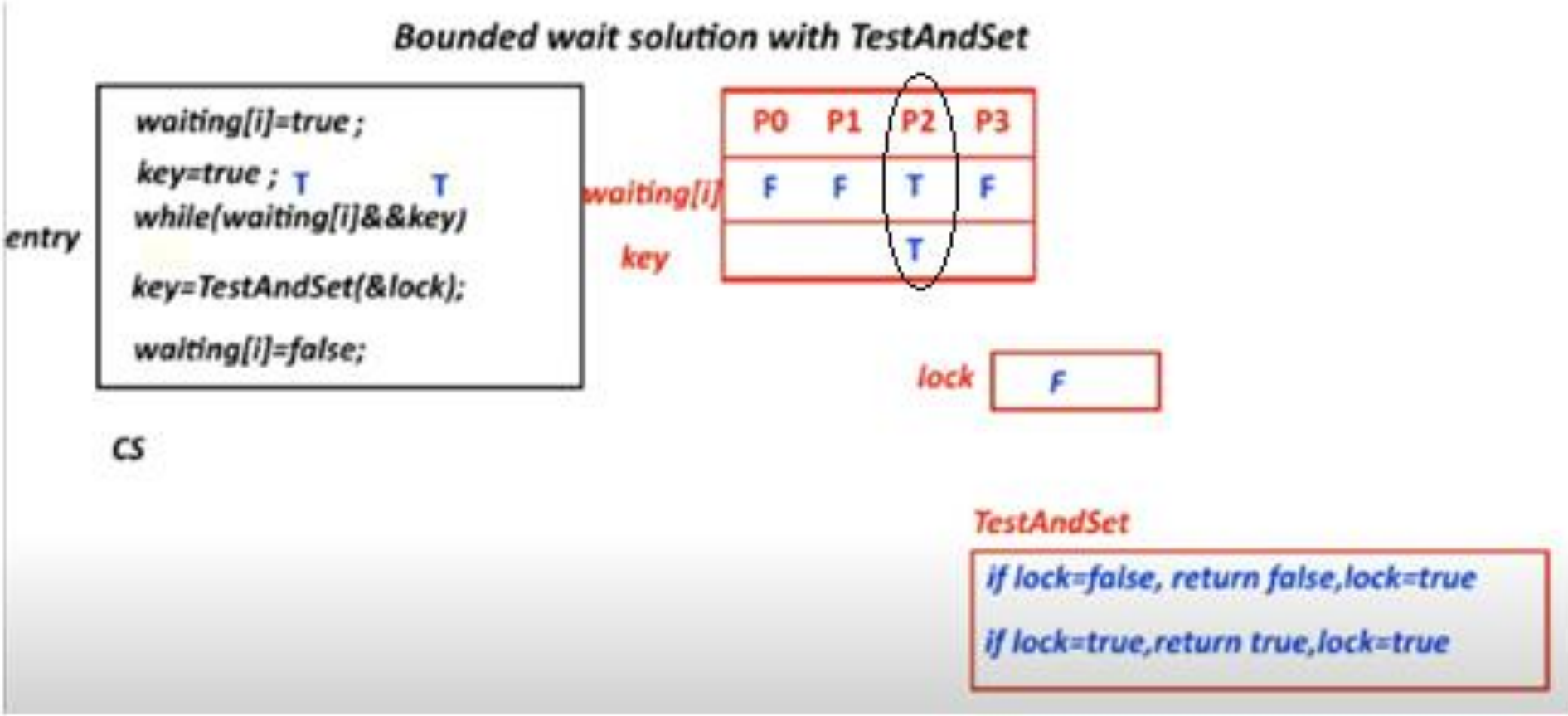
# test_and_set Kullanımıyla Bounded-waiting Özelliğinin Sağlandığı Bir Çözüm

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&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);
```

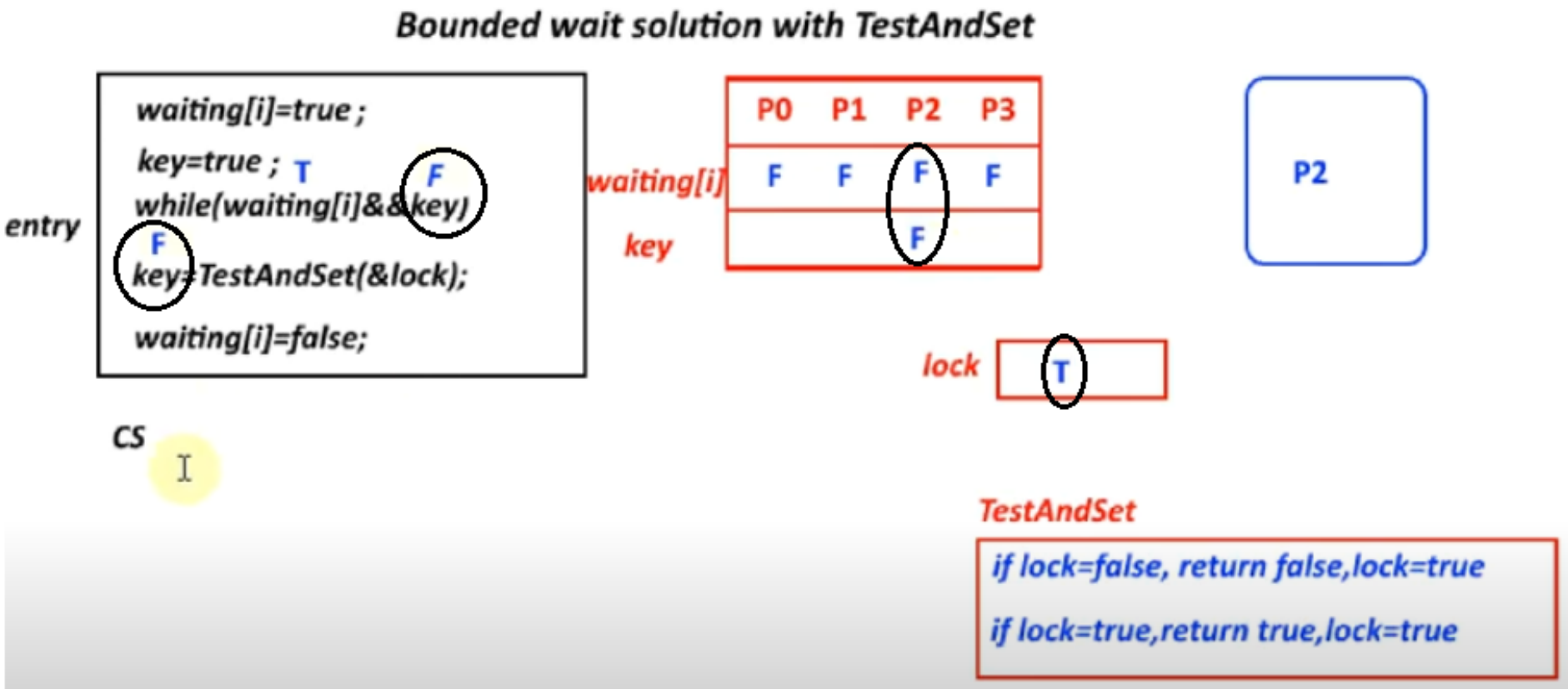
# Example - Bounded-waiting Mutual Exclusion with test_and_set



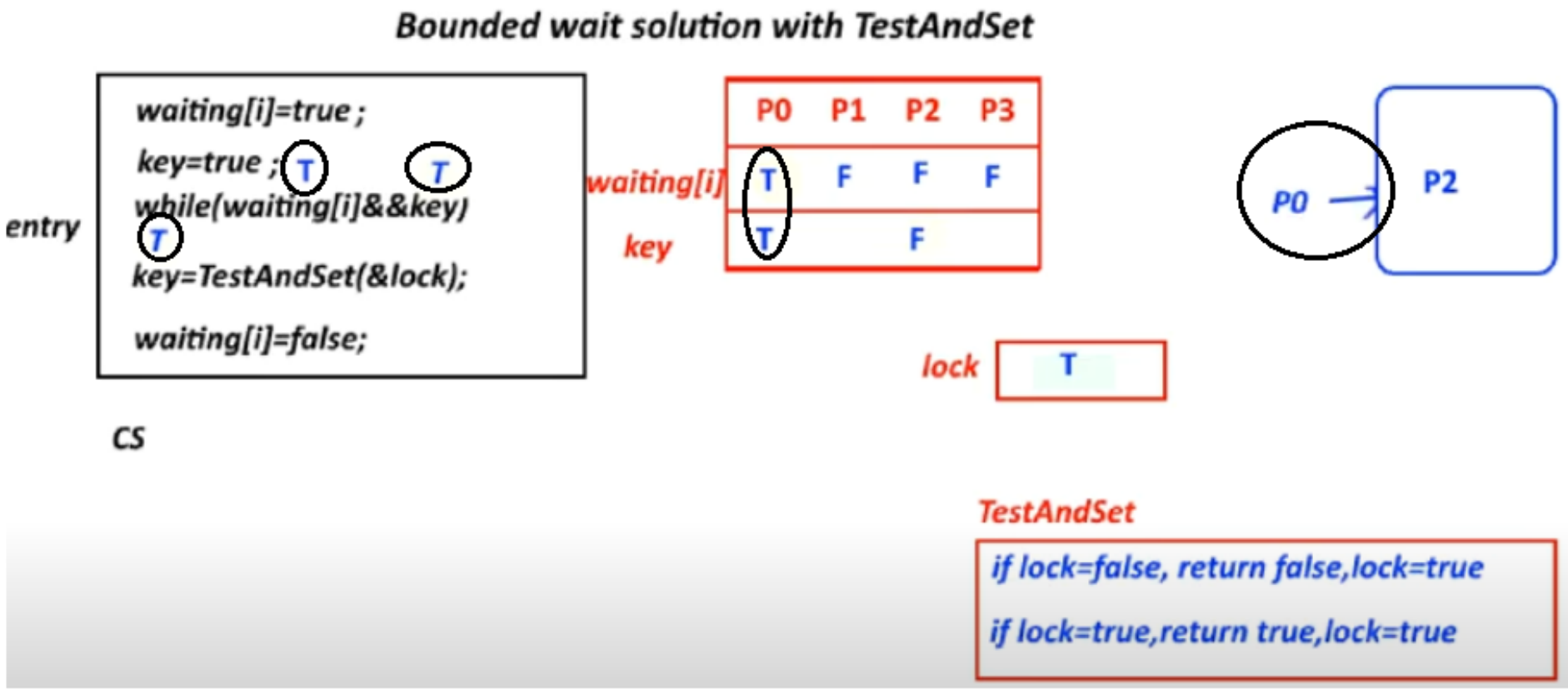
# Example - Bounded-waiting Mutual Exclusion with test_and_set



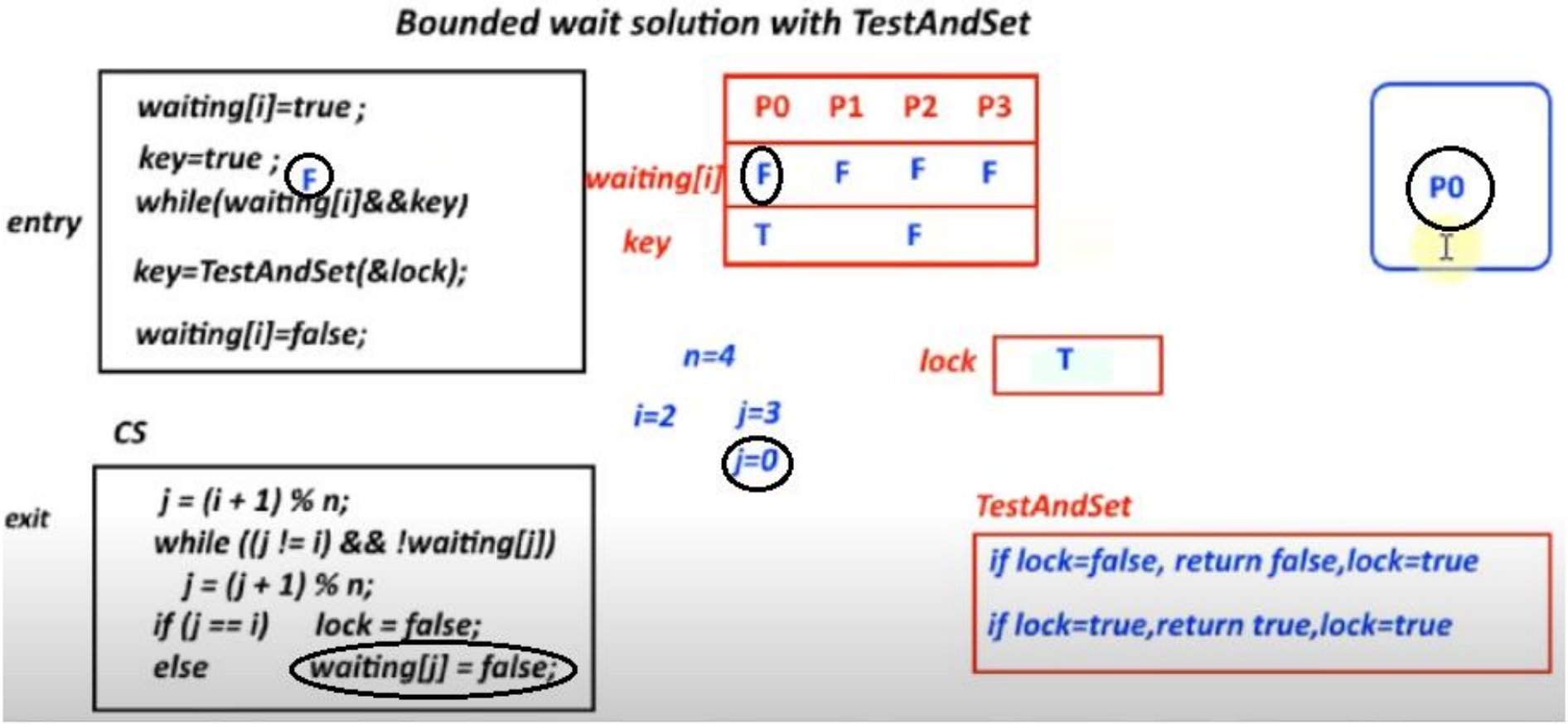
