İşletim Sistemleri
Process Senkronizasyonu

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

Process Synchronization

- Cooperating process'ler diğer process'leri etkilerler veya diğer process'lerden etkilenirler.
- Cooperating process'ler paylaşılmış hafıza alanıyla veya dosya sistemleri ile veri paylaşımı yaparlar.
- **Paylaşılmış veriye eşzamanlı erişim tutarsızlık problemlerine yol açabilir.**
- Paylaşılmış veri üzerinde işlem yapan process'ler arasında veriye erişimin yönetilmesi gereklidir.

Process Synchronization

- producer-consumer problemini hatırlayalım

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

Yeni eleman eklendi.

Bir eleman alındı.

- **`counter`** değişkeninin değeri buffer'a yeni eleman eklendiğinde artmakta, eleman alındığında azalmaktadır.

Process Synchronization

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

- **producer ve consumer kısımları farklı zamanlarda doğru çalışsa da eşzamanlı doğru çalışamayabilirler.**
- **`counter=5`** iken **`counter++`** ve **`counter--`** deyimlerinin aynı anda çalıştığını düşünelim.
- **Farklı zaman aralıklarında çalışmış olsalardı `counter=5` olacaktı.**

Process Synchronization

- `counter++` için makine komutları aşağıdaki gibi olabilir.

$$
\begin{aligned}
register_1 &= \texttt{counter} \\
register_1 &= register_1 + 1 \\
\texttt{counter} &= register_1
\end{aligned}
$$

- `counter--` için makine komutları aşağıdaki gibi olabilir.

$$
\begin{aligned}
register_2 &= \texttt{counter} \\
register_2 &= register_2 - 1 \\
\texttt{counter} &= register_2
\end{aligned}
$$

- $register_1$ ve $register_2$ aynı (AC) veya farklı register olabilir.

Process Synchronization

- `counter++` ve `counter--` işlemlerinin eşzamanlı olarak yürütülmesi (concurrent execution), önceki slaytta sunulan düşük düzeyli ifadelerin rasgele sırayla araya eklendiği sıralı bir yürütmeye eşdeğerdir.

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

- Yukarıdaki sırada buffer'daki eleman sayısı 4 olarak görülür, ancak gerçekte buffer'daki eleman 5 tanedir.
- T_4 ile T_5 yer değiştirirse buffer'daki eleman sayısı 6 olarak görülecektir.
- İki process `counter` değişkeni üzerinde eş zamanlı(concurrently) işlem yaptığından sonuç yanlış olmaktadır.

Process Synchronization

- Birden çok processin aynı verilere eşzamanlı olarak erişip değiştirdiği ve bu veriler üzerinde hesaplanacak sonucun erişimin gerçekleştiği sıraya bağlı olduğu duruma race condition denir.
- Önceki slayttaki race condition durumuna düşmemek için, bir seferde yalnızca bir processin counter değişkenini değiştirebildiğinden emin olmamız gerekir. **Böyle bir garanti verebilmek için processler senkronize edilmelidir**. (process synchronization)
- Verilen örnekteki gibi durumlar işletim sistemlerinde sıklıkla meydana gelir. Multicore sistemlerin getirdiği avantajlarla multithreaded uygulamalar geliştirmek önemli hale gelmiştir. Bu tür uygulamalarda veri paylaşan birkaç threadin farklı işlemci çekirdeklerinde paralel çalışması kuvvetle muhtemeldir.
- Bu tür işlemlerden kaynaklanan herhangi bir değişiklik birbirini etkilememelidir.

Konular

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

The Critical-Section Problem

- Bu bölümde critical section problemi tartışılarak **process senkronizasyonu** ele alınacaktır.
- Bir sistem, n tane process'e $\{P_0, P_1, \ldots, P_{n-1}\}$ sahip olsun.
- Her processin , en az bir başka processle paylaşılan verilere erişebildiği ve bunları güncellediği critical section adı verilen bir kod bölümü vardır.

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

The Critical-Section Problem

- Sistemin önemli özelliği, bir process kendi critical sectionında yürütülürken diğer processlerin critical sectionında **yürütülmesine izin verilmemesidir.**
- **Yani aynı anda iki process** critical sectionında **çalıştırmamalıdır.**
- Critical-section problemi, **processlerin paylaştıkları verileri senkronize bir şekilde kullanabilecekleri bir protokol tasarlamaktır.**

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

The Critical-Section Problem

- Kullanılan protokoller ile her process critical sectiona girmek için izin istemektedir.
- İzin için kullanılacak kod bölümüne entry section denir, critical sectionın ardından exit section gelebilir, kalan koda remainder section denir.
- P_i processinin genel yapısı:

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Algorithm for Process Pi

```
do {

        while (turn == j);

                critical section

        turn = j;

                remainder section

} while (true);
```

Solution to Critical-Section Problem

- The Critical-Section probleminin çözümü aşağıdaki üç gereksinimi sağlamak zorundadır:
  - Mutual exclusion (karşılıklı dışlama): Bir P_i process'i critical sectionda yürütülüyorsa diğer processlerin hiçbiri critical sectionda olamaz.
  - Progress (ilerleme): Hiçbir process critical sectionda (CS) çalışmıyorsa ve birden fazla process critical sectiona girmek istiyorsa bir sonraki CS'ye girecek pocessin seçimi süresiz olarak ertelenemez. Seçme işi yalnızca reminder sectionda olmayan processler (entry ya da exit sectionda olan processler) tarafından belirlenir. *
  - Bounded waiting (sınırlı bekleme): Bir process CS'ye girmek için istekte bulunduktan sonra diğer processlerin bu processten önce CS'yi kaç kez ele geçirebileceği sayında bir limit olmalıdır. İstekte bulunan process CS'ye girmek için en fazla bu limit kadar beklemelidir, daha sonra CS'ye girebilmelidir.
- The Critical-Section Probleminin çözümünde
  - Her processin sıfırdan farklı bir hızda yürütüldüğü varsayılmıştır.
  - Bununla birlikte, n tane processin göreceli hızına ilişkin hiçbir varsayımda bulunulmamıştır.*

Critical-Section Handling in OS

- Critical section yönetimi için işletim sistemleri açısından iki yaklaşım vardır:

1. Preemptive kernel: Bir process kernel modda çalışırken yüksek öncelikli başka bir process nedeniyle yürütülmesi önlenebilir(askıya alınabilir).
    - **Race condition** söz konusu olabilir.
    - Paylaşılan kernel verilerinin **race conditiondan** arınmasını sağlamak için dikkatlice tasarlanmaları gerekir.
2. Nonpreemptive kernel: Bir process kernel modda çalışırken yürütülmesinin önlenmesine/askıya alınmasına izin verilmez.
    - Kernel modundaki bir process, kernel modundan çıkana kadar veya bloke oluncaya kadar veya istemli olarak CPU'nun kontrolünü verene kadar çalışacaktır.
    - Kernelde aynı anda yalnızca bir process aktif olduğundan **race condition** söz konusu değildir.

Konular

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

Algorithm Peterson's Solution

- Peterson's solution algoritması şekilde verilmiştir.

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);

            critical section

        flag[i] = false;

