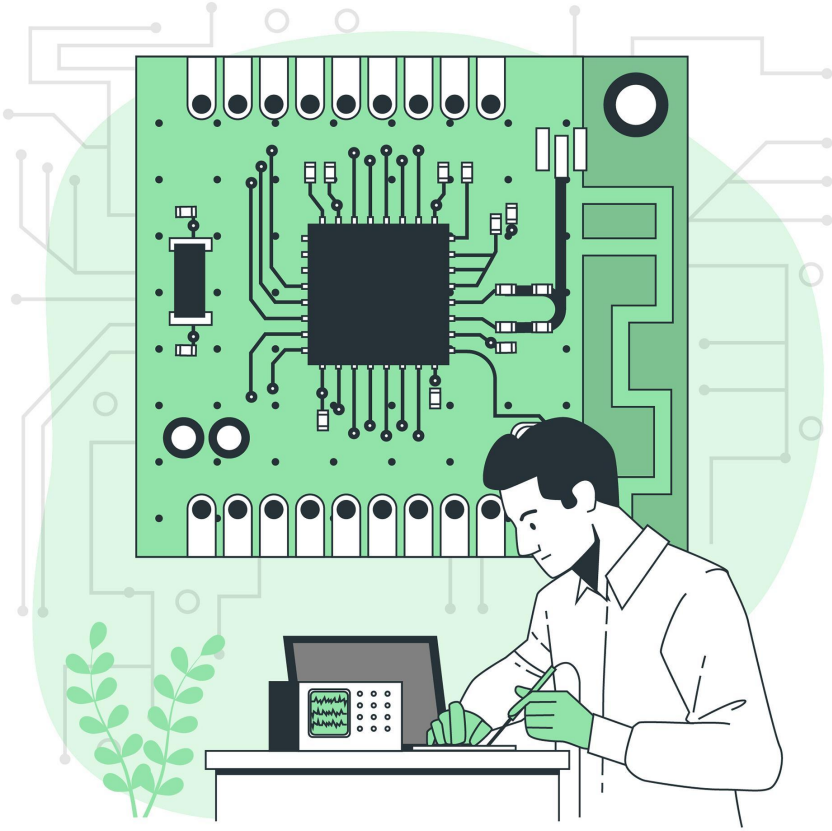
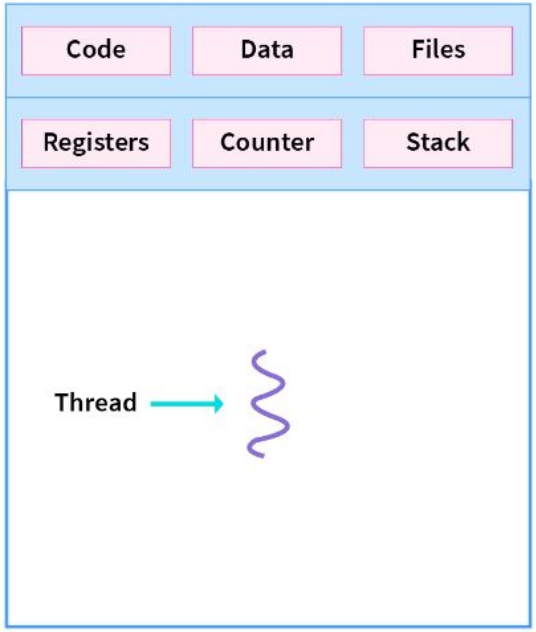
03
İş Parçacığı
Threads

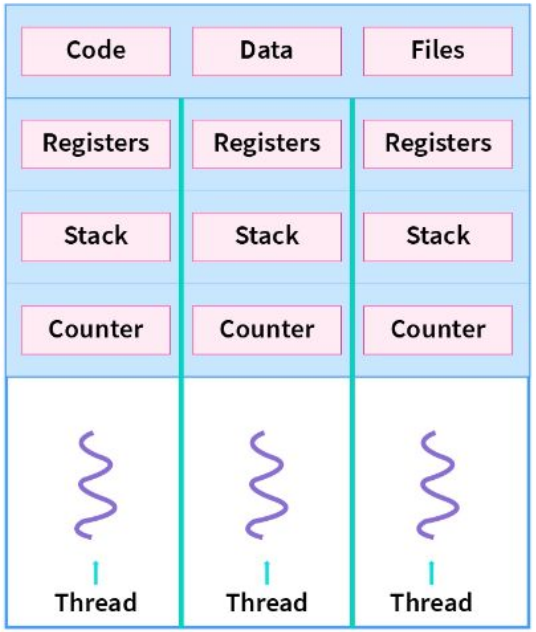


İçindekiler

İş Parçacığı Nedir?