# Example - Bounded-waiting Mutual Exclusion with test_and_set



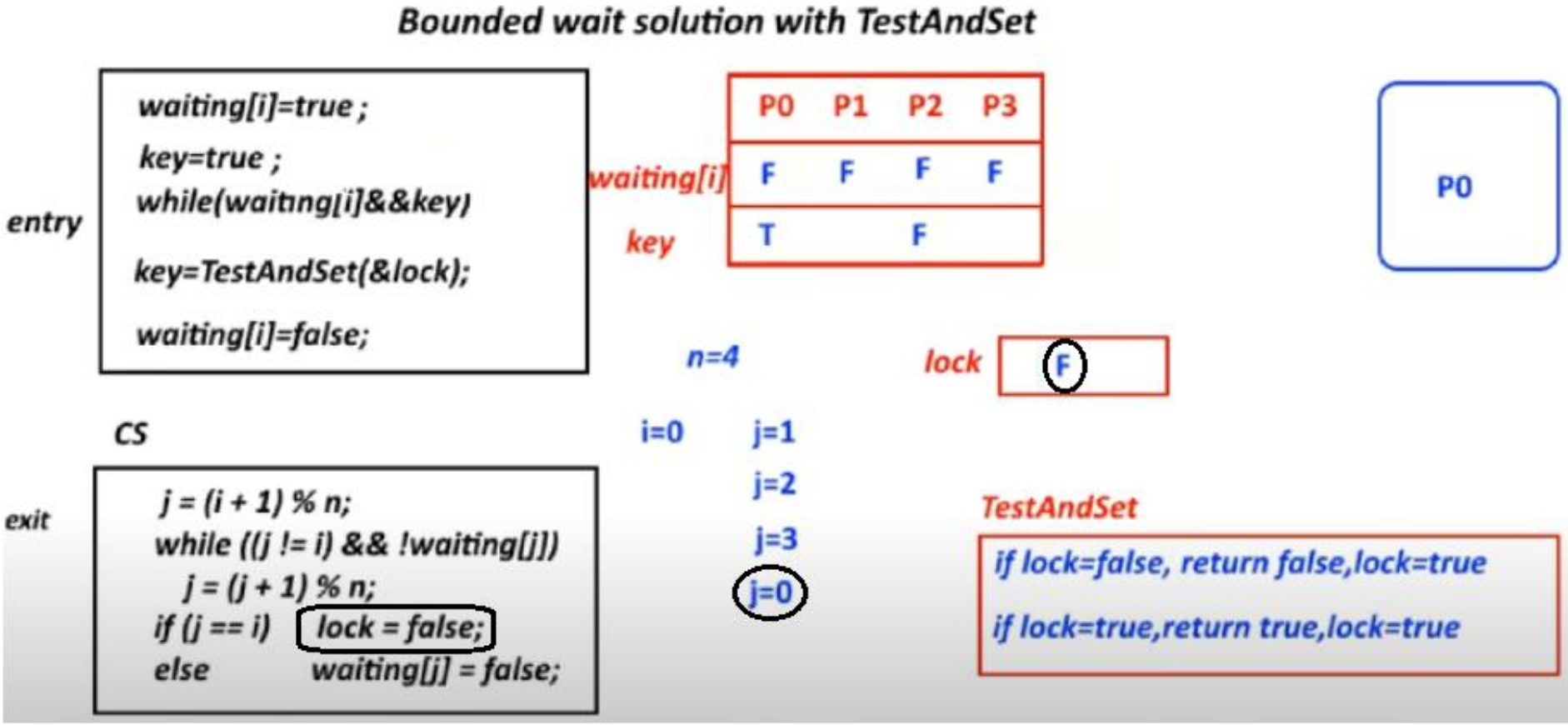
# Example - Bounded-waiting Mutual Exclusion with test_and_set



# Example - Bounded-waiting Mutual Exclusion with test_and_set



# Example - Bounded-waiting Mutual Exclusion with test_and_set



# Synchronization Hardware

- Memory barriers
- Hardware instructions
- **Atomic variables**

# Atomic Variables

- Tipik olarak, **compare-and-swap()** komutu, mutual exclusion sağlamak için doğrudan kullanılmaz. Bunun yerine, CS problemini çözen diğer araçları oluşturmak için temel bir yapı taşı olarak kullanılır.