            remainder section

} while (true);
```

```
int turn;
Boolean flag[2]
```

global değişkenler

entry section

exit section

Peterson's Solution

- P_0 ve P_1 process'leri için critical section çözümü:

P0	P1
flag[0]=T turn=1 while(flag[1]&& turn==1) ; cs	flag[1]=T turn=0 while(flag[0]&& turn==0) cs

flag[0]	flag[1]	turn
F	F	
T	F	
F	T	
T	T	0
T	T	1

Peterson's Solution

- P_0 ve P_1 process'leri için critical section çözümü:

P0	P1
flag[0]=T turn=1 while(flag[1]&& turn==1) ; cs	flag[1]=T turn=0 while(flag[0]&& turn==0) cs

flag[0]	flag[1]	turn
F	F	
T	F	
F	T	
T	T	0
T	T	1

Peterson's Solution

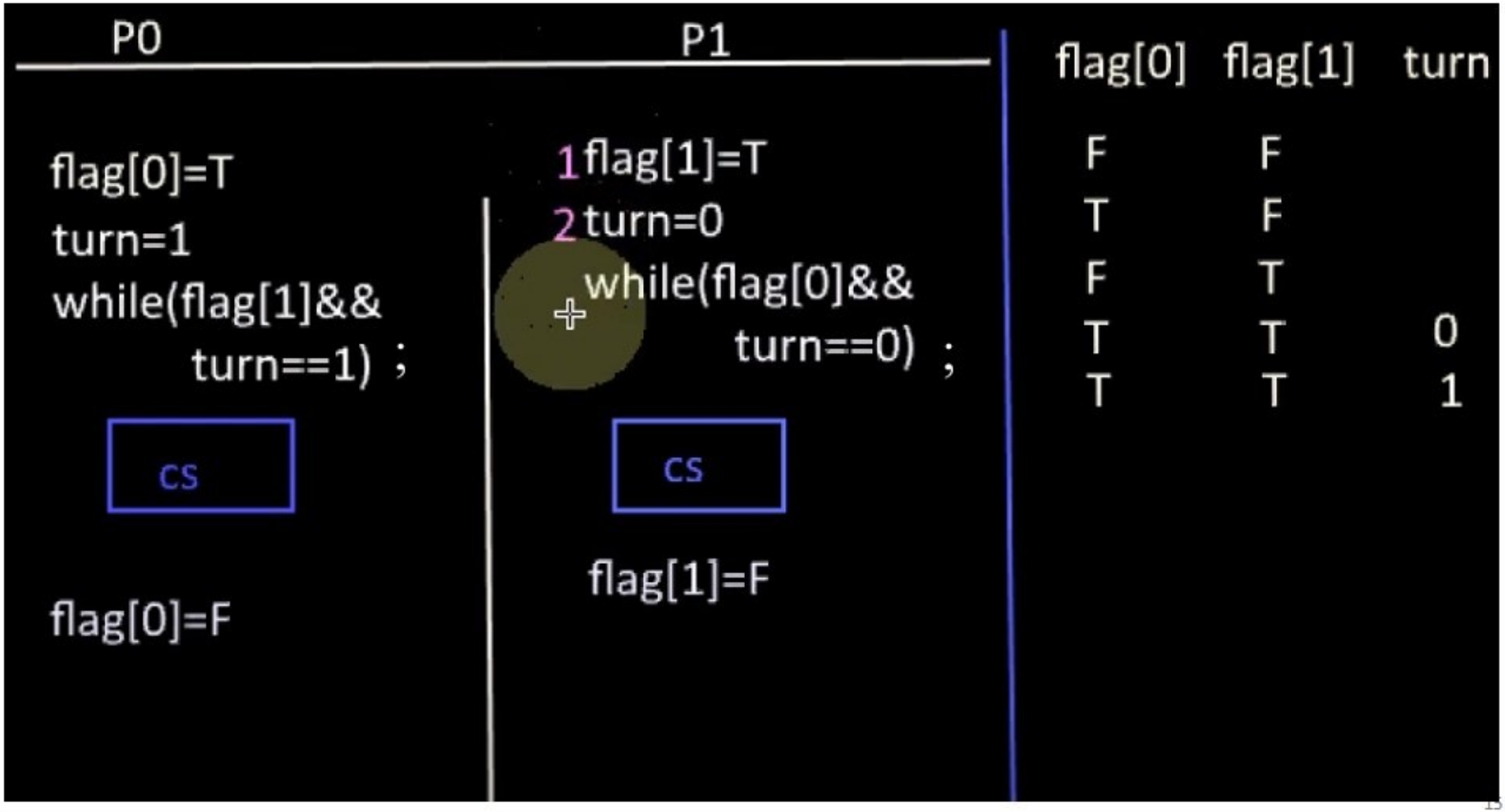
- P_0 ve P_1 process'leri için critical section çözümü:

P0	P1
flag[0]=T	flag[1]=T
turn=1	turn=0
while(flag[1]&& turn==1) ;	while(flag[0]&& turn==0) ;
CS	CS
flag[0]=F	flag[1]=F

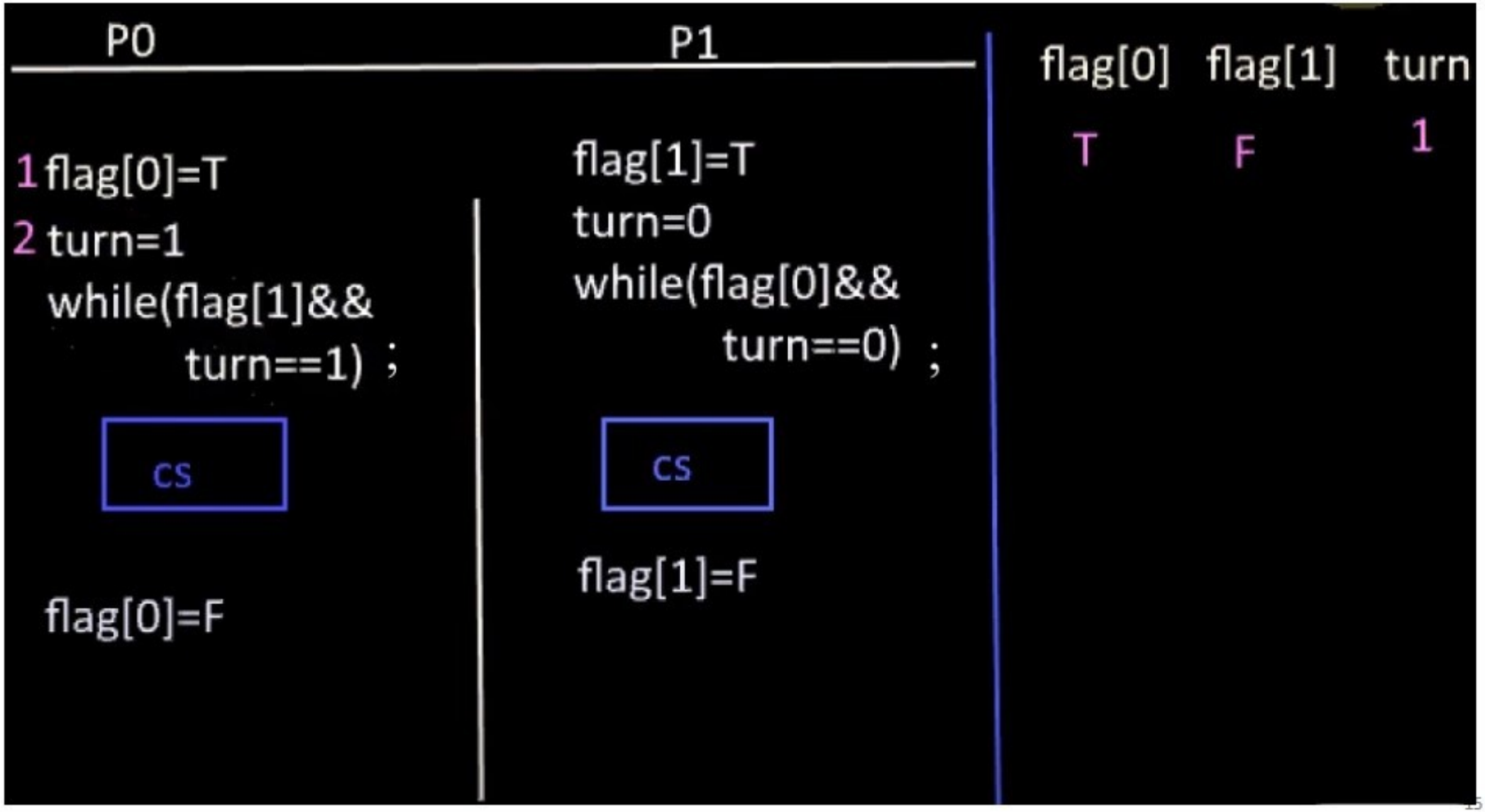
flag[0]	flag[1]	turn
F	F	
T	F	
F	T	
T	T	0
T	T	1

Peterson's Solution

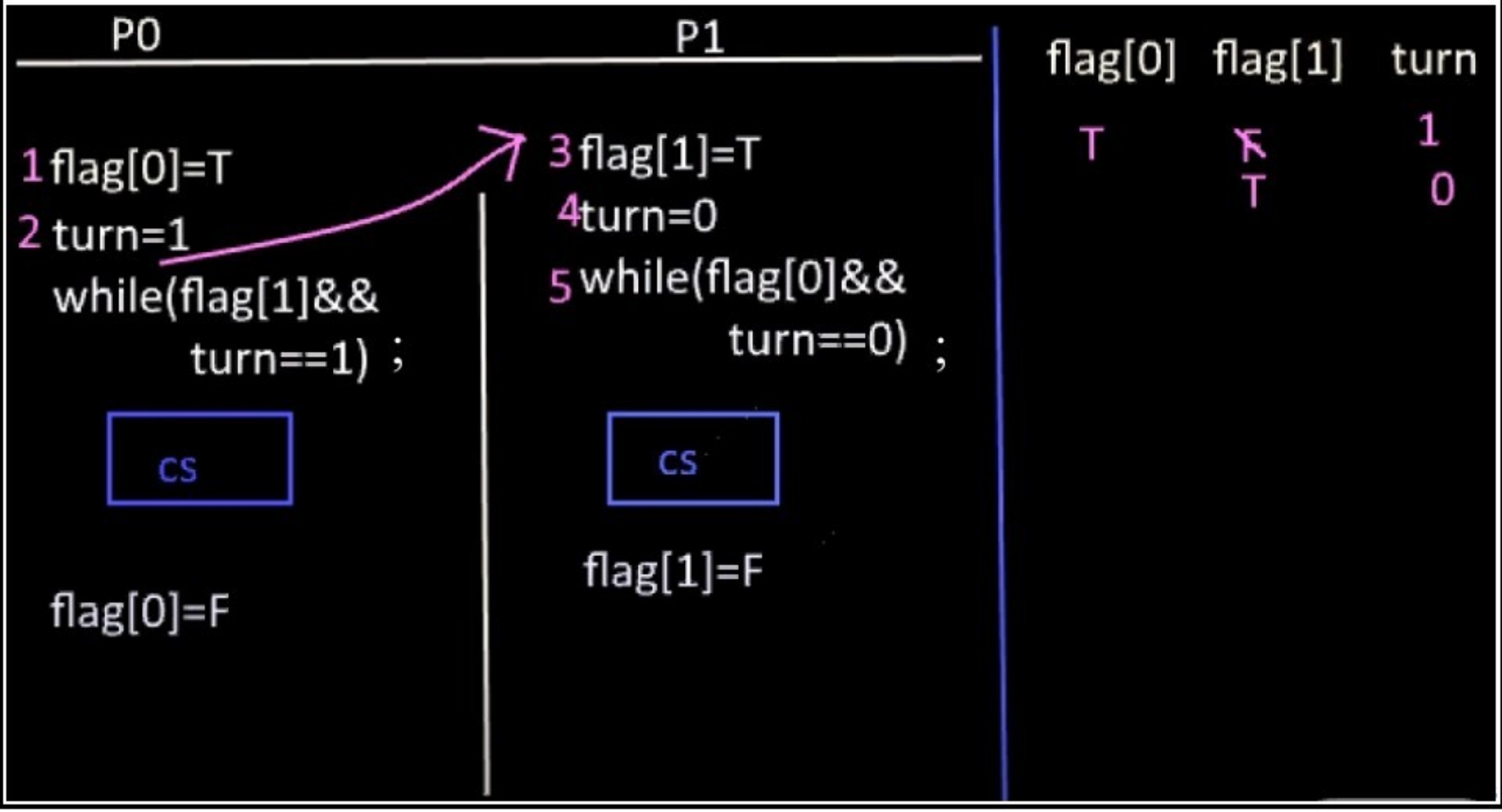
- P_0 ve P_1 process'leri için critical section çözümü:



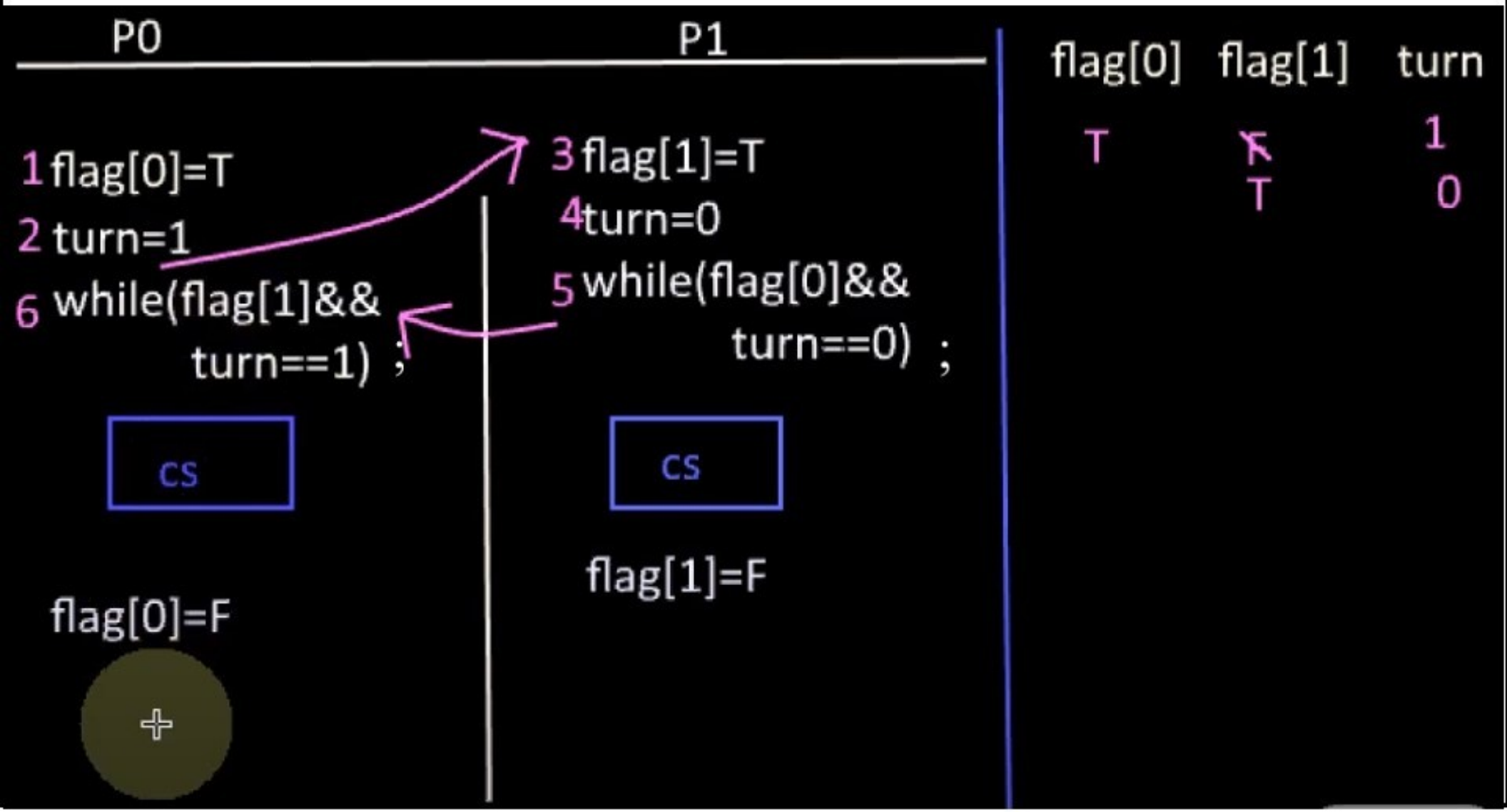
Örnek Çalıştırma 1- Peterson's Solution



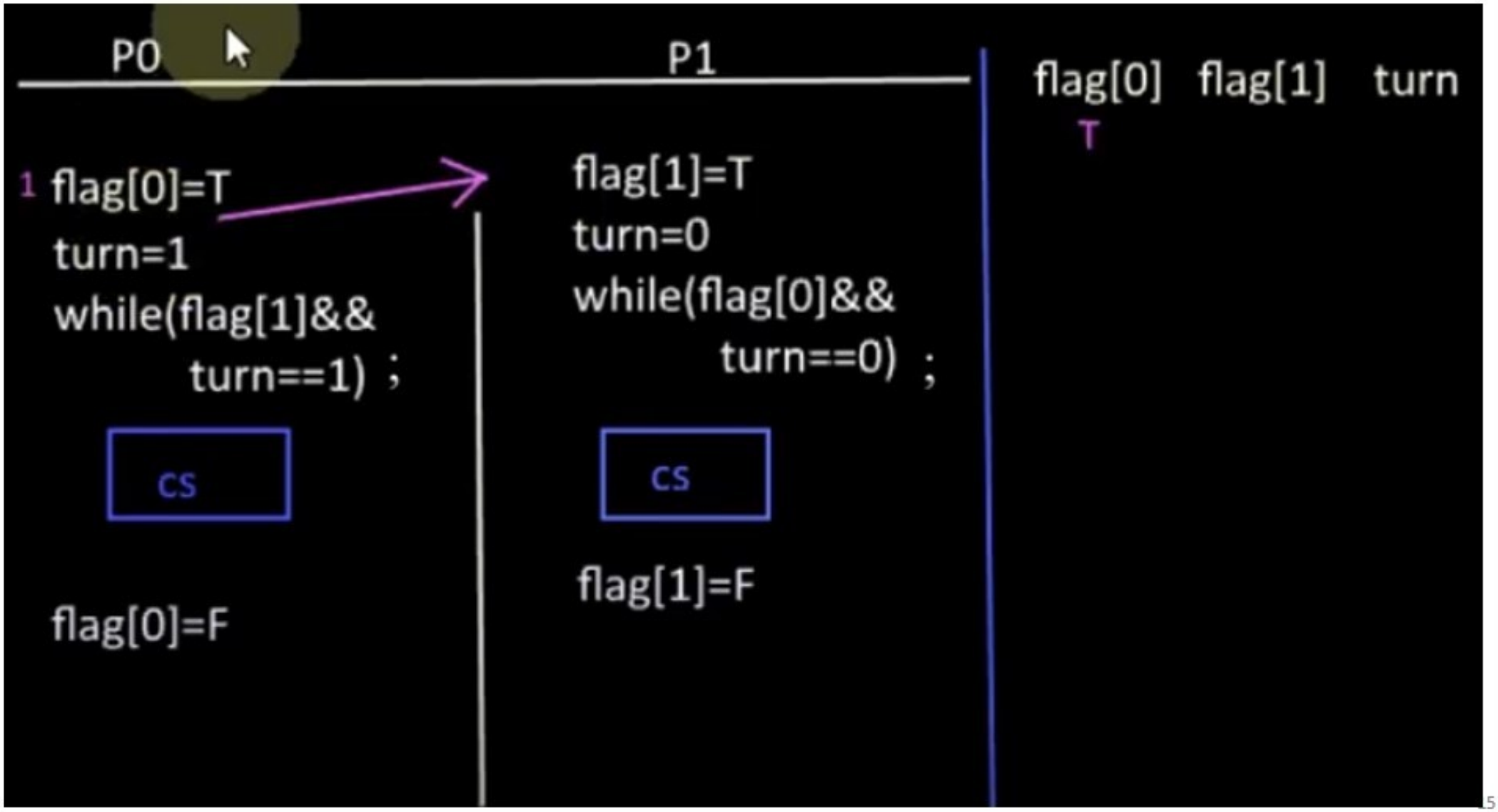