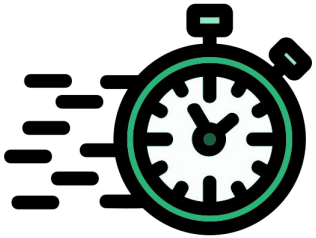


Single-threaded process



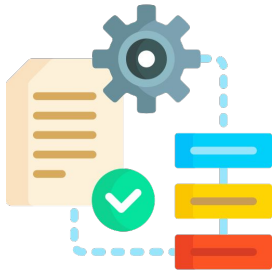
Multithreaded process

İş Parçacığının Faydaları

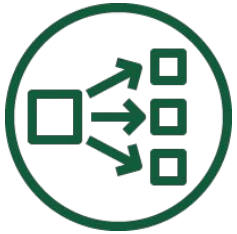


Hızlı yanıt verme kapasitesi

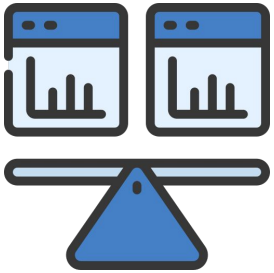
Daha az hafıza gereksinimi



Oluşturma ve yok etme kolaylığı



Parallel Çalışma



İşlemciler arasında yük dengelemesi

İş Parçacığı Örneği

HayaliRoman.docx

Page: 600

Bolum1.docx ..... BolumN.docx

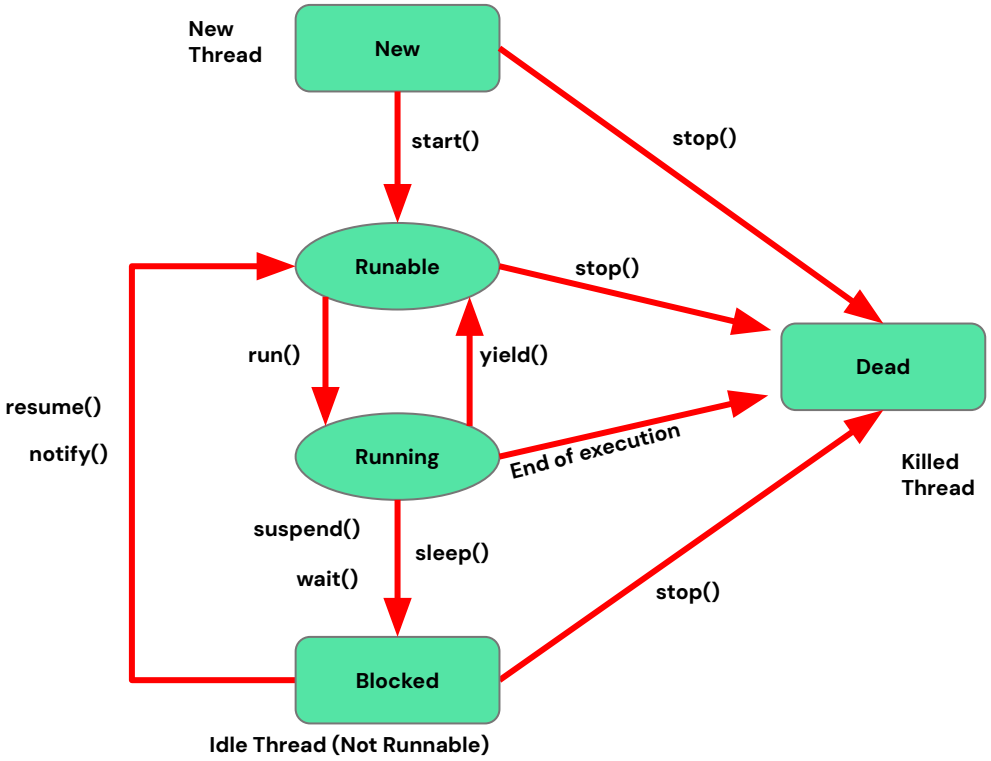
İş Parçacığı Örneği

ön
iş parçacığı

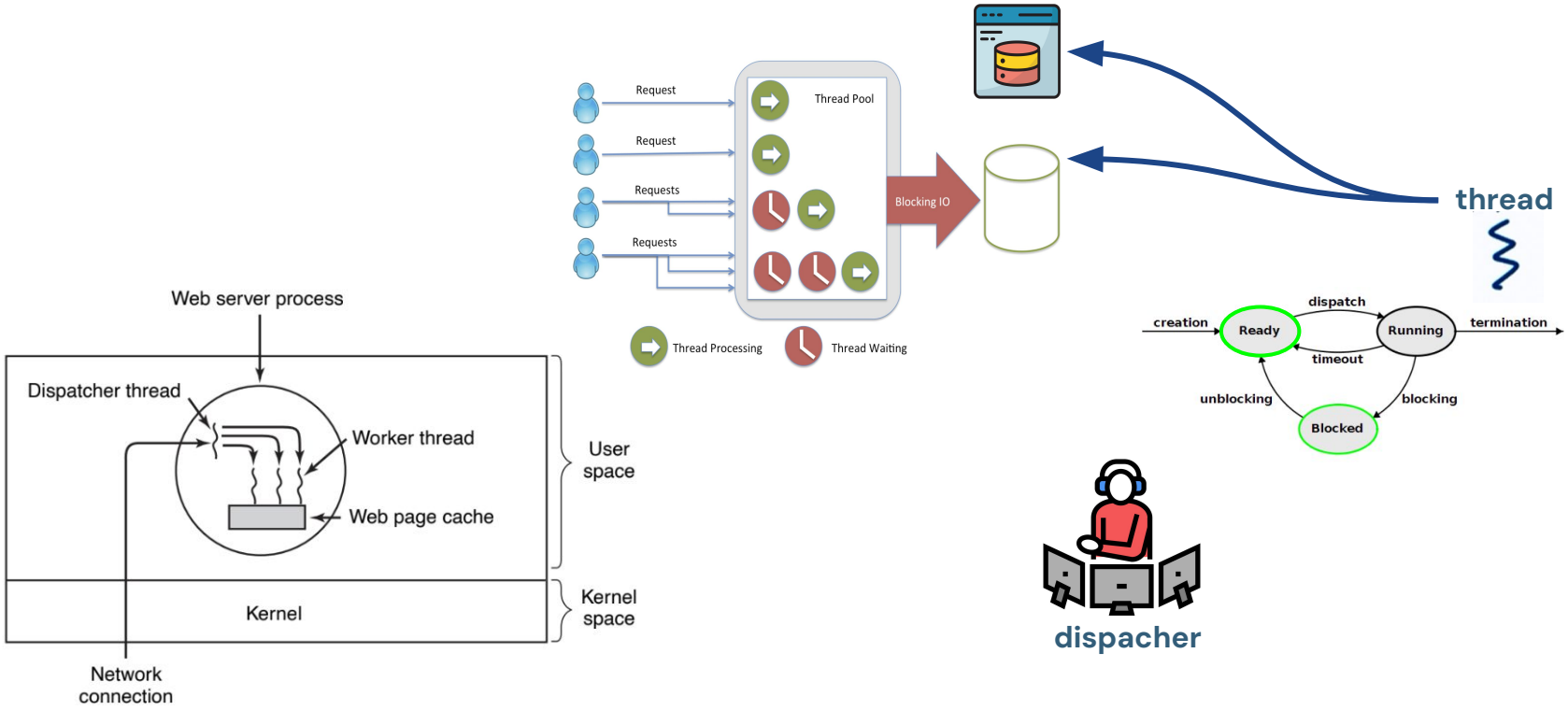
arkaplan
iş parçacığı

kesme ile çalışan
iş parçacığı

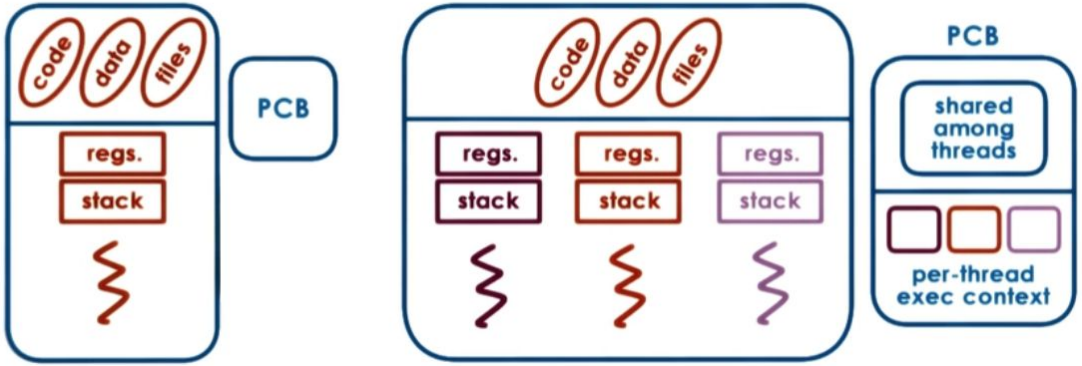
İş Parçacığı Yaşam Döngüsü



İş Parçacığı Yaşam Döngüsü Örneği

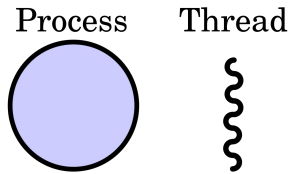


Klasik İş Parçacığı Modeli



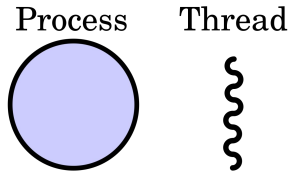
Süreç ile İş Parçacığı Farkları

Süreç	İş Parçacığı



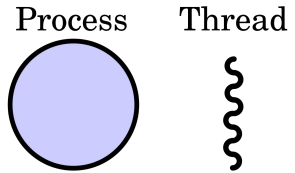
Süreç ile İş Parçacığı Farkları

	Süreç	İş Parçacığı
İletişim	Uzun	Kısa



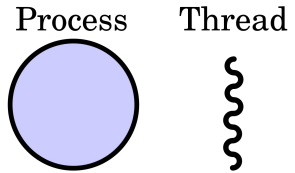
Süreç ile İş Parçacığı Farkları

	Süreç	İş Parçacığı
İletişim	Uzun	Kısa
Kaynak Kullanımı	Fazla	Az



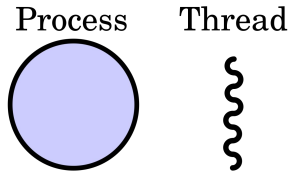
Süreç ile İş Parçacığı Farkları

	Süreç	İş Parçacığı
İletişim	Uzun	Kısa
Kaynak Kullanımı	Fazla	Az
Hafıza	Paylaşmaz	Paylaşır



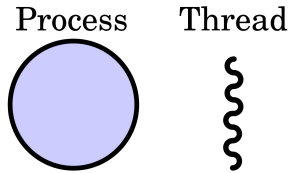
Süreç ile İş Parçacığı Farkları

	Süreç	İş Parçacığı
İletişim	Uzun	Kısa
Kaynak Kullanımı	Fazla	Az
Hafıza	Paylaşmaz	Paylaşır
Veri Paylaşımı	Yok	Var



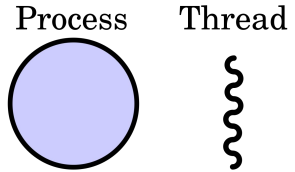
Süreç ile İş Parçacığı Farkları

	Süreç	İş Parçacığı
İletişim	Uzun	Kısa
Kaynak Kullanımı	Fazla	Az
Hafıza	Paylaşmaz	Paylaşır
Veri Paylaşımı	Yok	Var
Hafıza Kullanımı	Fazla	Az



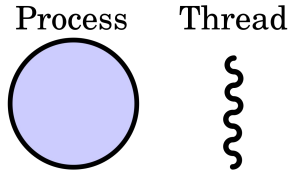
Süreç ile İş Parçacığı Farkları

	Süreç	İş Parçacığı
İletişim	Uzun	Kısa
Kaynak Kullanımı	Fazla	Az
Hafıza	Paylaşmaz	Paylaşır
Veri Paylaşımı	Yok	Var
Hafıza Kullanımı	Fazla	Az
Hata etkisi	Etkilemez	Etkilenir

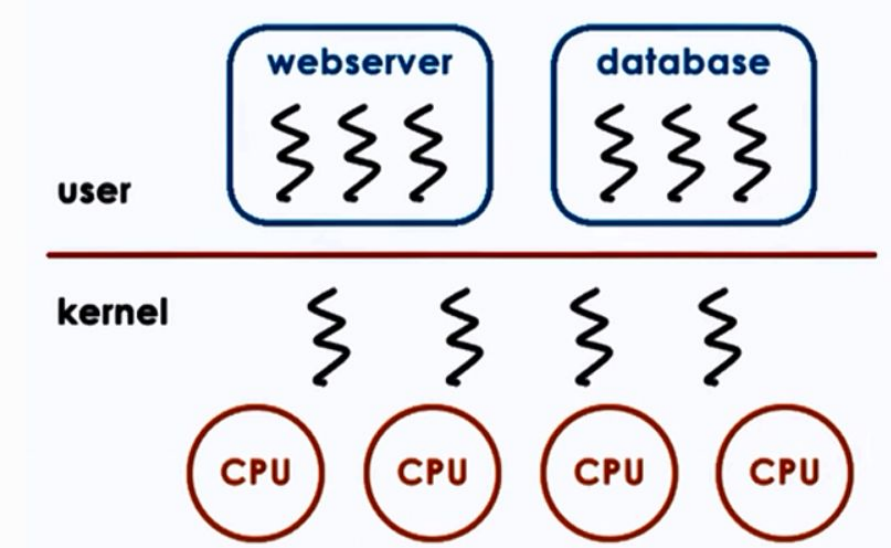


Süreç ile İş Parçacığı Farkları

	Süreç	İş Parçacığı
İletişim	Uzun	Kısa
Kaynak Kullanımı	Fazla	Az
Hafıza	Paylaşmaz	Paylaşır
Veri Paylaşımı	Yok	Var
Hafıza Kullanımı	Fazla	Az
Hata etkisi	Etkilemez	Etkilenir
Sonlanma Zamanı	Fazla	Az



Çekirdek/Kullanıcı İş Parçacıkları



Çekirdek/Kullanıcı İş Parçacıkları

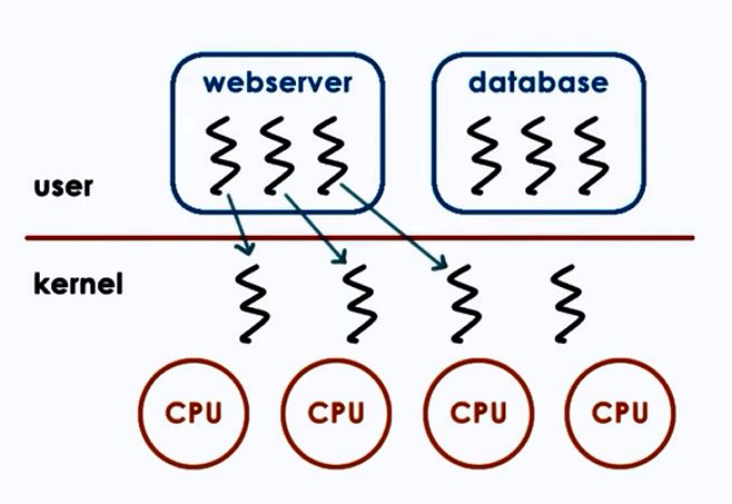
One to One model:

Avantajları:

- OS tüm iş parçacıklarını bilir
- Senkronizasyon kolaydır
- Engelleme (Bloking) işlemi kolaydır

Dezavantajları:

- Her işleme OS müfahildir
- OS iş parçacıklarını sınırlayabilir
- Taşınamaz



Çekirdek/Kullanıcı İş Parçacıkları

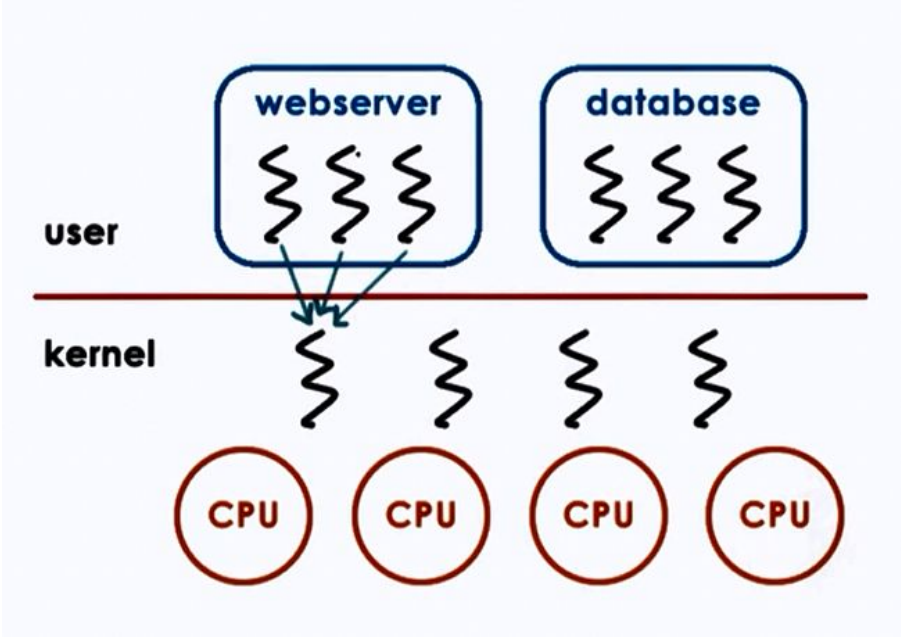
Many to One model:

Avantajları:

- Taşınabilir
- İş parçacıklarında OS sınırlaması yoktur

Dezavantajları:

- Kullanıcı iş parçacığı G/Ç işleminde bloke edilirse OS süreci engelleyebilir.



Çekirdek/Kullanıcı İş Parçacıkları

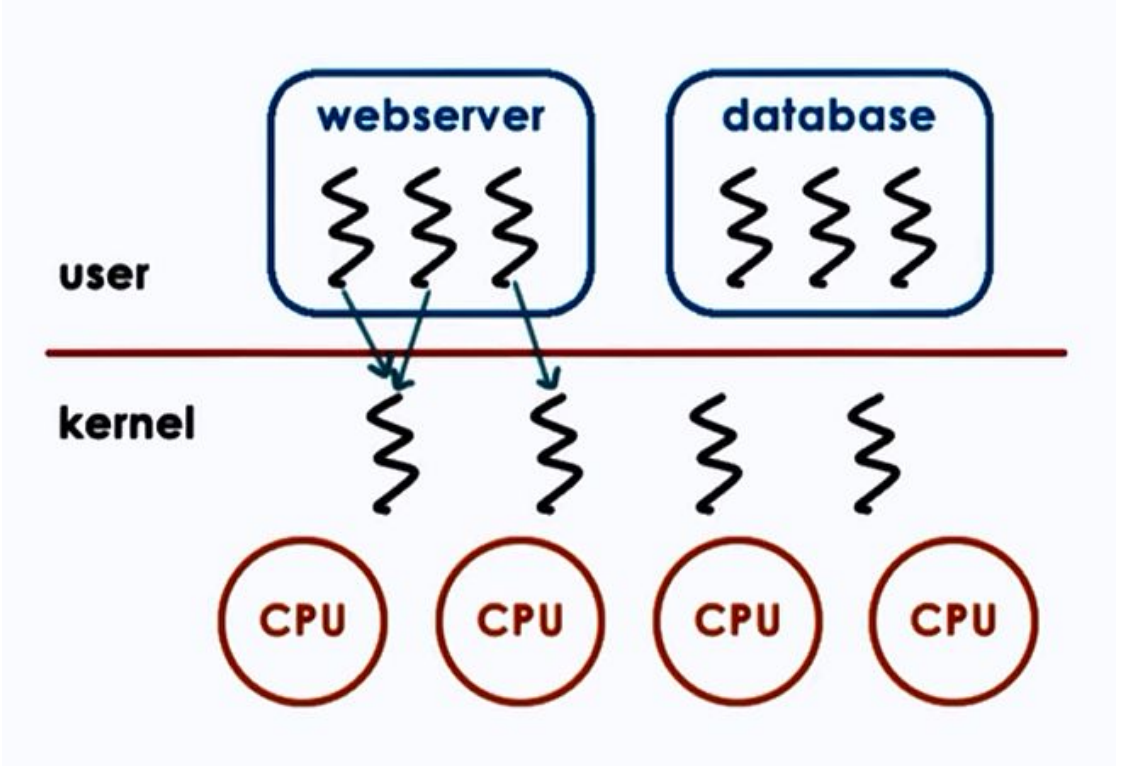
Many to Many model:

Avantajları:

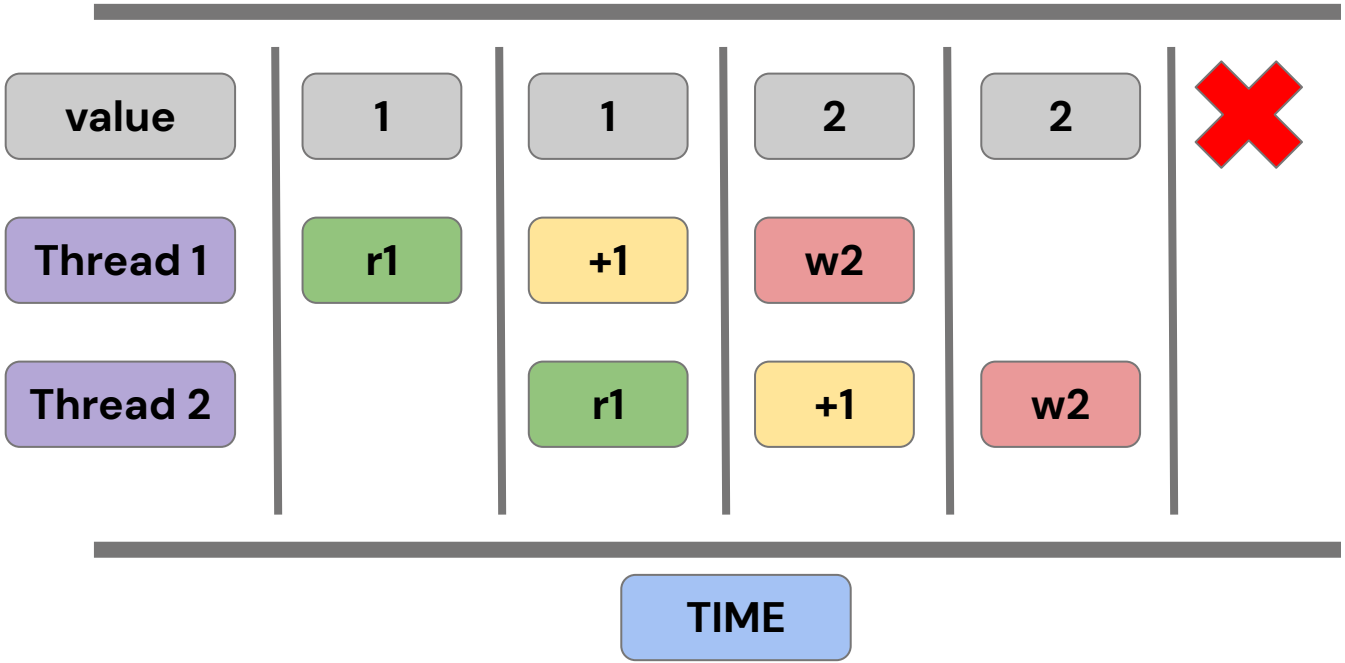
- En iyi modeldir
- Bağlı veya ilişkisiz iş parçacıkları olabilir

Dezavantajları:

- Koordinasyon gerektirir



Race Condition (Yarış Durumu)



Race Condition Neden Olur?

1. **Paylaşılan Kaynaklara Eş Zamanlı Erişim**

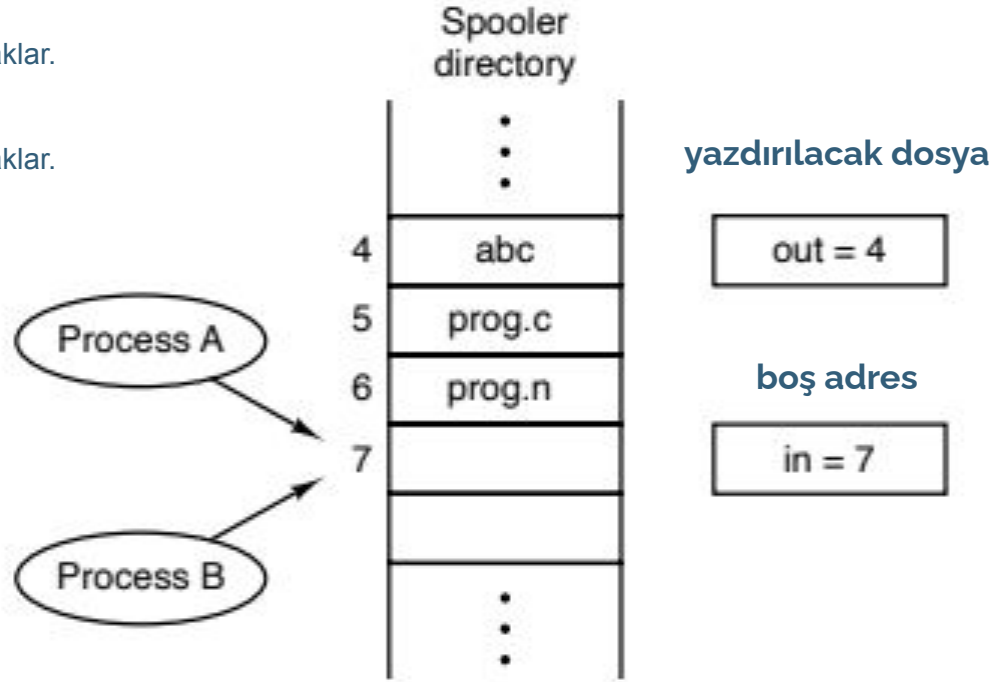
   Birden fazla süreç veya iş parçacığı, aynı veriye veya kaynağa eş zamanlı olarak erişmeye çalışır.

2. **Yetersiz Senkronizasyon**

   Süreçler veya iş parçacıkları arasındaki etkileşimler yeterince senkronize edilmediğinde, beklenmedik sıralamalar ve sonuçlar ortaya çıkabilir.

Race Condition: Yazdırma Kuyruğu (Printer Spooler)

1. **A**, **in** değerini (7) okur ve kendi yerel değişkeninde saklar.
2. **CPU**, **A**'yı durdurarak **B**'ye geçiş yapar.
3. **B, in** değerini (7) okur ve kendi yerel değişkeninde saklar.
4. **B** yazdıracağı dosya adını adres 7'ye kaydeder
5. **B, in** değerini 8 olarak günceller.
6. **A** çalışmaya başlar
7. **A** yazdıracağı dosyanın adını adres 7'ye kaydeder.
8. **A in** değerini 8 olarak günceller.



Race Condition Çözüm Yöntemleri

Kilitler (Locks)

Kritik bölgelere erişimi sınırlamak için kilitler kullanılır. Örneğin, mutex kilitleri.

Semaforlar

Belirli kaynaklara erişimi sınırlamak ve senkronizasyon sağlamak için semaforlar kullanılır.

Atomik İşlemler

Birtakım işlemler, bölünemez ve kesintisiz olarak atomik olarak işaretlenebilir.

Transactionlar

Bazı sistemler, işlemleri başlatıp tamamlanana kadar kaynakları koruyan transaction mekanizmaları kullanır.

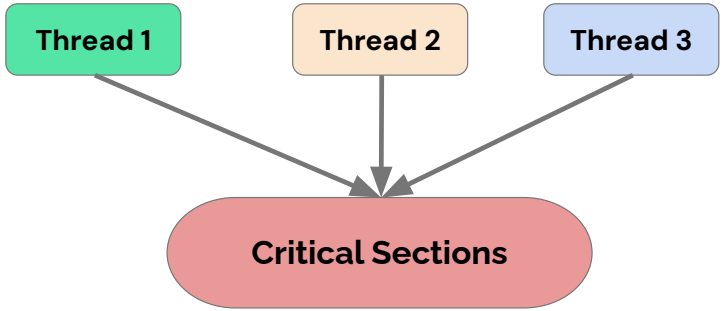
Critical Sections (Kritik Bölgeler)

Süreçlerin **paylaşılan kaynaklara** erişen program bölümü, **kritik bölge (critical region)** veya **kritik kesit (critical section)** olarak adlandırılır.

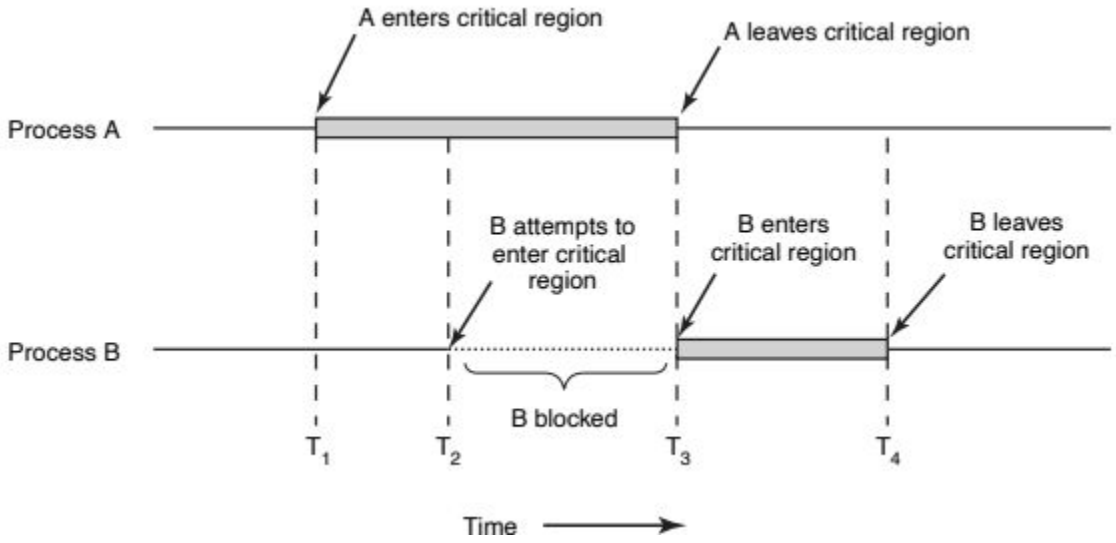
karşılıklı dışlama (mutual exclusion) : **Birden fazla sürecin aynı anda bu paylaşılan verilere erişmesini engelleme**

Çözüm:

1. **İki süreç aynı anda kritik bölgelerinde olamaz.**
2. **Süreçlerin hızları veya CPU sayısı hakkında varsayım yapılamaz.**
3. **Kritik bölgesinin dışında çalışan bir süreç, başka bir sürecin çalışmasını engelleyemez (bloklamaz).**
4. **Hiçbir süreç, kritik bölgesine girmek için sonsuza kadar beklemek zorunda kalmamalıdır.**



Critical Sections (Kritik Bölgeler)

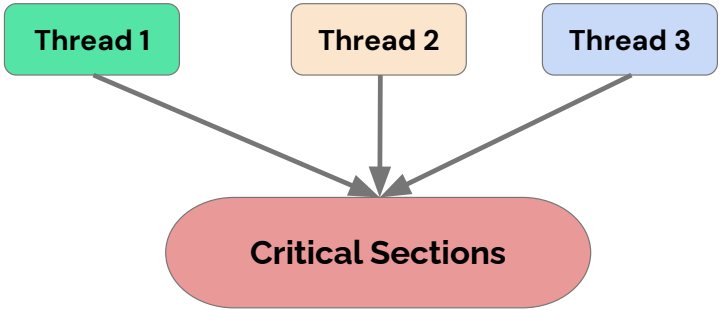


Yoğun Bekleme ile Karşılıklı Dışlama Kritik Bölge Yönetimi

Kesme (Interrupt) Devre Dışı Bırakma Yöntemi

Süreç, **kritik bölgesine girer girmez tüm kesmeleri devre dışı bırakır**

kritik bölgeden çıkmadan hemen önce bunları tekrar **etkinleştirir**.



Sakıncaları

Kullanıcı süreçlerine kesmeleri devre dışı bırakma yetkisi vermek **tehlikelidir**:

Birden fazla CPU varsa— kesmeleri devre dışı bırakmak **sadece o anki CPU'yu etkiler**.

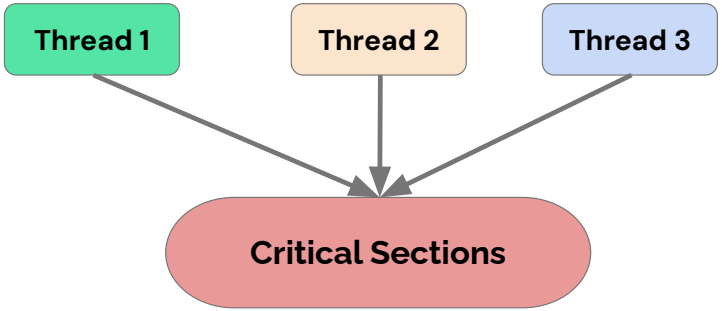
Yoğun Bekleme ile Karşılıklı Dışlama Kritik Bölge Yönetimi

Kilit Değişkenleri (Lock Variables)

Süreç **kilit değişkenini kontrol eder**.

Eğer kilidin değeri **0** ise (yani hiçbir süreç kritik bölgede değilse), süreç **kilidi 1 yapar** ve **kritik bölgesine girer**.

Eğer kilidin değeri **1** ise (yani başka bir süreç içerideyse),süreç **kilidin tekrar 0 olmasını bekler**.



Sakıncaları

Bir süreç, kilidin değerini okur ve **0** görür.

Tam kilidi **1** yapmadan önce, işlemci başka bir süreci çalıştırır.

İkinci süreç çalışır, kilidi **1** yapar ve kritik bölgesine girer.