- Diğer bir donanımsal araç atomic variables'dır, integer ve boolean gibi temel veri türleri üzerinde atomik işlemler sağlarlar.
- Daha önce race condition için verilen örnekte counter değişkeninde olduğu gibi bir integer'in artırılması ve azaltılması race condition'a neden olabiliyordu.
- Tek bir değişken güncellenirken mutual exclusion'ı sağlamak için atomic variables kullanılabilir.

# Atomic Variables

- Atomik variables'ı destekleyen çoğu sistem, atomik değişkenlere erişmek ve bunları işlemek için fonksiyonların yanı sıra özel atomik veri türleri sağlar.
- Bu fonksiyonlar genellikle compare-and-swap() işlemleri kullanılarak uygulanır. Örnek olarak, aşağıdaki increment fonksiyonu atomik variable olan sequence değişkenini atomik olarak artırır:

```
increment(&sequence);
```

# Atomic Variables

- increment() fonksiyonu compare-and-swap() kullanılarak aşağıdaki gibi gerçekleştirilebilir:

```
void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v,temp,temp+1));
}
```

# Atomic Variables

- Atomik variables, atomik güncellemeler sağlamasına rağmen her koşulda race condition'ı tamamen çözemezler.
- Bounded buffer probleminde, count için bir atomik tamsayı kullanabiliriz. Bu, count güncellemelerin atomik olmasını sağlayacaktır.
  - Örneğin buffer boş olsun iki consumer while döngüsünde beklesin,
  - bir producer buffer'e bir eleman eklerse her iki consumer de while döngüsünden çıkabilir,
  - count 1 olmasına rağmen her ikisi de tüketmeye çalışabilir.