Örnek Çalıştırma 1- Peterson's Solution



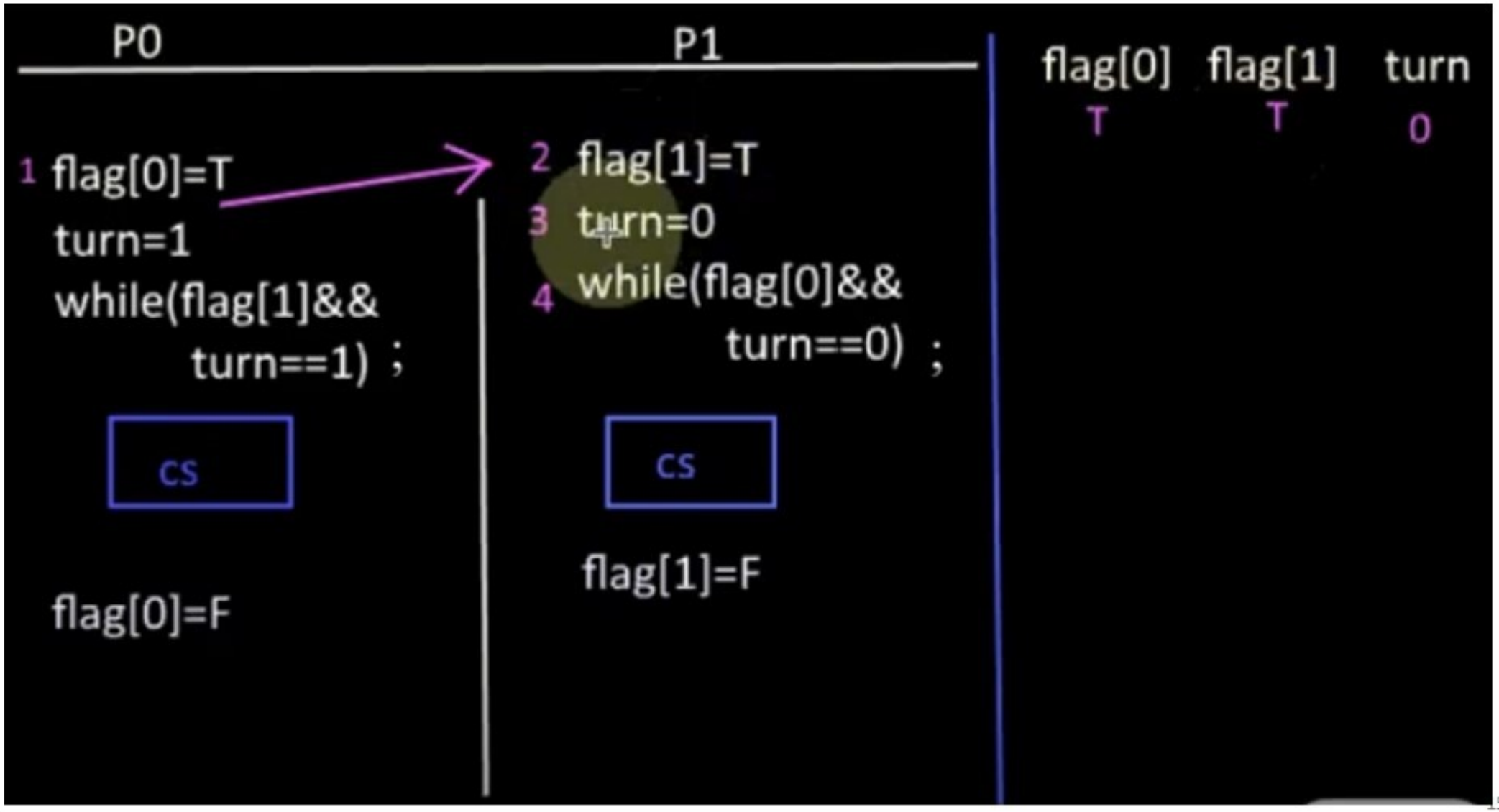
Örnek Çalıştırma 1- Peterson's Solution



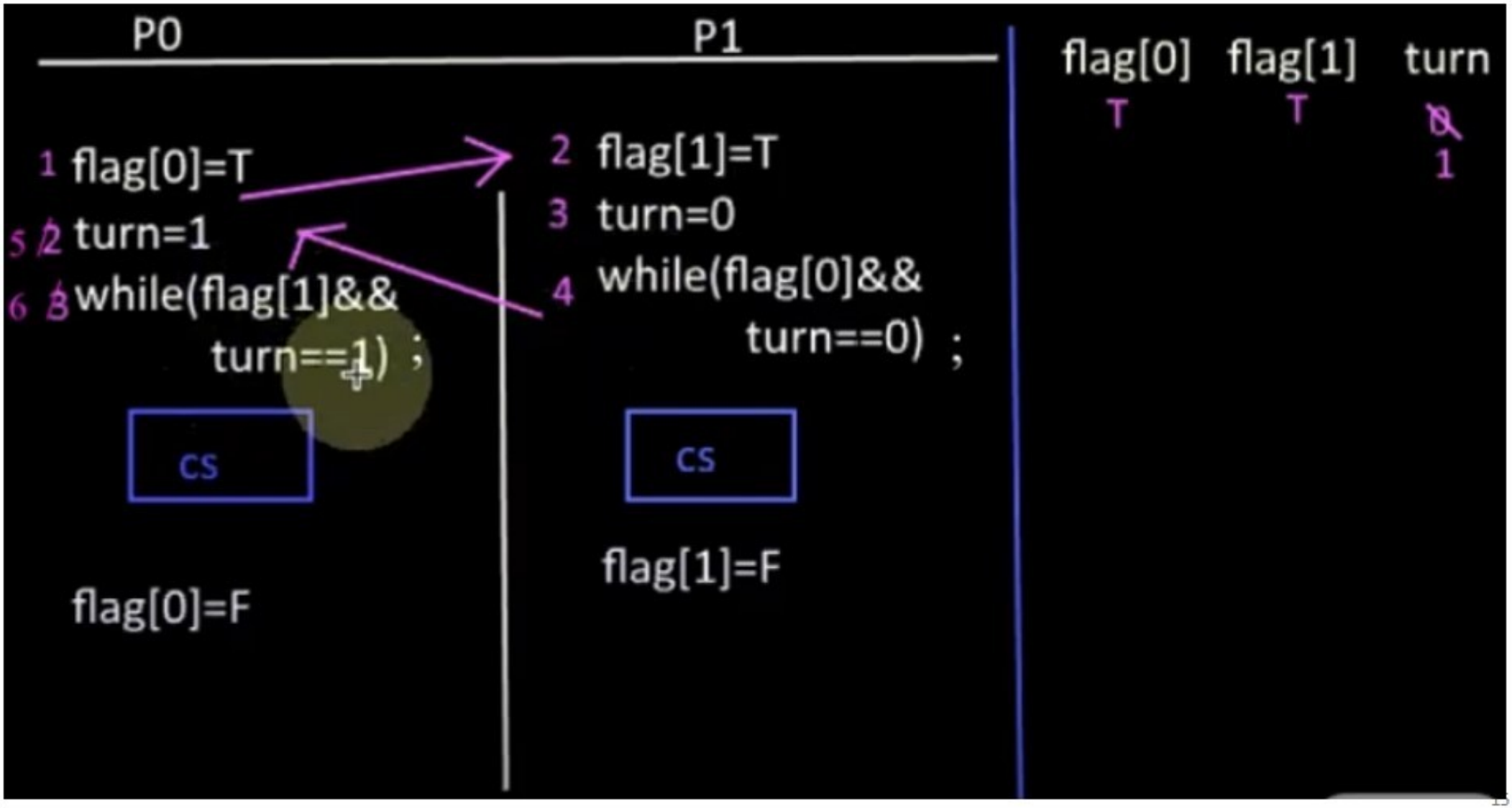
Örnek Çalıştırma 2- Peterson's Solution



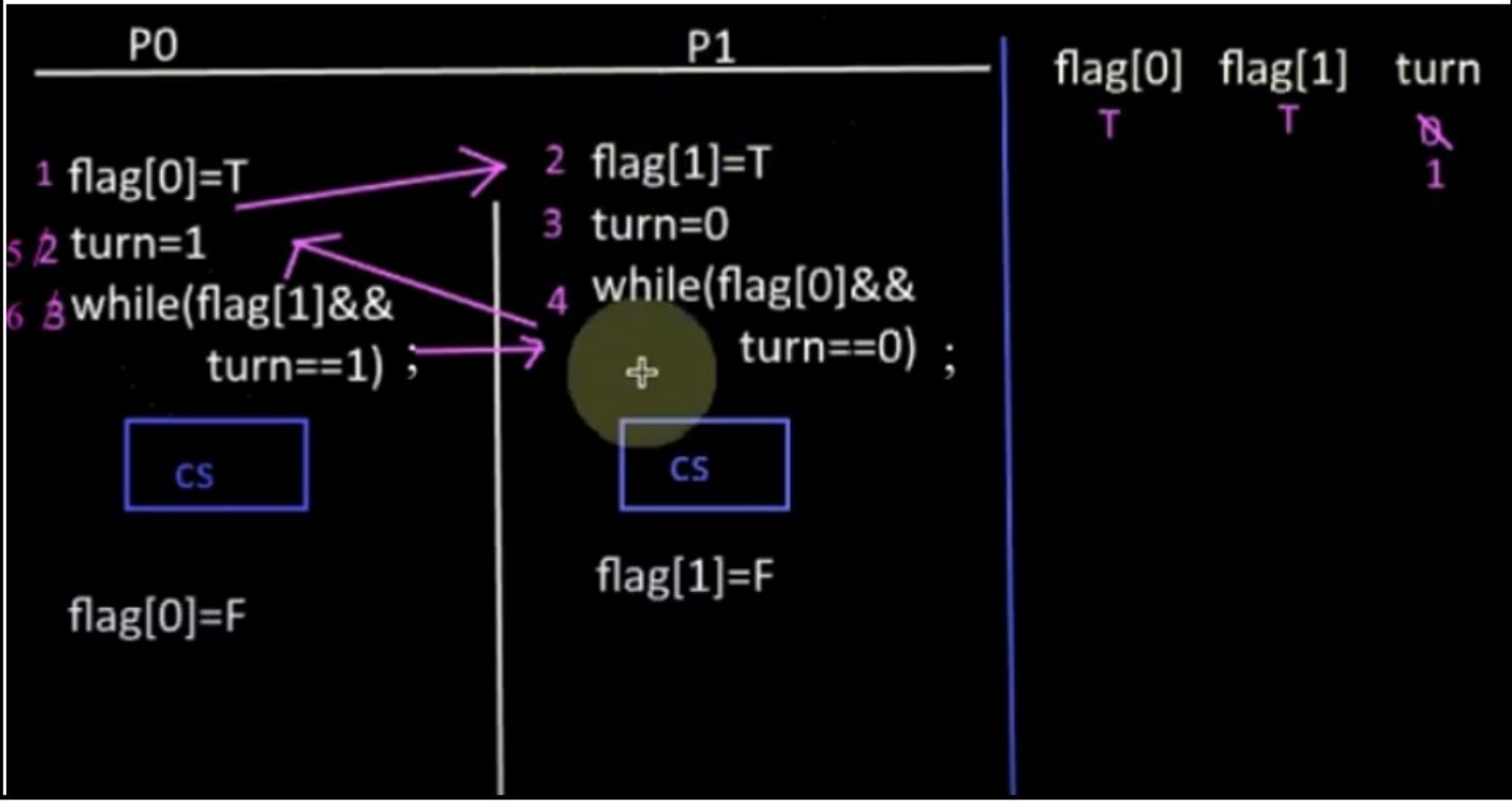
Örnek Çalıştırma 2- Peterson's Solution



Örnek Çalıştırma 2- Peterson's Solution



Örnek Çalıştırma 2- Peterson's Solution

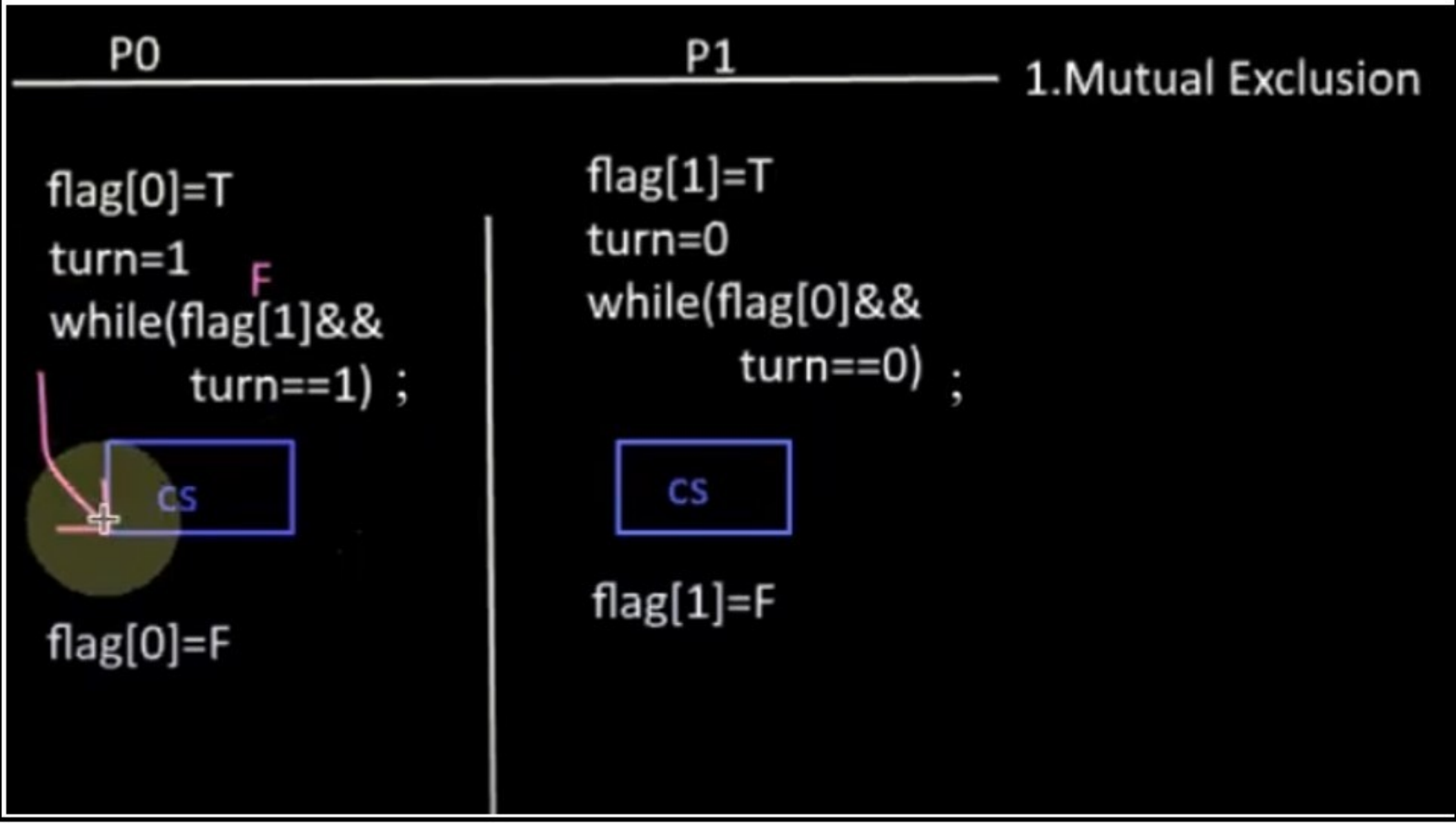


Peterson's Solution Proof

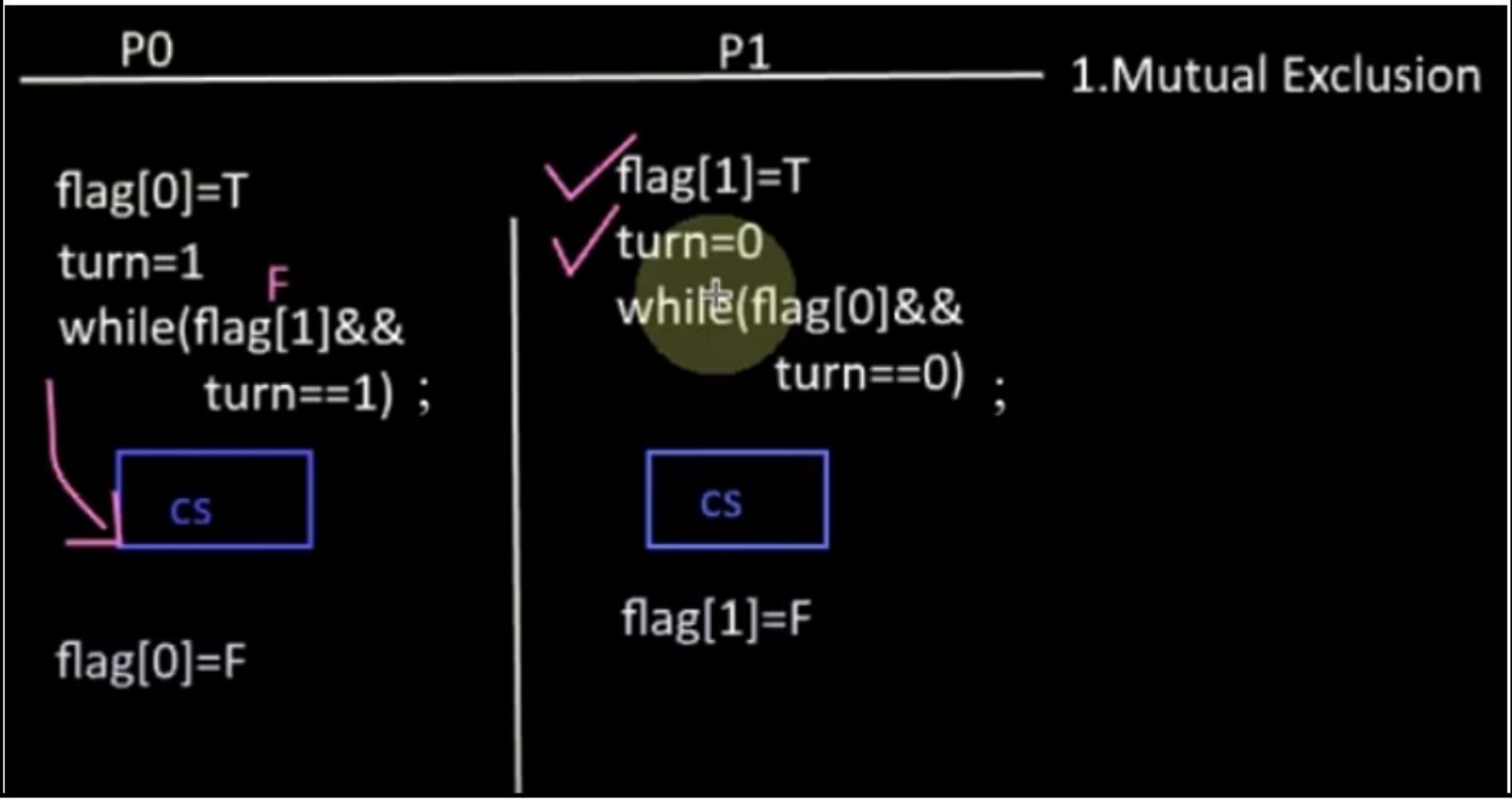
- Peterson's solutionun doğru olup olmadığı critical section probleminin çözümü için sağlanması gereken üç gereksinimin karşılanması ile ispatlanır:

    1. Mutual exclusion
    2. Progress
    3. Bounded waitinig

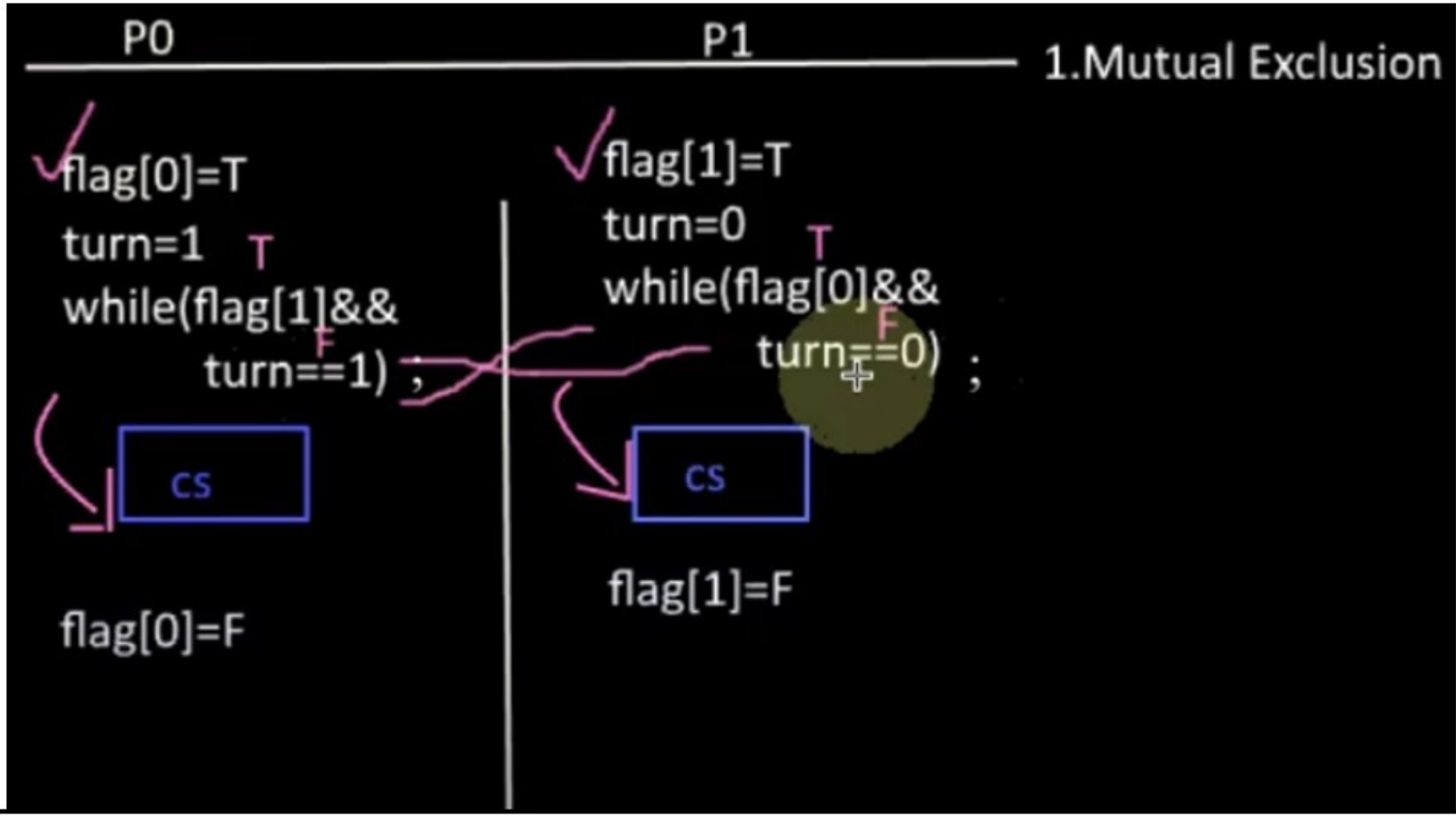
Peterson's Solution **mutual exclusion**'u sağlıyor mu?



Peterson's Solution **mutual exclusion**'u sağlıyor mu?



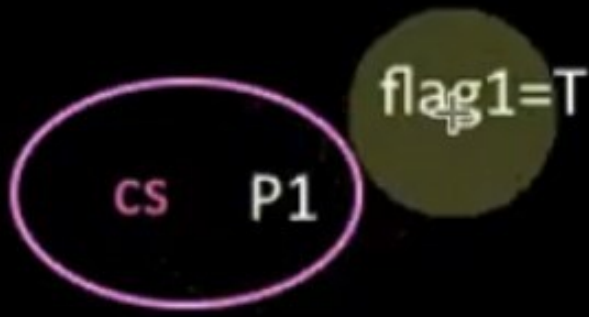
Peterson's Solution **mutual exclusion**'u sağlıyor mu?



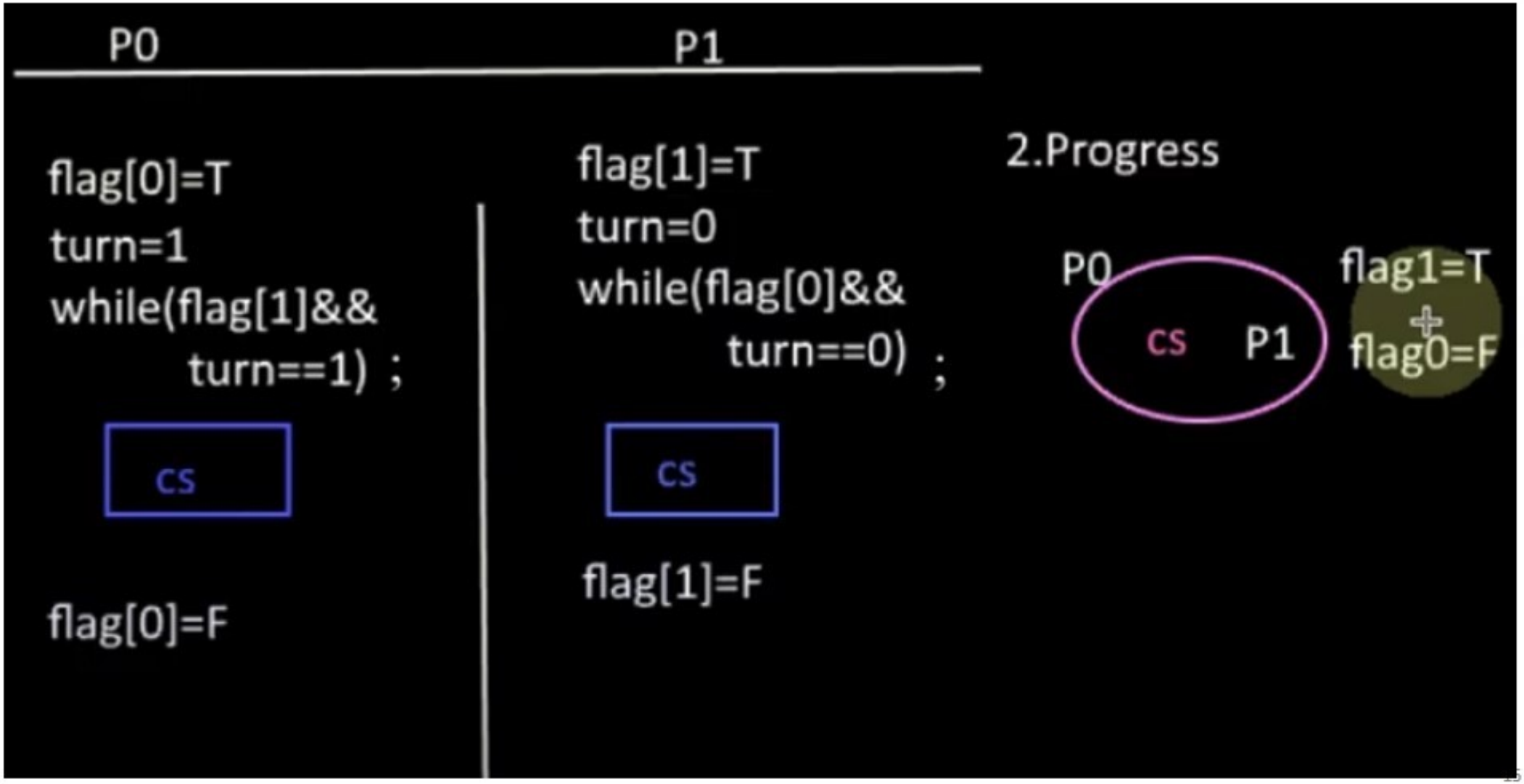
Peterson's Solution **progres**'i sağlıyor mu?

P0	P1
flag[0]=T	flag[1]=T
turn=1	turn=0
while(flag[1]&& turn==1) ;	while(flag[0]&& turn==0) ;
cs	cs