Daha sonra ilk süreç tekrar çalışmaya başladığında, kendi kodunu kaldığı yerden devam ettirir yani **kilidi 1 yapar** (önceden 0 olduğunu sanarak).

Böylece **iki süreç aynı anda kritik bölgelerinde olur**, ve **karşılıklı dışlama (mutual exclusion)** bozulur.

Yoğun Bekleme ile Karşılıklı Dışlama Kritik Bölge Yönetimi

Sıkı Sıralama (Strict Alternation)

Süreç 0 → Süreç 1 → Süreç 0 → Süreç 1 ...

Başlangıçta `turn = 0` olduğu için, **process 0** kritik bölgesine girebilir.
Process 1 ise `turn`'ün 1 olmasını bekler ve bu sırada **yoğun bekleme (busy waiting)** döngüsünde takılır.

Process 0 kritik bölgesinden çıkınca `turn = 1` yapar, böylece **process 1**'in sıradaki giriş hakkı olur.

Sakıncaları

Şimdi **process 0** çok hızlı davranır, kritik bölgesine girer, çıkar ve `turn = 1` yapar.
Artık **turn = 1**, her iki süreç de kritik olmayan bölgededir.
process 0 işini erken bitirip yeniden kritik bölgesine girmek isterse, `turn = 1` olduğu için **giremez**.
process 0, sadece diğer süreç (process 1) `turn`'ü tekrar 0 yapana kadar bekler.

(a) Process 0

```
while (TRUE) {
    while (turn != 0) { }   /* loop */
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(b) Process 1

```
while (TRUE) {
    while (turn != 1) { }   /* loop */
    critical_region();
    turn = 0;
    noncritical_region();
}
```

Yoğun Bekleme ile Karşılıklı Dışlama Kritik Bölge Yönetimi

Peterson Algoritması

```
#define FALSE 0
#define TRUE 1
#define N 2      /* süreç sayısı */

int turn;           /* kimin sırası? */
int interested[N];    /* her eleman başlangıçta FALSE */

void enter_region(int process) {  /* process = 0 veya 1 */
    int other;                  /* diğer sürecin numarası */
    other = 1 - process;          /* process 0 ise other 1, tersi durumda 0 */

    interested[process] = TRUE;   /* sürecin ilgilendiğini belirt */
    turn = process;               /* sırasını kendine ver */

    /* diğer süreç ilgileniyor ve sıra da bu süreçteyse, bekle */
    while (turn == process && interested[other] == TRUE)
        ;  /* boş döngü - busy waiting */
}

void leave_region(int process) {  /* kritik bölgeden çıkış */
    interested[process] = FALSE;  /* artık ilgilenmiyor */

}
```

1. **Başlangıç durumu:**
   Hiçbir süreç kritik bölgede değildir
   (`interested[0] = interested[1] = FALSE`).

2. **Process 0 giriş yapmak ister:**
   - `interested[0] = TRUE` yapar (kritik bölgeye girmek istiyor).
   - `turn = 0` yapar (önceliği kendisine verir).
   - **Process 1** şu anda ilgilenmiyorsa (`interested[1] = FALSE`), **Process 0** hemen kritik bölgeye girer.

3. **Process 1 de giriş yapmak isterse:**
   - `interested[1] = TRUE` ve `turn = 1` olur.
   - Ancak `interested[0]` hâlâ TRUE olduğu için **Process 1**, döngüde bekler (`while` içinde).
   - Process 0 kritik bölgeden çıkıp `interested[0] = FALSE` yaptığında, **Process 1** beklemeyi bırakır ve içeri girer.

Yoğun Bekleme ile Karşılıklı Dışlama Kritik Bölge Yönetimi

TSL Komutunun Çalışma Prensibi

Bir **TSL RX, LOCK** komutu şu şekilde çalışır:

1. **Bellekteki `lock` değişkeninin değerini** alır ve **RX register'ına (yazmaç)** kopyalar.
2. Ardından **`lock` değişkenine 1 yazar** (yani onu kilitli hale getirir).
3. Bu iki işlem (okuma ve yazma) **atomik** olarak yapılır — yani **bölünemez**.
   Hiçbir başka işlemci bu iki işlem arasına giremez.

Bu işlem sırasında, **TSL komutunu çalıştıran CPU**, **bellek veri yolunu (memory bus) kilitler**.
Böylece başka işlemciler bellek erişimi yapamaz, ta ki komut tamamen bitene kadar.

Sakıncası

Öncelik Terslenmesi (Priority Inversion) Durumu

- **H:** yüksek öncelikli süreç
- **L:** düşük öncelikli süreç

Sleep and Wakeup Sistem Çağrıları

Önceki çözümlerin hepsi **yoğun bekleme (busy waiting)** gerektirir.

Bu tür sorunlardan kaçınmak için, **CPU zamanını boşa harcamadan beklemeyi sağlayan** yeni mekanizmalar geliştirilmiştir.

Sleep:

- Süreci **bloklar (askıya alır)**.
- Yani süreç **uyur** ve CPU artık ona zaman ayırmaz.
- Süreç yalnızca **bir başka süreç tarafından uyandırıldığında** tekrar çalışmaya başlar.

Wakeup:

- **Uyuyan bir süreci yeniden aktif hâle getirir.**
- Tek parametresi vardır: uyandırılacak sürecin kimliği (process ID).

Üretici-Tüketici Problemi (The Producer–Consumer Problem)

Problem:

İki süreç **ortak arabelleği (buffer)** payllaşsın:

- **Üretici (Producer):** Arabelleğe yeni veri (öğe) ekler.
- **Tüketici (Consumer):** Arabellekten veri çıkarır.

Çözüm:

1. **Üretici** yeni bir öğe eklemek ister, ancak **arabellek doludur**
   a. Üretici **uykuya (sleep)** geçmelidir.
   b. Üretici, **tüketici bir öğe alıp yer açtığında** uyandırılır (**wakeup**).

2. **Tüketici**, arabellekten öğe almak ister, ancak **arabellek boştur**
   a. Tüketici **uykuya geçer**.
   b. Tüketici, **üretici bir öğe eklediğinde** uyandırılır.

Bu çözüm **yarış durumu (race condition)** engelleyemez.

Üretici-Tüketici Problemi (The Producer–Consumer Problem)

Sayacı (count) Kullanmak

Arabellekteki öğe sayısını izlemek için bir değişken kullanılır:

int count; /* arabellekteki öğe sayısı */

- Arabelleğin kapasitesi `N` olsun.

- **Üretici**:
  - Eğer `count == N` → arabellek dolu → **sleep()** çağır.
  - Değilse, öğe ekle (`insert_item()`) ve `count` değerini **1 artır**.
  - Eğer `count == 1` olduysa, **tüketiciyi uyandır (wakeup)**.

- **Tüketici**:
  - Eğer `count == 0` → arabellek boş → **sleep()** çağır.
  - Değilse, öğe çıkar (`remove_item()`) ve `count` değerini **1 azalt**.
  - Eğer `count == N-1` olduysa, **üreticiyi uyandır (wakeup)**.

Üretici-Tüketici Problemi (The Producer–Consumer Problem)

Sayacı (count) Kullanmak

```
#define N 100          /* arabellek boyutu */
int count = 0;         /* arabellek içindeki öğe sayısı */
void producer(void) {
    int item;
    while (TRUE) {
        item = produce_item();      /* yeni öğe üret */
        if (count == N) sleep();    /* doluysa uyu */
        insert_item(item);          /* arabelleğe ekle */
        count = count + 1;          /* sayacı artır */
        if (count == 1) wakeup(consumer);
                                    /* arabellek boştuysa, tüketiciyi uyandır */
    }
}
```

```
void consumer(void) {
    int item;
    while (TRUE) {
        if (count == 0) sleep();    /* boşsa uyu */
        item = remove_item();       /* öğeyi al */
        count = count - 1;          /* sayacı azalt */
        if (count == N - 1) wakeup(producer);
                                    /* arabellek doluyduysa, üreticiyi uyandır
*/
        consume_item(item);         /* öğeyi işle */
    }
}
```

Üretici-Tüketici Problemi (The Producer–Consumer Problem)

Sayacı (count) Kullanmak ❌ Ölümcül Yarış Durumu (Fatal Race Condition)

`count` değişkenine erişim **korunmadığı** için, yarış durumu meydana gelebilir.

Örnek:

1. Arabellek **boş** (`count = 0`).
2. **Tüketici** `count`'u okur (0'dır) ve tam `sleep()`'e gitmeden **zamanlayıcı CPU** onu durdurur.
3. **Üretici** çalışmaya başlar:
    - Bir öğe ekler.
    - `count = 1` olur.
    - Üretici, "demek ki tüketici uyuyor" diye düşünerek `wakeup(consumer)` çağırır.
4. Sonra **tüketici** yeniden çalışır, eski `count` değerini (0) görür ve **uykuya geçer.**
5. Üretici ise sonunda tamponu doldurur ve o da **uyur.**
6. Sonuç:
    - Her iki süreç de uyur.
    - Hiçbiri bir daha uyanmaz.
    - Sistem **ölü duruma (deadlock)** girer.

Semafor

1965 yılında E. W. Dijkstra, “uyandırmaların” (wakeups) sayısını tutmak için, “semafor” adını verdiği yeni bir değişken türü tanıttı.

semafor = 0 hiç bir uyandırma yok

semafor > 0 uyandırma/lar beklemede

Semafor Down ve Up İşlemleri

- **down (P)** işlemi:
    - semafor > 0, 1 azaltılır (uyandırma kullanılır)
    - Semafor = 0, işlem yapan süreç uykuya geçirilir ve down işlemi tamamlanmaz.
    - Bu işlem **atomik** (bölünmez) biçimde gerçekleştirilir; yani işlem başladığında başka hiçbir süreç semafora erişemez.
- **up (V)** işlemi:
    - Semaforun değerini 1 artırır.
    - Eğer semaforda bekleyen (uykuda olan) süreçler varsa, sistem tarafından uyandırılır
    - Bu işlem de **atomiktir**; hiçbir süreç up işlemi sırasında bloke olmaz.

Semafor

1. **Binary Semafor (İkili Semafor)**

   Bu tür semaforlar yalnızca iki değere sahip olabilir: 0 veya 1.

   İkili semaforlar, mutex (karşılıklı dışlama) görevi görür.

   Bir iş parçacığı/süreç, kaynağa erişim izni almak için semaforun değerini bekler.

   Değer 0 ise, kaynağa erişim izni verilmez ve thread veya işlem bekler.

Semafor

2. Counting Semafor (Sayısal Semafor)

Bu tür semaforlar, herhangi bir pozitif tam sayı değerine sahip olabilir.

Sayısal semaforlar, belirli bir kaynağın aynı anda kaç iş parçacığı/süreç tarafından kullanılabileceğini kontrol etmek için kullanılır.

Bir süreç kaynağı kullanmaya başladığında, semaforun değeri azaltılır.

Süreç kaynağı bıraktığında, semaforun değeri artırılır.

Semafor İşlemleri

1. **Acquire() / Wait (P) İşlemi**

   Bir iş parçacığı/süreç, kaynağa erişim izni almak için semaforun değerini azaltır.
   Eğer semafor değeri 0 ise, iş parçacığı/süreç beklemeye alınır.

2. **release() / Signal (V) İşlemi**

   Bir iş parçacığı/süreç, kaynağı kullanımı bıraktığında semaforun değerini artırır.
   Bu, diğer bekleyen tiş parçacığı/süreçlere kaynağa erişim izni verir.

Semafor Örnek 1: Binary Semafor

İkili semafor, bir kaynağın tek bir iş parçacığı/süreç tarafından aynı anda kullanılmasını sağlamak için kullanılır.

Örneğin, bir yazıcıya aynı anda sadece bir kullanıcının yazdırmasına izin vermek için ikili semafor kullanabilirsiniz.

Semafor Örnek 1: Binary Semafor

```python
import threading
# İkili semafor oluşturuldu, başlangıç değeri 1.
binary_semaphore = threading.Semaphore( 1)

def access_resource ():
    # Kaynağa erişim izni alınır (Wait işlemi).
    binary_semaphore.acquire()
    print("Resource accessed." )
    # Kaynağı kullanmak için gerekli işlemler yapılır.
    binary_semaphore.release()
    # Kaynağa erişim izni serbest bırakılır (Signal işlemi).