flag[0]=F	flag[1]=F

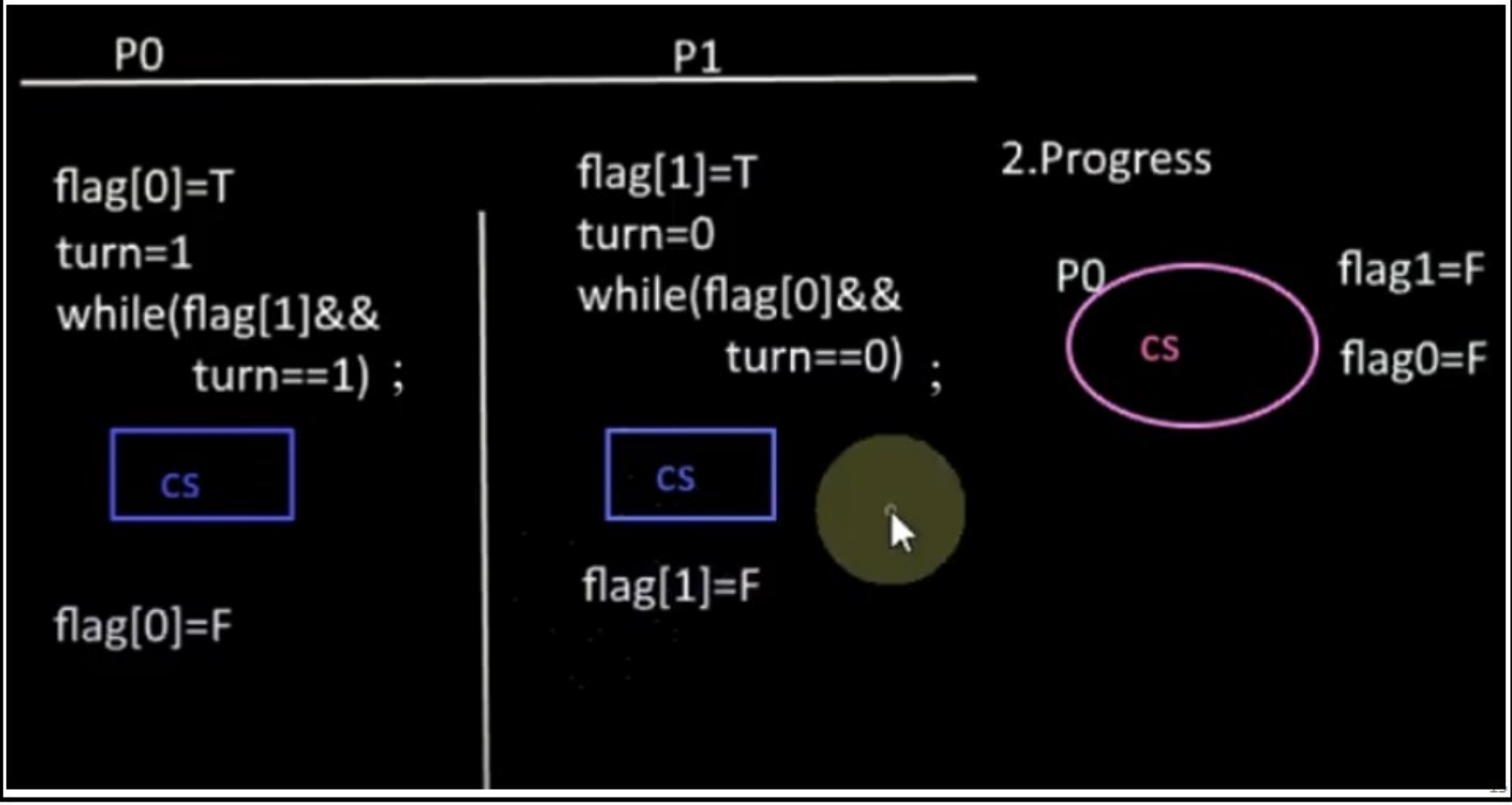
2.Progress



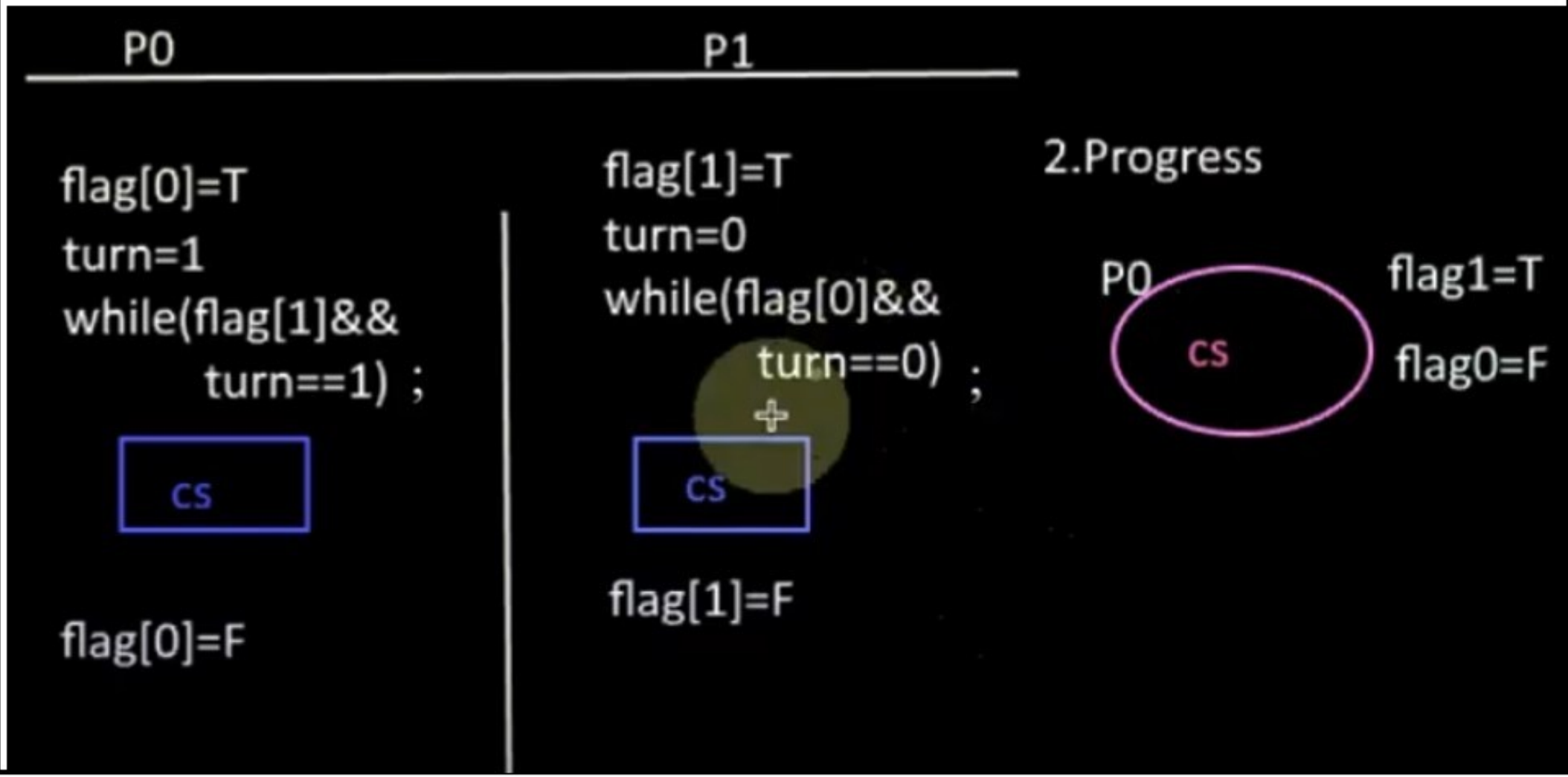
Peterson's Solution **progres**'i sağlıyor mu?



Peterson's Solution **progres**'i sağlıyor mu?



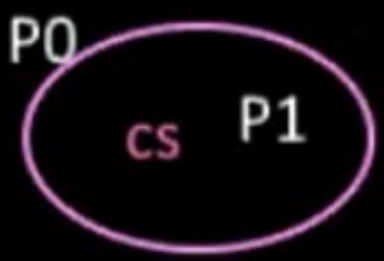
Peterson's Solution **progres**'i sağlıyor mu?



Peterson's Solution **progres**'i sağlıyor mu?

P0	P1
flag[0]=T turn=1 while(flag[1]&& turn==1) ; cs flag[0]=F	flag[1]=T turn=0 while(flag[0]&& turn==0) ; cs flag[1]=F

2.Progress



flag1=T

flag0=F

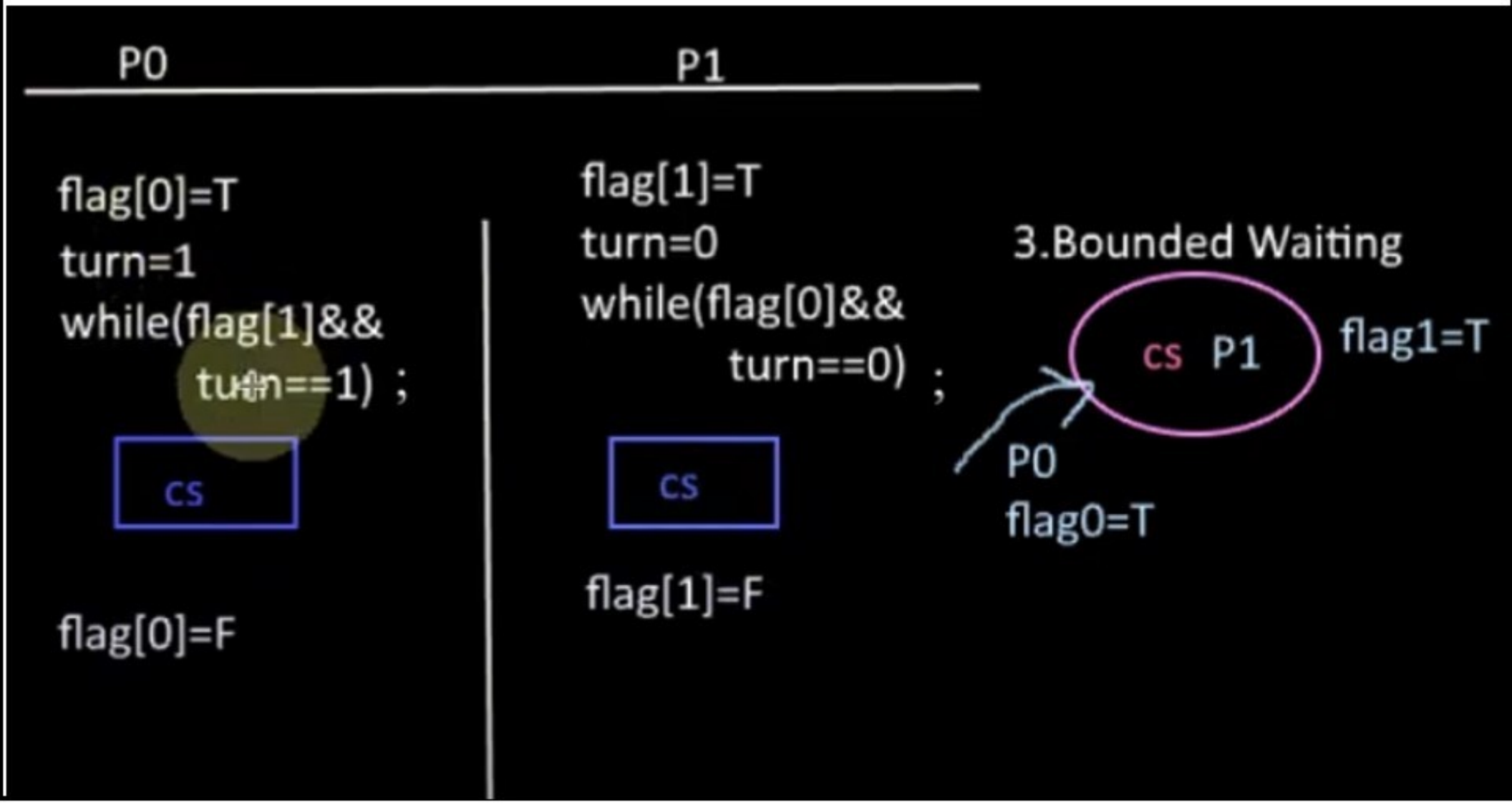
Peterson's Solution **bounded waiting**'i sağlıyor mu?

P0	P1
flag[0]=T turn=1 while(flag[1]&& turn==1) ; cs flag[0]=F	flag[1]=T turn=0 while(flag[0]&& turn==0) ; cs flag[1]=F

3.Bounded Waiting

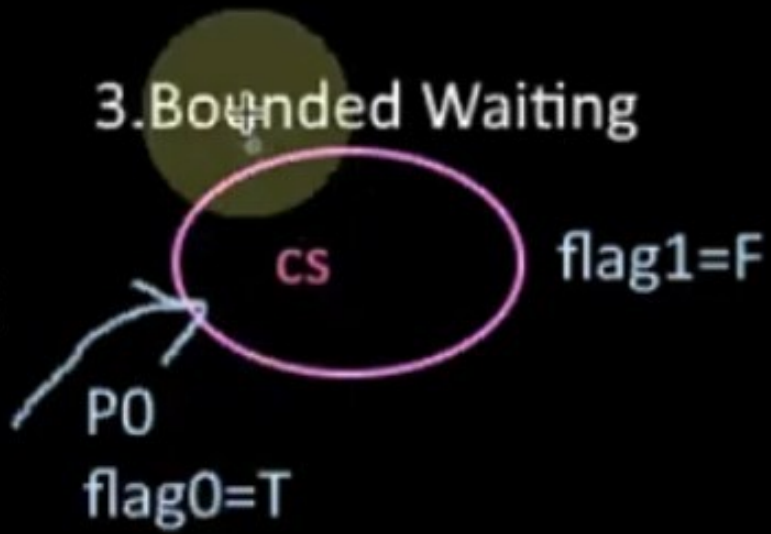
flag1=T

Peterson's Solution **bounded waiting**'i sağlıyor mu?

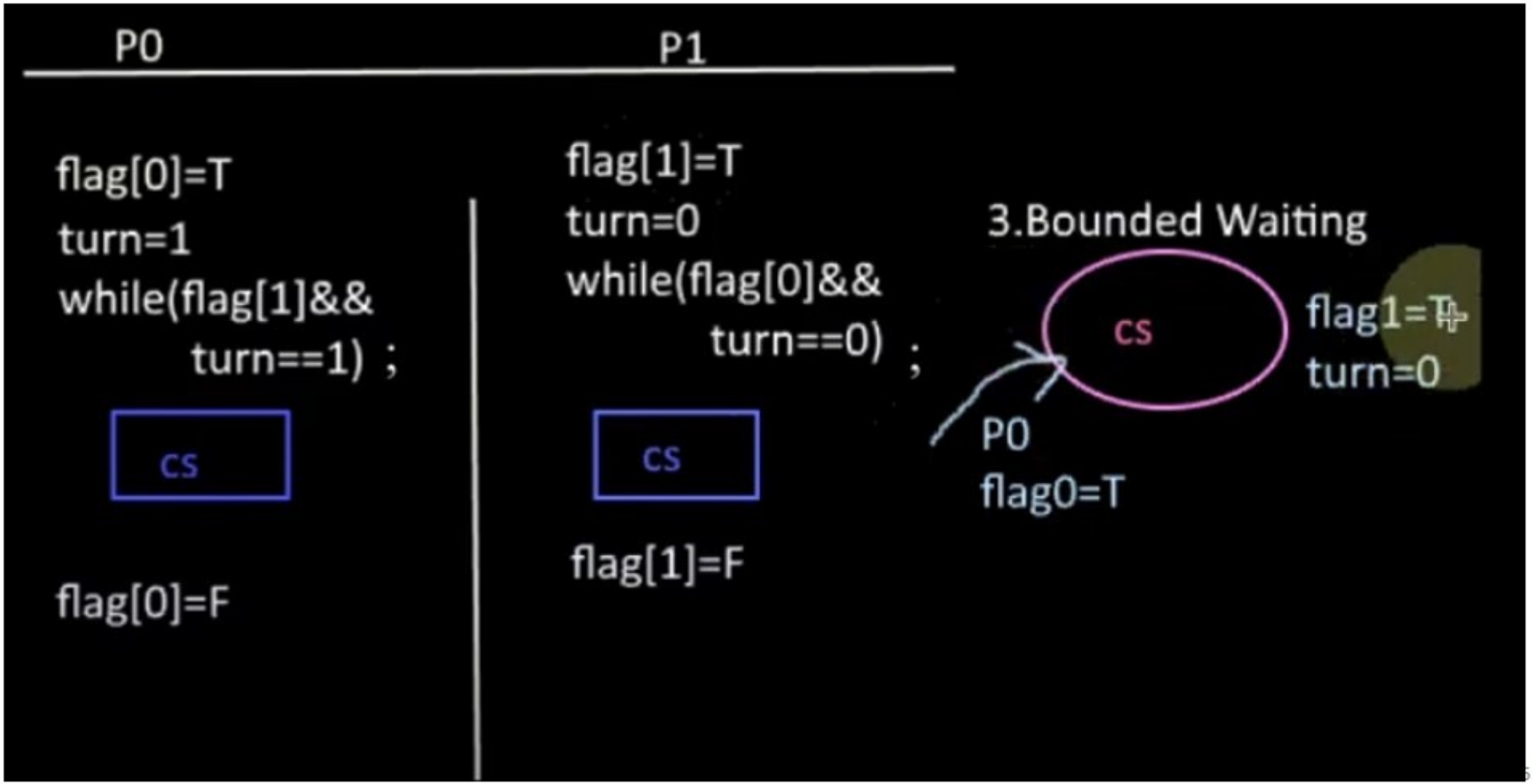


Peterson's Solution **bounded waiting**'i sağlıyor mu?

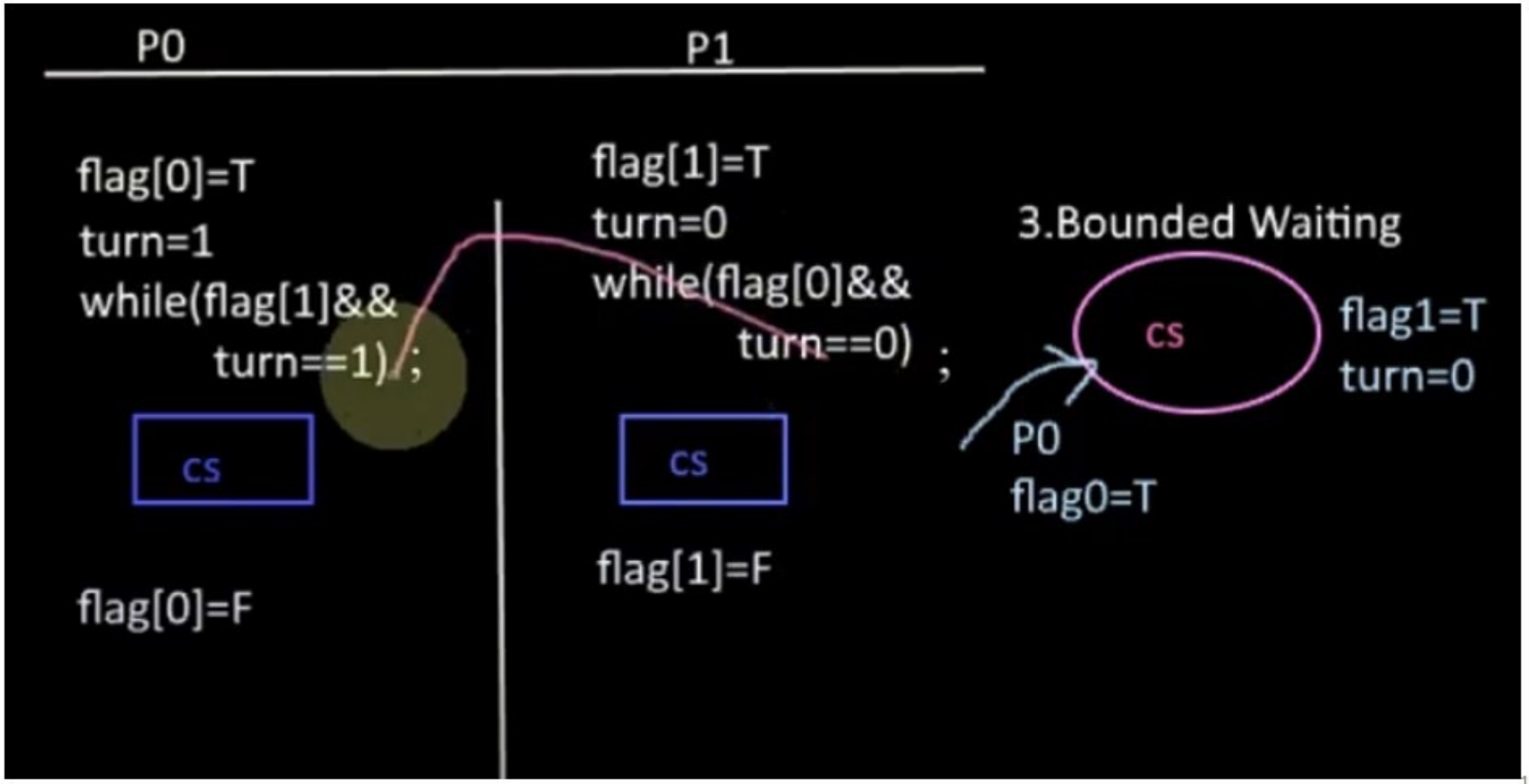
P0	P1
flag[0]=T turn=1 while(flag[1]&& turn==1) ; cs flag[0]=F	flag[1]=T turn=0 while(flag[0]&& turn==0) ; cs flag[1]=F



Peterson's Solution **bounded waiting**'i sağlıyor mu?