# İki thread, aynı anda kaynağa erişmek isterse, biri bekler.
thread1 = threading.Thread( target=access_resource)
thread2 = threading.Thread( target=access_resource)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

`Semaphore(1)` → Başlangıç değeri 1 olan bir semafor oluşturur

```python
def access_resource():
```

`acquire()` (Kilit alma):

- Semafor değeri > 0 ise → Değeri 1 azaltır ve devam eder
- Semafor değeri = 0 ise → Thread **bloke olur** ve bekler

`release()` (Kilidi serbest bırakma):

- Semafor değerini 1 artırır
- Bekleyen başka bir thread varsa, o thread uyandırılır

Semafor Örnek 1: Binary Semafor

```python
import threading
# İkili semafor oluşturuldu, başlangıç değeri 1.
binary_semaphore = threading.Semaphore( 1)

def access_resource ():
    # Kaynağa erişim izni alınır (Wait işlemi).
    binary_semaphore.acquire()
    print ("Resource accessed." )
    # Kaynağı kullanmak için gerekli işlemler yapılır.
    binary_semaphore.release()
    # Kaynağa erişim izni serbest bırakılır (Signal işlemi).

# İki thread, aynı anda kaynağa erişmek isterse, biri bekler.
thread1 = threading.Thread( target=access_resource)
thread2 = threading.Thread( target=access_resource)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

3. Thread'lerin Olası Çalışma Senaryosu:

1. `thread1` çalışır → `acquire()` yapar (semafor: 1→0) → Kaynağa erişir
2. `thread2` çalışır → `acquire()` yapmak ister ama **semafor = 0** → BEKLER
3. `thread1` işini bitirir → `release()` yapar (semafor: 0→1)
4. `thread2` uyandırılır → Kaynağa erişir
5. `thread2` işini bitirir → `release()` yapar

4. Thread'lerin Bitmesini Bekleme

thread1.join()
thread2.join()

Çıktı

Resource accessed.
Resource accessed.

Semafor Örnek 1: Binary Semafor

```python
from threading import Semaphore, Thread
import time

printer_semaphore = Semaphore( 1)

def print_document (user):
   printer_semaphore.acquire()
   print (f"{user}is printing...")
   time.sleep( 1)
   print (f"{user}is printing...")
   printer_semaphore.release()

user1 = Thread( target=print_document,  args=("User 1",))
user2 = Thread( target=print_document,  args=("User 2",))

user1.start()
user2.start()

user1.join()
user2.join()
```

1. t=0.0s → User 1 başlar, yazıcıyı kilitler (semafor: 1→0)
2. t=0.0s → User 2 başlar, yazıcı kilitli olduğu için**BEKLER**
3. t=1.0s → User 1 işini bitirir, yazıcıyı serbest bırakır (semafor: 0→1)
4. t=1.0s → User 2 artık yazıcıyı kullanabilir
5. t=2.0s → User 2 işini bitirir

Olası Çıktı:

```
User 1 is printing...
User 1 finished printing.
User 2 is printing...
User 2 finished printing.
```

Semafor Örnek 1: Counting Semafor

Sayısal semaforlar, birden fazla kaynağın aynı anda erişimini kontrol etmek için kullanılabilir.

Örneğin, bir havuzdaki yüzme kulvarlarına aynı anda sadece belirli bir sayıda kişinin girmesine izin vermek için sayısal semafor kullanabilirsiniz.

Semafor Örnek 1: Counting Semafor

```
from threading import Semaphore, Thread
import time

# Sayısal semafor oluşturuldu, başlangıç değeri 2
pool_semaphore = Semaphore( 2)

def swimmer(name):
   print(f"{name} is waiting to enter the pool..." )
   pool semaphore.acquire()   # Kaynağa erişim izni alınır (Wait işlemi)
   print(f"{name} is swimming in the pool." )
   time.sleep( 2)  # Yüzme işlemini simüle eder
   print(f"{name} finished swimming." )
   pool_semaphore.release()   # Kaynağa erişim izni serbest bırakılır (Signal
işlemi)

# Üç thread, aynı anda kaynağa erişmek isterse, ikisi erişebilir, biri bekler
swimmer1 = Thread( target=swimmer, args=("Swimmer 1",))
swimmer2 = Thread( target=swimmer, args=("Swimmer 2",))
swimmer3 = Thread( target=swimmer, args=("Swimmer 3",))

swimmer1.start()
swimmer2.start()
swimmer3.start()

swimmer1.join()
swimmer2.join()
swimmer3.join()

print("\nAll swimmers have finished!" )
```

t = 0.01s - Swimmer 1 Aktif

pool_semaphore.acquire() # Semafor: 2 → 1

- **Swimmer 1** havuza girmek için semafordan izin ister
- Semafor > 0 olduğu için **hemen izin verilir**
- **Semafor değeri: 2 → 1** (1 yer kaldı)

t = 0.02s - Swimmer 2 Aktif

pool_semaphore.acquire() # Semafor: 1 → 0

- **Swimmer 2** havuza girmek ister
- Semafor hala > 0 (1 yer var)
- **Semafor değeri: 1 → 0** (havuz dolu!)

Semafor Örnek 1: Counting Semafor

```python
from threading import Semaphore, Thread
import time

# Sayısal semafor oluşturuldu, başlangıç değeri 2
pool_semaphore = Semaphore( 2)

def swimmer(name):
    print(f"{name} is waiting to enter the pool..." )
    pool semaphore.acquire()    # Kaynağa erişim izni alınır (Wait işlemi)
    print(f"{name} is swimming in the pool." )
    time.sleep( 2)  # Yüzme işlemini simüle eder
    print(f"{name} finished swimming." )
    pool_semaphore.release()    # Kaynağa erişim izni serbest bırakılır (Signal
işlemi)

# Üç thread, aynı anda kaynağa erişmek isterse, ikisi erişebilir, biri bekler
swimmer1 = Thread( target=swimmer, args=("Swimmer 1",))
swimmer2 = Thread( target=swimmer, args=("Swimmer 2",))
swimmer3 = Thread( target=swimmer, args=("Swimmer 3",))

swimmer1.start()
swimmer2.start()
swimmer3.start()

swimmer1.join()
swimmer2.join()
swimmer3.join()

print("\nAll swimmers have finished!" )
```

t = 0.03s - Swimmer 3 BEKLEMEDE! 🚫

pool_semaphore.acquire() # Semafor = 0 → BLOKE OLDU!

- **Swimmer 3** havuza girmek ister
- **ANCAK** semafor = 0 (havuz dolu)
- Thread **bloke olur** ve BEKLER ⏸️
- Bir yer boşalana kadar burada kalır!

t = 2.01s - Swimmer 1 Çıkıyor

pool_semaphore.release() # Semafor: 0 → 1

- **Swimmer 1**'in 2 saniyesi doldu
- Havuzdan çıkıyor
- **Semafor değeri: 0 → 1** (1 yer açıldı!)
- İşletim sistemi **bekleyen thread'i uyandırır** → Swimmer 3 artık girebilir! ✅

Semafor Örnek 1: Counting Semafor

```python
from threading import Semaphore, Thread
import time

# Sayısal semafor oluşturuldu, başlangıç değeri 2
pool_semaphore = Semaphore( 2)

def swimmer(name):
    print(f"{name} is waiting to enter the pool..." )
    pool semaphore.acquire()    # Kaynağa erişim izni alınır (Wait işlemi)
    print(f"{name} is swimming in the pool." )
    time.sleep( 2)  # Yüzme işlemini simüle eder
    print(f"{name} finished swimming." )
    pool_semaphore.release()    # Kaynağa erişim izni serbest bırakılır (Signal
işlemi)

# Üç thread, aynı anda kaynağa erişmek isterse, ikisi erişebilir, biri bekler
swimmer1 = Thread( target=swimmer, args=("Swimmer 1",))
swimmer2 = Thread( target=swimmer, args=("Swimmer 2",))
swimmer3 = Thread( target=swimmer, args=("Swimmer 3",))

swimmer1.start()
swimmer2.start()
swimmer3.start()

swimmer1.join()
swimmer2.join()
swimmer3.join()

print("\nAll swimmers have finished!" )
```

t = 2.01s - Swimmer 1 Çıkıyor

pool_semaphore.release() # Semafor: 0 → 1

- **Swimmer 1**'in 2 saniyesi doldu
- Havuzdan çıkıyor
- **Semafor değeri: 0 → 1** (1 yer açıldı!)
- İşletim sistemi **bekleyen thread'i uyandırır** → Swimmer 3 artık girebilir! ✅

t = 2.01s - Swimmer 3 Giriyor

pool_semaphore.acquire() # Artık izin alabilir! Semafor: 1 → 0

time.sleep(2) # 2 saniye yüzecek

- Bekleyen **Swimmer 3** uyandırıldı
- Havuza giriş yapıyor
- **Semafor değeri: 1 → 0**

Semafor Örnek 1: Counting Semafor

```
from threading import Semaphore, Thread
import time

# Sayısal semafor oluşturuldu, başlangıç değeri 2
pool_semaphore = Semaphore( 2)

def swimmer(name):
   print(f"{name} is waiting to enter the pool..." )
   pool semaphore.acquire()   # Kaynağa erişim izni alınır (Wait işlemi)
   print(f"{name} is swimming in the pool." )
   time.sleep( 2)  # Yüzme işlemini simüle eder
   print(f"{name} finished swimming." )
   pool_semaphore.release()   # Kaynağa erişim izni serbest bırakılır (Signal
işlemi)

# Üç thread, aynı anda kaynağa erişmek isterse, ikisi erişebilir, biri bekler
swimmer1 = Thread( target=swimmer, args=("Swimmer 1",))
swimmer2 = Thread( target=swimmer, args=("Swimmer 2",))
swimmer3 = Thread( target=swimmer, args=("Swimmer 3",))

swimmer1.start()
swimmer2.start()
swimmer3.start()

swimmer1.join()
swimmer2.join()
swimmer3.join()

print("\nAll swimmers have finished!" )
```

t = 2.02s - Swimmer 2 Çıkıyor

pool_semaphore.release() # Semafor: 0 → 1

- **Swimmer 2**'nin de 2 saniyesi doldu
- Havuzdan çıkıyor
- **Semafor değeri: 0 → 1**
- Ama bekleyen kimse yok artık

t = 4.01s - Swimmer 3 Çıkıyor

pool_semaphore.release() # Semafor: 1 → 2

- **Swimmer 3** işini bitirdi
- **Semafor değeri: 1 → 2** (başlangıç haline döndü)

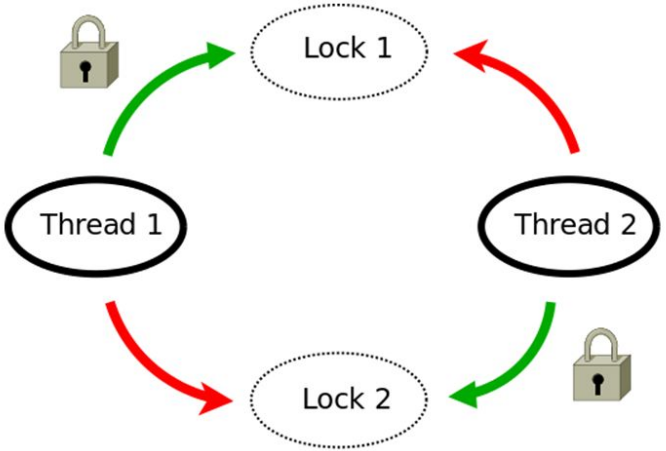
Mutex

Semafor, sayma özelliğine ihtiyaç duyulmadığında, **mutex** (karşılıklı dışlama kilidi) adı verilen basitleştirilmiş bir versiyonu bazen kullanılır.

Bir mutex, **kilitli** veya **kilitsiz** durumunda olabilen paylaşılan bir değişkendir.

`0` → kilitsiz

`=!0` → kilitli



Mutex

1. **Lock (Kilitleme) İşlemi:**

   Bir iş parçacığı/süreç, belirli bir kaynağa erişmek istediğinde mutex'i kilitleyebilir.

   Eğer mutex zaten kilitliyse, işlem beklemeye alınır.

   Eğer mutex serbestse, iş parçacığı/süreç mutex'i kilitleyerek kaynağa erişim izni alır.

2. **Unlock (Kilidi Açma) İşlemi**

   Bir iş parçacığı/süreç, kaynağı kullanımı bıraktığında semaforun değerini artırır.

   Bu, diğer bekleyen iş parçacığı/süreçlere kaynağa erişim izni verir.

Mutex Örnek 1: Dosya İşleme

```
from threading import Thread, Lock

file mutex = Lock()
file_contents = []

def write to file (data):
   with file mutex:
       file contents.append(data)
       print (f"Written: {data}")

def read from file ():
   with file mutex:
       return file_contents.copy()
# Yazma işlemleri
thread1 = Thread( target=write_to_file, args=("Data from Thread
1",))
thread2 = Thread( target=write_to_file, args=("Data from Thread
2",))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("\n--- Yazma işlemleri tamamlandı --- \n")