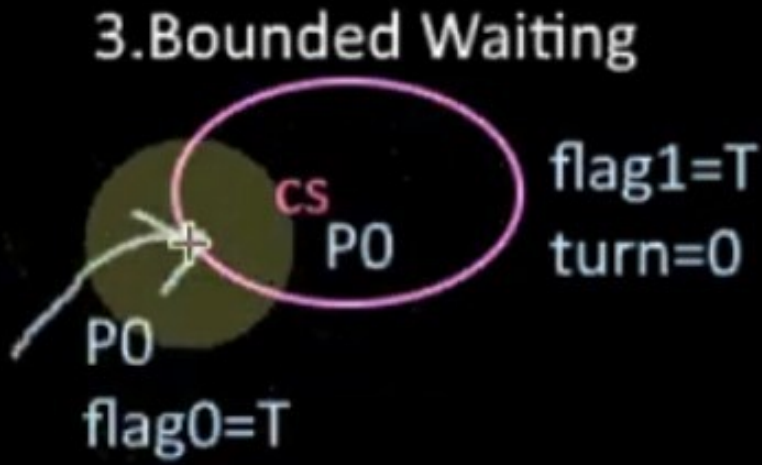


Peterson's Solution **bounded waiting**'i sağlıyor mu?



Peterson's Solution **bounded waiting**'i sağlıyor mu?

P0	P1
flag[0]=T	flag[1]=T
turn=1	turn=0
while(flag[1]&& turn==1) ;	while(flag[0]&& turn==0) ;
cs	cs
flag[0]=F	flag[1]=F



Peterson's Solution modern bilgisayar mimarilerinde geçerli mi?

- **Örnek:**

İki thread tarafından paylaşılan veriler

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

Thread1

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

Thread2

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