# Okuma işlemleri
def read thread ():
   data = read from file()      print (f"Read: {data}")
read thread1 = Thread( target=read thread)
read thread2 = Thread( target=read_thread)
read thread1.start()
read thread2.start()
read thread1.join()
read thread2.join()
print ("\n--- Tüm işlemler tamamlandı ---" )
```

file_mutex = Lock() # Mutex (kilit) oluşturuldu

file_contents = [] # Paylaşılan veri yapısı (dosya simülasyonu)

t = 0.01s - Thread 1 Kilidi Alıyor

def write_to_file(data):

with file_mutex: # Thread 1 mutex'i KILITLEDI

- **Thread 1** mutex'i alır (acquire)
- Mutex **kilitli** hale gelir
- Thread 2 bu noktada **beklemek zorunda**

with bloğu bitince mutex OTOMATİK serbest bırakılır

t = 0.02s - Thread 2 Sırasını Bekliyor

def write_to_file(data):

with file_mutex: # Thread 2 mutex almaya çalışıyor...

- **Thread 2** mutex'i almak ister
- ANCAK Thread 1 hala kilidi tutuyor
- **Thread 2 BLOKE OLUR** (bekler)

Mutex Örnek 1: Dosya İşleme

```
from threading import Thread, Lock

file mutex = Lock()
file_contents = []

def write to file (data):
   with file mutex:
       file contents.append(data)
       print (f"Written: {data}")

def read from file ():
   with file mutex:
       return file_contents.copy()
# Yazma işlemleri
thread1 = Thread( target=write_to_file, args=("Data from Thread
1",))
thread2 = Thread( target=write_to_file, args=("Data from Thread
2",))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("\n--- Yazma işlemleri tamamlandı --- \n")

# Okuma işlemleri
def read thread ():
   data = read from file()      print (f"Read: {data}")
read thread1 = Thread( target=read thread)
read thread2 = Thread( target=read_thread)
read thread1.start()
read thread2.start()
read thread1.join()
read thread2.join()
print ("\n--- Tüm işlemler tamamlandı ---" )
```

file_mutex = Lock() # Mutex (kilit) oluşturuldu

file_contents = [] # Paylaşılan veri yapısı (dosya simülasyonu)

t = 0.01s - Thread 1 Kilidi Alıyor

def write_to_file(data):

with file_mutex: # Thread 1 mutex'i KILITLEDI

- **Thread 1** mutex'i alır (acquire)
- Mutex **kilitli** hale gelir
- Thread 2 bu noktada **beklemek zorunda**

with bloğu bitince mutex OTOMATİK serbest bırakılır

t = 0.03s - Thread 1 Kilidi Serbest Bırakıyor

Thread 1'in with bloğu bitti → mutex serbest!

- Thread 1 işini bitirdi
- Mutex otomatik serbest bırakıldı
- İşletim sistemi **bekleyen Thread 2'yi uyandırır**

Mutex Örnek 1: Dosya İşleme

```
from threading import Thread, Lock

file mutex = Lock()
file_contents = []

def write to file (data):
   with file mutex:
       file contents.append(data)
       print(f"Written: {data}")

def read from file ():
   with file mutex:
       return file_contents.copy()
# Yazma işlemleri
thread1 = Thread( target=write_to_file, args=("Data from Thread
1",))
thread2 = Thread( target=write_to_file, args=("Data from Thread
2",))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print("\n--- Yazma işlemleri tamamlandı --- \n")

# Okuma işlemleri
def read thread ():
   data = read from file()      print(f"Read: {data}")
read thread1 = Thread( target=read thread)
read thread2 = Thread( target=read_thread)
read thread1.start()
read thread2.start()
read thread1.join()
read thread2.join()
print("\n--- Tüm işlemler tamamlandı ---" )
```

t = 0.04s - Thread 2 Kilidi Alıyor

```
def write_to_file(data):
   with file_mutex:  # Thread 2 artık mutex'i aldı
      file_contents.append(data)
      # file_contents = ["Data from Thread 1", "Data from Thread 2"]

   # with bloğu bitince mutex serbest
```

t = 0.07s - Read Thread 1 Kilidi Alıyor

```
def read_from_file():
   with file_mutex:  # Read Thread 1 mutex'i aldı
      return file_contents.copy()
      # ["Data from Thread 1", "Data from Thread 2"] kopyası döndü
```

- **Read Thread 1** mutex'i alır
- `file_contents.copy()` ile liste kopyası oluşturur
- Mutex serbest bırakılır

Mutex Örnek 1: Dosya İşleme

```
from threading import Thread, Lock

file mutex = Lock()
file_contents = []

def write to file (data):
   with file mutex:
       file contents.append(data)
       print (f"Written: {data}")

def read from file ():
   with file mutex:
       return file_contents.copy()
# Yazma işlemleri
thread1 = Thread( target=write_to_file, args=("Data from Thread
1",))
thread2 = Thread( target=write_to_file, args=("Data from Thread
2",))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("\n--- Yazma işlemleri tamamlandı --- \n")

# Okuma işlemleri
def read thread ():
   data = read from file()      print (f"Read: {data}")
read thread1 = Thread( target=read thread)
read thread2 = Thread( target=read_thread)
read thread1.start()
read thread2.start()
read thread1.join()
read thread2.join()
print ("\n--- Tüm işlemler tamamlandı ---" )
```

t = 0.08s - Read Thread 2 Sırasını Bekliyor (Kısa Süre)

```
def read_from_file():
   with file_mutex:  # Read Thread 2 mutex almaya çalışıyor
```

- **Read Thread 2** mutex'i almak ister
- Eğer Read Thread 1 hala tutuyorsa **bekler**
- Read Thread 1 hızlıca bittiği için çok kısa bekler (veya hiç beklemez)

t = 0.09s - Read Thread 2 Kilidi Alıyor

```
def read_from_file():
   with file_mutex:  # Read Thread 2 mutex'i aldı
      return file_contents.copy()
      # ["Data from Thread 1", "Data from Thread 2"] kopyası döndü
```

Mutex Örnek 1: Dosya İşleme

```
from threading import Thread, Lock

file mutex = Lock()
file_contents = []

def write to file (data):
   with file mutex:
       file contents.append(data)
       print(f"Written: {data}")

def read from file ():
   with file mutex:
       return file_contents.copy()
# Yazma işlemleri
thread1 = Thread( target=write_to_file, args=("Data from Thread
1",))
thread2 = Thread( target=write_to_file, args=("Data from Thread
2",))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print("\n--- Yazma işlemleri tamamlandı --- \n")

# Okuma işlemleri
def read thread ():
   data = read from file()      print(f"Read: {data}")
read thread1 = Thread( target=read thread)
read thread2 = Thread( target=read_thread)
read thread1.start()
read thread2.start()
read thread1.join()
read thread2.join()
print("\n--- Tüm işlemler tamamlandı ---" )
```

Mutex Kilit Mekanizması

Thread 1: acquire() → KRİTİK BÖLGE → release()

↓

Thread 2: BEKLER → acquire() → KRİTİK BÖLGE → release()

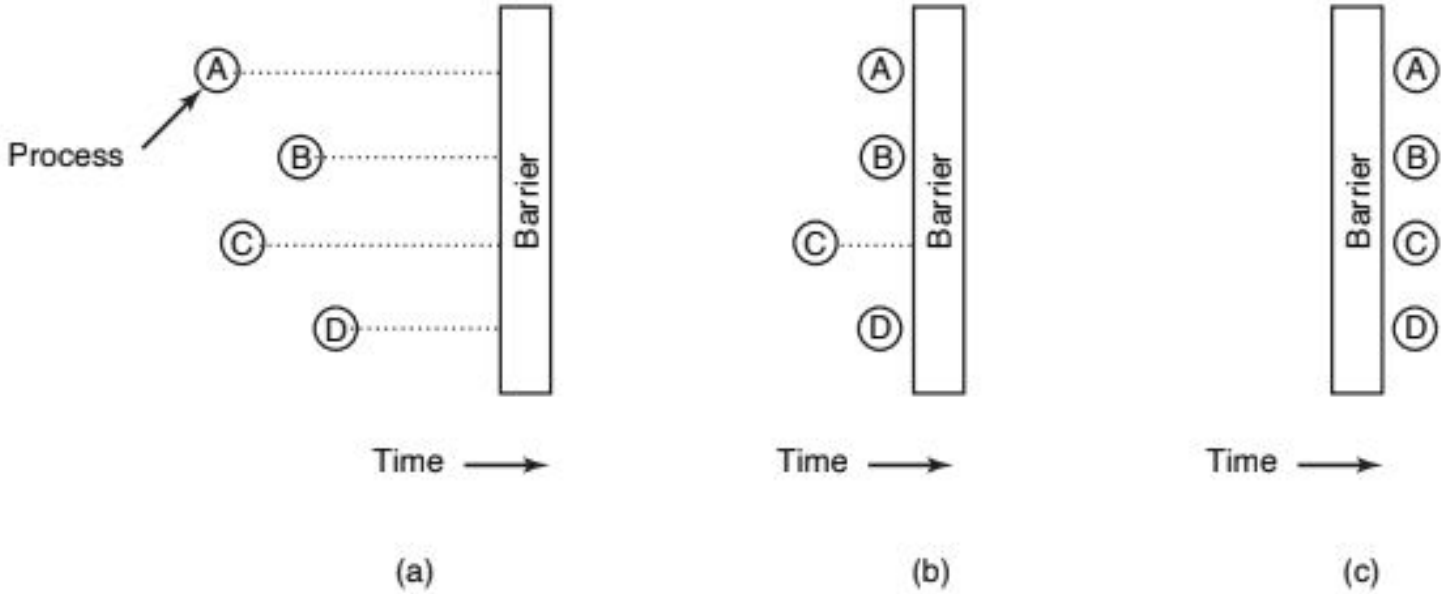
Bu sayede **race condition** (yarış durumu) ve **veri tutarsızlığı** önlenir!

Mutex Örnek 1: Kaynak Paylaşımı

```python
from threading import Thread, Lock
shared_resource = []
mutex = Lock()

def append_to_shared(data):
    with mutex:
        shared_resource.append(data)
        print(f"Appended: {data}")

def remove_from_shared():
    with mutex:
        if shared_resource:
            return shared_resource.pop()
        else:
            return None

# FAZE 1: Veri Ekleme
print("=== Veri Ekleme Başlıyor ===")
thread1 = Thread(target=append_to_shared, args=("Data 1",))
thread2 = Thread(target=append_to_shared, args=("Data 2",))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print(f"\nMevcut veriler: {shared_resource}\n")

# FAZE 2: Veri Kullanma
print("=== Veri Kullanma Başlıyor ===")
def use_shared():
    data = remove_from_shared()
    if data:
        print(f"Used: {data}")
    else:
        print("No data available")

use_thread1 = Thread(target=use_shared)
use_thread2 = Thread(target=use_shared)
use_thread1.start()
use_thread2.start()
use_thread1.join()
use_thread2.join()
print(f"\nKalan veriler: {shared_resource}")
print("\n=== Tüm İşlemler Tamamlandı ===")
```

```
=== Veri Ekleme Başlıyor ===
Appended: Data 1
Appended: Data 2

Mevcut veriler: ['Data 1', 'Data 2']

=== Veri Kullanma Başlıyor ===
Used: Data 2
Used: Data 1

Kalan veriler: []

=== Tüm İşlemler Tamamlandı ===
```

Barriers (Engelleyiciler)



Barriers (Engelleyiciler)

1. **Wait (Bekleme) İşlemi**

   Her iş parçacığı/süreç, belirli bir noktada Wait işlemi yaparak barrier'a katılır.

   Barrier, beklenen sayıda katılımcıya ulaştığında, tüm katılımcıların devam etmesine izin verir.

2. **Reset (Sıfırlama) İşlemi:**

   Barrier, tüm katılımcılar devam ettikten sonra sıfırlanır ve bir sonraki kullanımda tekrar kullanılabilir hale gelir.

Barriers Örnek 1: Yarış

```python
from threading import Thread, Barrier
import time

# 5 katılımcı olması bekleniyor
barrier = Barrier( 5)

def runner(name):
    print(f"{name} is ready and waiting at the starting line..." )
    time.sleep( 0.1)  # Koşucuların farklı zamanlarda gelmesini simüle eder
    print(f"{name} reached the barrier." )
    barrier.wait()   # Tüm koşucuların hazır olmasını bekle
    print(f"{name} started running!  🏃")

print("=== Race is starting ===" )
print("Waiting for all 5 runners to be ready... \n")

# 5 koşucu oluşturuluyor
runners = []
for i in range(1, 6):
    runner thread = Thread( target=runner, args=(f"Runner {i}",))
    runners.append(runner thread)
    runner_thread.start()

# Tüm koşucuların bitmesini bekle
for runner thread in runners:
    runner_thread.join()

print("\n=== All runners have started! ===" )
```

t = 0.01s - Runner 1 Başlıyor

python
```
for i in range(1, 6):  # i = 1
    runner_thread = Thread(target=runner, args=(f"Runner 1",))
    runners.append(runner_thread)

        runner_thread.start()  # Runner 1 thread'i başladı
```

Runner 1 fonksiyonu çalışıyor:

python
```
def runner(name):  # name = "Runner 1"
    print(f"{name} is ready and waiting at the starting line...")
    # Çıktı: "Runner 1 is ready and waiting at the starting line..."

    time.sleep(0.1)  # 0.1 saniye bekle (yavaşça gelmeyi simüle eder)

    print(f"{name} reached the barrier.")
    # Çıktı: "Runner 1 reached the barrier."

    barrier.wait()  # 🚫 BARRIER'DA BLOKE OLDU!
```

Barrier Durumu:
Barrier kapasitesi: 5
Bekleyen thread: 1 (Runner 1 ⏸️)

Durum: 🚫 KAPALI (henüz 5 değil)

Barriers Örnek 1: Yarış

```python
from threading import Thread, Barrier
import time

# 5 katılımcı olması bekleniyor
barrier = Barrier( 5)

def runner(name):
    print(f"{name} is ready and waiting at the starting line..." )
    time.sleep( 0.1)  # Koşucuların farklı zamanlarda gelmesini simüle eder
    print(f"{name} reached the barrier." )
    barrier.wait()   # Tüm koşucuların hazır olmasını bekle
    print(f"{name} started running!  🏃")

print("=== Race is starting ===" )
print("Waiting for all 5 runners to be ready... \n")

# 5 koşucu oluşturuluyor
runners = []
for i in range(1, 6):
    runner thread = Thread( target=runner, args=(f"Runner {i}",))
    runners.append(runner thread)
    runner_thread.start()

# Tüm koşucuların bitmesini bekle
for runner thread in runners:
    runner_thread.join()

print("\n=== All runners have started! ===" )
```

t = 0.02s - Runner 2 Başlıyor

python
for i in range(1, 6): *# i = 2*
runner_thread = Thread(target=runner, args=(f"Runner 2",))

runner_thread.start()

Runner 2 fonksiyonu:

python
print(f"{name} is ready and waiting at the starting line...")
 # Çıktı: "Runner 2 is ready and waiting at the starting line..."

time.sleep(0.1)

print(f"{name} reached the barrier.")
 # Çıktı: "Runner 2 reached the barrier."

barrier.wait() # 🚫 *BARRIER'DA BLOKE OLDU!*
```

**Barrier Durumu:**
```
Barrier kapasitesi: 5
Bekleyen thread: 2 (Runner 1 ⏸️, Runner 2 ⏸️)

Durum: 🚫 KAPALI

Barriers Örnek 1: Yarış

```python
from threading import Thread, Barrier
import time

# 5 katılımcı olması bekleniyor
barrier = Barrier(5)

def runner(name):
    print(f"{name} is ready and waiting at the starting line...")
    time.sleep(0.1)  # Koşucuların farklı zamanlarda gelmesini simüle eder
    print(f"{name} reached the barrier.")
    barrier.wait()   # Tüm koşucuların hazır olmasını bekle
    print(f"{name} started running! 🏃")

print("=== Race is starting ===")
print("Waiting for all 5 runners to be ready... \n")

# 5 koşucu oluşturuluyor
runners = []
for i in range(1, 6):
    runner_thread = Thread(target=runner, args=(f"Runner {i}",))
    runners.append(runner_thread)
    runner_thread.start()

# Tüm koşucuların bitmesini bekle
for runner_thread in runners:
    runner_thread.join()

print("\n=== All runners have started! ===")
```

t = 0.03s - Runner 3 Başlıyor

python

 # Runner 3 aynı şekilde başlıyor

Runner 3 fonksiyonu:

python

```
print(f"Runner 3 is ready and waiting at the starting line...")
# Çıktı: "Runner 3 is ready and waiting at the starting line..."

time.sleep(0.1)

print(f"Runner 3 reached the barrier.")
# Çıktı: "Runner 3 reached the barrier."

barrier.wait()  # 🚫 BARRIER'DA BLOKE OLDU!
```

Barrier Durumu:

```
Barrier kapasitesi: 5
Bekleyen thread: 3 (Runner 1 ⏸️, Runner 2 ⏸️, Runner 3 ⏸️)
    Durum: 🚫 KAPALI
```

Barriers Örnek 1: Yarış

```python
from threading import Thread, Barrier
import time

# 5 katılımcı olması bekleniyor
barrier = Barrier( 5)

def runner(name):
    print(f"{name} is ready and waiting at the starting line..." )
    time.sleep( 0.1)  # Koşucuların farklı zamanlarda gelmesini simüle eder
    print(f"{name} reached the barrier." )
    barrier.wait()   # Tüm koşucuların hazır olmasını bekle
    print(f"{name} started running!  🏃")

print("=== Race is starting ===" )
print("Waiting for all 5 runners to be ready... \n")

# 5 koşucu oluşturuluyor
runners = []
for i in range(1, 6):
    runner thread = Thread( target=runner, args=(f"Runner {i}",))
    runners.append(runner thread)
    runner_thread.start()

# Tüm koşucuların bitmesini bekle
for runner thread in runners:
    runner_thread.join()

print("\n=== All runners have started! ===" )
```

t = 0.04s - Runner 4 Başlıyor

python

 # Runner 4 aynı şekilde başlıyor

Runner 4 fonksiyonu:

python

```
print(f"Runner 4 is ready and waiting at the starting line...")
# Çıktı: "Runner 4 is ready and waiting at the starting line..."

time.sleep(0.1)

print(f"Runner 4 reached the barrier.")
# Çıktı: "Runner 4 reached the barrier."

barrier.wait()  # 🚫 BARRIER'DA BLOKE OLDU!
```

Barrier Durumu:

```
Barrier kapasitesi: 5
Bekleyen thread: 4 (Runner 1-4 hepsi ⏸️)
    Durum: 🚫 KAPALI (1 kişi daha lazım!)
```

Barriers Örnek 1: Yarış

```python
from threading import Thread, Barrier
import time

# 5 katılımcı olması bekleniyor
barrier = Barrier( 5)

def runner(name):
    print(f"{name} is ready and waiting at the starting line..." )
    time.sleep( 0.1)  # Koşucuların farklı zamanlarda gelmesini simüle eder
    print(f"{name} reached the barrier." )
    barrier.wait()   # Tüm koşucuların hazır olmasını bekle
    print(f"{name} started running!  🏃")

print("=== Race is starting ===" )
print("Waiting for all 5 runners to be ready... \n")

# 5 koşucu oluşturuluyor
runners = []
for i in range(1, 6):
    runner thread = Thread( target=runner, args=(f"Runner {i}",))
    runners.append(runner thread)
    runner_thread.start()

# Tüm koşucuların bitmesini bekle
for runner thread in runners:
    runner_thread.join()

print("\n=== All runners have started! ===" )
```

t = 0.05s - Runner 5 Başlıyor (KRİTİK NOKTA!)

python

 # Runner 5 başlıyor

Runner 5 fonksiyonu:

python

print(f"Runner 5 is ready and waiting at the starting line...")

 # Çıktı: "Runner 5 is ready and waiting at the starting line..."

time.sleep(0.1)

print(f"Runner 5 reached the barrier.")

 # Çıktı: "Runner 5 reached the barrier."

barrier.wait() # ✅ *5. KİŞİ GELDİ!*

```

**🎆 BARRİER AÇILDI!**

```

Barrier kapasitesi: 5

Bekleyen thread: 5 (Tam kapasitede!)

Durum: ✅ AÇILDI! (Hepsi serbest!)

Barriers Örnek 1: Yarış

```python
from threading import Thread, Barrier
import time

# 5 katılımcı olması bekleniyor
barrier = Barrier( 5)

def runner(name):
    print(f"{name} is ready and waiting at the starting line..." )
    time.sleep( 0.1)  # Koşucuların farklı zamanlarda gelmesini simüle eder
    print(f"{name} reached the barrier." )
    barrier.wait()   # Tüm koşucuların hazır olmasını bekle
    print(f"{name} started running!  🏃")

print("=== Race is starting ===" )
print("Waiting for all 5 runners to be ready... \n")

# 5 koşucu oluşturuluyor
runners = []
for i in range(1, 6):
    runner thread = Thread( target=runner, args=(f"Runner {i}",))
    runners.append(runner thread)
    runner_thread.start()

# Tüm koşucuların bitmesini bekle
for runner thread in runners:
    runner_thread.join()

print("\n=== All runners have started! ===" )
```

t = 0.15s - TÜM RUNNER'LAR AYNI ANDA DEVAM EDİYOR!

python

```
# Runner 1:
barrier.wait()  # BLOKE değil artık, geçti! ✅
print(f"Runner 1 started running! 🏃")

# Runner 2:
barrier.wait()  # BLOKE değil artık, geçti! ✅
print(f"Runner 2 started running! 🏃")

# Runner 3:
barrier.wait()  # BLOKE değil artık, geçti! ✅
print(f"Runner 3 started running! 🏃")

# Runner 4:
barrier.wait()  # BLOKE değil artık, geçti! ✅
print(f"Runner 4 started running! 🏃")

# Runner 5:
barrier.wait()  # BLOKE değil artık, geçti! ✅
print(f"Runner 5 started running! 🏃")
```

Barriers Örnek 1: Yarış

```python
from threading import Thread, Barrier
import time

# 5 katılımcı olması bekleniyor
barrier = Barrier( 5)

def runner(name):
    print(f"{name} is ready and waiting at the starting line..." )
    time.sleep( 0.1)  # Koşucuların farklı zamanlarda gelmesini simüle eder
    print(f"{name} reached the barrier." )
    barrier.wait()   # Tüm koşucuların hazır olmasını bekle
    print(f"{name} started running!  🏃")

print("=== Race is starting ===" )
print("Waiting for all 5 runners to be ready... \n")

# 5 koşucu oluşturuluyor
runners = []
for i in range(1, 6):
    runner thread = Thread( target=runner, args=(f"Runner {i}",))
    runners.append(runner thread)
    runner_thread.start()

# Tüm koşucuların bitmesini bekle
for runner thread in runners:
    runner_thread.join()

print("\n=== All runners have started! ===" )
```

```
**Çıktı (neredeyse eşzamanlı):**
```

```
Runner 1 started running! 🏃
Runner 2 started running! 🏃
Runner 3 started running! 🏃
Runner 4 started running! 🏃
Runner 5 started running! 🏃
```

Barrier Durumu:

```
Barrier kapasitesi: 5
Bekleyen thread: 0 (hepsi geçti)
```

Durum: 🔄 SIFIRLANDI (tekrar kullanıma hazır)

Barriers Örnek 1: Yarış

```python
from threading import Thread, Barrier
import time

# 5 katılımcı olması bekleniyor
barrier = Barrier( 5)

def runner(name):
    print(f"{name} is ready and waiting at the starting line..." )
    time.sleep( 0.1)  # Koşucuların farklı zamanlarda gelmesini simüle eder
    print(f"{name} reached the barrier." )
    barrier.wait()   # Tüm koşucuların hazır olmasını bekle
    print(f"{name} started running!  🏃")

print("=== Race is starting ===" )
print("Waiting for all 5 runners to be ready... \n")

# 5 koşucu oluşturuluyor
runners = []
for i in range(1, 6):
    runner thread = Thread( target=runner, args=(f"Runner {i}",))
    runners.append(runner thread)
    runner_thread.start()

# Tüm koşucuların bitmesini bekle
for runner thread in runners:
    runner_thread.join()

print("\n=== All runners have started! ===" )
```

ÇIKTI:

```
=== Race is starting ===
Waiting for all 5 runners to be ready...

Runner 1 is ready and waiting at the starting line...
Runner 1 reached the barrier.
Runner 2 is ready and waiting at the starting line...
Runner 2 reached the barrier.
Runner 3 is ready and waiting at the starting line...
Runner 3 reached the barrier.
Runner 4 is ready and waiting at the starting line...
Runner 4 reached the barrier.
Runner 5 is ready and waiting at the starting line...
Runner 5 reached the barrier.
Runner 1 started running! 🏃
Runner 2 started running! 🏃
Runner 3 started running! 🏃
Runner 4 started running! 🏃
Runner 5 started running! 🏃

=== All runners have started! ===
```

Barriers Örnek 1: Üretici-Tüketici Problemi

Üretici-tüketici problemi:

Bir üretici işlemi ile bir tüketici işlemi arasındaki senkronizasyon gerektiren bir senaryoyu modellemek için kullanılabilir.

Bu örnekte, bir barrier, üretici ve tüketici işlemlerin senkronizasyonunu sağlar.

Barriers Örnek 1: Üretici-Tüketici Problemi

```python
from threading import Thread, Barrier, Lock
import time

# 3 thread için barrier (1 producer + 2 consumer)
barrier = Barrier(3)
shared_resource = []
resource_lock = Lock() # Liste erişimi için güvenlik

def producer():
    for i in range(5):
        # Veri üret
        with resource_lock:
            shared_resource.append(f"Item {i}")
            print(f"Producer: Produced Item{i}")
        # Tüm thread'lerin senkronize olmasını bekle
        print(f"Producer: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"Producer: Passed barrier (round{i})")
        time.sleep(0.1)  # Küçük gecikme ekle

def consumer(name):
    for i in range(5):
        # Producer'ın üretmesini bekle
        print(f"{name}: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"{name}: Passed barrier (round{i})")
        # Veriyi tüket
        with resource_lock:
            if shared_resource:
                item = shared_resource.pop(0)  # FIFO için pop(0)
                print(f"{name}: Consumed {item}")
            else:
                print(f"{name}: No item available")
        time.sleep(0.1)
```

```python
print("=== Producer-Consumer with Barrier ===\n")
producer_thread = Thread(target=producer)
consumer_thread1 = Thread(target=consumer, args=("Consumer 1",))
consumer_thread2 = Thread(target=consumer, args=("Consumer 2",))

producer_thread.start()
consumer_thread1.start()
consumer_thread2.start()

producer_thread.join()
consumer_thread1.join()
consumer_thread2.join()

print("\n=== All threads completed ===")
print(f"Remaining items: {shared_resource}")
```

Barrier kapasitesi: 3 (1 Producer + 2 Consumer)

shared_resource: []

Bekleyen thread'ler: 0

t = 0.00s - Program Başlıyor

producer_thread.start()

consumer_thread1.start()

consumer_thread2.start()

Barriers Örnek 1: Üretici-Tüketici Problemi

```python
from threading import Thread, Barrier, Lock
import time

# 3 thread için barrier (1 producer + 2 consumer)
barrier = Barrier(3)
shared_resource = []
resource_lock = Lock()  # Liste erişimi için güvenlik

def producer():
    for i in range(5):
        # Veri üret
        with resource_lock:
            shared_resource.append(f"Item {i}")
            print(f"Producer: Produced Item{i}")
        # Tüm thread'lerin senkronize olmasını bekle
        print(f"Producer: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"Producer: Passed barrier (round{i})")
        time.sleep(0.1)  # Küçük gecikme ekle

def consumer(name):
    for i in range(5):
        # Producer'ın üretmesini bekle
        print(f"{name}: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"{name}: Passed barrier (round{i})")
        # Veriyi tüket
        with resource_lock:
            if shared_resource:
                item = shared_resource.pop(0)  # FIFO için pop(0)
                print(f"{name}: Consumed {item}")
            else:
                print(f"{name}: No item available")
        time.sleep(0.1)

print("=== Producer-Consumer with Barrier ===\n")
producer_thread = Thread(target=producer)
consumer_thread1 = Thread(target=consumer, args=("Consumer 1",))
consumer_thread2 = Thread(target=consumer, args=("Consumer 2",))

producer_thread.start()
consumer_thread1.start()
consumer_thread2.start()

producer_thread.join()
consumer_thread1.join()
consumer_thread2.join()

print("\n=== All threads completed ===")
print(f"Remaining items: {shared_resource}")
```

t = 0.01s - Producer Round 0 Başlıyor

```
def producer():
    for i in range(5):  # i = 0 (İlk round)
                with resource_lock:  # Lock aldı 🔒


        # Lock serbest 🔓


        barrier.wait()  # 🚫 BARRIER'DA BLOKE OLDU! (1/3)
```

Barrier Durumu:

Bekleyen: 1/3 (Producer ⏸️)
Durum: 🚫 KAPALI

Barriers Örnek 1: Üretici-Tüketici Problemi

```python
from threading import Thread, Barrier, Lock
import time

# 3 thread için barrier (1 producer + 2 consumer)
barrier = Barrier(3)
shared_resource = []
resource_lock = Lock()  # Liste erişimi için güvenlik

def producer():
    for i in range(5):
        # Veri üret
        with resource_lock:
            shared_resource.append(f"Item {i}")
            print(f"Producer: Produced Item{i}")
        # Tüm thread'lerin senkronize olmasını bekle
        print(f"Producer: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"Producer: Passed barrier (round{i})")
        time.sleep(0.1)  # Küçük gecikme ekle

def consumer(name):
    for i in range(5):
        # Producer'ın üretmesini bekle
        print(f"{name}: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"{name}: Passed barrier (round{i})")
        # Veriyi tüket
        with resource_lock:
            if shared_resource:
                item = shared_resource.pop(0)  # FIFO için pop(0)
                print(f"{name}: Consumed {item}")
            else:
                print(f"{name}: No item available")
        time.sleep(0.1)

print("=== Producer-Consumer with Barrier ===\n")
producer_thread = Thread(target=producer)
consumer_thread1 = Thread(target=consumer, args=("Consumer 1",))
consumer_thread2 = Thread(target=consumer, args=("Consumer 2",))

producer_thread.start()
consumer_thread1.start()
consumer_thread2.start()

producer_thread.join()
consumer_thread1.join()
consumer_thread2.join()

print("\n=== All threads completed ===")
print(f"Remaining items: {shared_resource}")
```

t = 0.02s - Consumer 1 Başlıyor

```
def consumer(name):  # name = "Consumer 1"
    for i in range(5):  # i = 0 (İlk round)

        print(f"{name}: Waiting at barrier (round {i})")
        # Çıktı: "Consumer 1: Waiting at barrier (round 0)"

        barrier.wait()  # 🚫 BARRIER'DA BLOKE OLDU! (2/3)
```

Barrier Durumu:

Bekleyen: 2/3 (Producer ⏸️, Consumer 1 ⏸️)
Durum: 🚫 KAPALI (1 thread daha lazım!)

Barriers Örnek 1: Üretici-Tüketici Problemi

```python
from threading import Thread, Barrier, Lock
import time

# 3 thread için barrier (1 producer + 2 consumer)
barrier = Barrier(3)
shared_resource = []
resource_lock = Lock()  # Liste erişimi için güvenlik

def producer():
    for i in range(5):
        # Veri üret
        with resource_lock:
            shared_resource.append(f"Item {i}")
            print(f"Producer: Produced Item{i}")
        # Tüm thread'lerin senkronize olmasını bekle
        print(f"Producer: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"Producer: Passed barrier (round{i})")
        time.sleep(0.1)  # Küçük gecikme ekle

def consumer(name):
    for i in range(5):
        # Producer'ın üretmesini bekle
        print(f"{name}: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"{name}: Passed barrier (round{i})")
        # Veriyi tüket
        with resource_lock:
            if shared_resource:
                item = shared_resource.pop(0)  # FIFO için pop(0)
                print(f"{name}: Consumed {item}")
            else:
                print(f"{name}: No item available")
        time.sleep(0.1)

print("=== Producer-Consumer with Barrier ===\n")
producer_thread = Thread(target=producer)
consumer_thread1 = Thread(target=consumer, args=("Consumer 1",))
consumer_thread2 = Thread(target=consumer, args=("Consumer 2",))

producer_thread.start()
consumer_thread1.start()
consumer_thread2.start()

producer_thread.join()
consumer_thread1.join()
consumer_thread2.join()

print("\n=== All threads completed ===")
print(f"Remaining items: {shared_resource}")
```

t = 0.03s - Consumer 2 Başlıyor (KRİTİK NOKTA!)

```
def consumer(name):  # name = "Consumer 2"
    for i in range(5):  # i = 0

        print(f"{name}: Waiting at barrier (round {i})")
        # Çıktı: "Consumer 2: Waiting at barrier (round 0)"

        barrier.wait()  # ✅ 3. THREAD GELDİ! (3/3)
```

🎆 **BARRIER AÇILDI!**

Bekleyen: 3/3 (TAM KAPASİTE!)

Durum: ✅ AÇILDI! (Hepsi serbest bırakıldı!)

Barriers Örnek 1: Üretici-Tüketici Problemi

```python
from threading import Thread, Barrier, Lock
import time

# 3 thread için barrier (1 producer + 2 consumer)
barrier = Barrier(3)
shared_resource = []
resource_lock = Lock()  # Liste erişimi için güvenlik

def producer():
    for i in range(5):
        # Veri üret
        with resource_lock:
            shared_resource.append(f"Item {i}")
            print(f"Producer: Produced Item{i}")
        # Tüm thread'lerin senkronize olmasını bekle
        print(f"Producer: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"Producer: Passed barrier (round{i})")
        time.sleep(0.1)  # Küçük gecikme ekle

def consumer(name):
    for i in range(5):
        # Producer'ın üretmesini bekle
        print(f"{name}: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"{name}: Passed barrier (round{i})")
        # Veriyi tüket
        with resource_lock:
            if shared_resource:
                item = shared_resource.pop(0)  # FIFO için pop(0)
                print(f"{name}: Consumed {item}")
            else:
                print(f"{name}: No item available")
        time.sleep(0.1)

print("=== Producer-Consumer with Barrier ===\n")
producer_thread = Thread(target=producer)
consumer_thread1 = Thread(target=consumer, args=("Consumer 1",))
consumer_thread2 = Thread(target=consumer, args=("Consumer 2",))

producer_thread.start()
consumer_thread1.start()
consumer_thread2.start()

producer_thread.join()
consumer_thread1.join()
consumer_thread2.join()

print("\n=== All threads completed ===")
print(f"Remaining items: {shared_resource}")
```

t = 0.04s - TÜM THREAD'LER AYNI ANDA DEVAM EDİYOR

Producer:

```
barrier.wait()  # Geçti! ✅
print(f"Producer: Passed barrier (round {i})")
# Çıktı: "Producer: Passed barrier (round 0)"

time.sleep(0.1)  # Kısa bekle
# Round 0 bitti, for döngüsü devam edecek (round 1'e geçecek)
```

Barriers Örnek 1: Üretici-Tüketici Problemi

```python
from threading import Thread, Barrier, Lock
import time

# 3 thread için barrier (1 producer + 2 consumer)
barrier = Barrier(3)
shared_resource = []
resource_lock = Lock()  # Liste erişimi için güvenlik

def producer():
    for i in range(5):
        # Veri üret
        with resource_lock:
            shared_resource.append(f"Item {i}")
            print(f"Producer: Produced Item{i}")
        # Tüm thread'lerin senkronize olmasını bekle
        print(f"Producer: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"Producer: Passed barrier (round{i})")
        time.sleep(0.1)  # Küçük gecikme ekle

def consumer(name):
    for i in range(5):
        # Producer'ın üretmesini bekle
        print(f"{name}: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"{name}: Passed barrier (round{i})")
        # Veriyi tüket
        with resource_lock:
            if shared_resource:
                item = shared_resource.pop(0)  # FIFO için pop(0)
                print(f"{name}: Consumed {item}")
            else:
                print(f"{name}: No item available")
        time.sleep(0.1)

print("=== Producer-Consumer with Barrier ===\n")
producer_thread = Thread(target=producer)
consumer_thread1 = Thread(target=consumer, args=("Consumer 1",))
consumer_thread2 = Thread(target=consumer, args=("Consumer 2",))

producer_thread.start()
consumer_thread1.start()
consumer_thread2.start()

producer_thread.join()
consumer_thread1.join()
consumer_thread2.join()

print("\n=== All threads completed ===")
print(f"Remaining items: {shared_resource}")
```

Consumer 1:

```
barrier.wait()  # Geçti! ✅
print(f"{name}: Passed barrier (round {i})")

with resource_lock:  # Lock aldı 🔒
    if shared_resource:  # ["Item 0"] - Dolu!
        item = shared_resource.pop(0)  # "Item 0" aldı

        print(f"{name}: Consumed {item}")
        # Çıktı: "Consumer 1: Consumed Item 0"
# Lock serbest 🔓
```

Barriers Örnek 1: Üretici-Tüketici Problemi

```python
from threading import Thread, Barrier, Lock
import time

# 3 thread için barrier (1 producer + 2 consumer)
barrier = Barrier(3)
shared_resource = []
resource_lock = Lock()  # Liste erişimi için güvenlik

def producer():
    for i in range(5):
        # Veri üret
        with resource_lock:
            shared_resource.append(f"Item {i}")
            print(f"Producer: Produced Item{i}")
        # Tüm thread'lerin senkronize olmasını bekle
        print(f"Producer: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"Producer: Passed barrier (round{i})")
        time.sleep(0.1)  # Küçük gecikme ekle

def consumer(name):
    for i in range(5):
        # Producer'ın üretmesini bekle
        print(f"{name}: Waiting at barrier (round{i})")
        barrier.wait()
        print(f"{name}: Passed barrier (round{i})")
        # Veriyi tüket
        with resource_lock:
            if shared_resource:
                item = shared_resource.pop(0)  # FIFO için pop(0)
                print(f"{name}: Consumed {item}")
            else:
                print(f"{name}: No item available")
        time.sleep(0.1)

print("=== Producer-Consumer with Barrier ===\n")
producer_thread = Thread(target=producer)
consumer_thread1 = Thread(target=consumer, args=("Consumer 1",))
consumer_thread2 = Thread(target=consumer, args=("Consumer 2",))

producer_thread.start()
consumer_thread1.start()
consumer_thread2.start()

producer_thread.join()
consumer_thread1.join()
consumer_thread2.join()

print("\n=== All threads completed ===")
print(f"Remaining items: {shared_resource}")
```

Consumer 2:

```
barrier.wait()  # Geçti! ✅
print(f"{name}: Passed barrier (round {i})")

with resource_lock:  # Lock almaya çalışıyor...
```

Consumer 1 hala lock'u tutuyor olabilir, o zaman **bekler** ⏸️. Consumer 1 bıraktıktan sonra:

```
with resource_lock:  # Lock aldı 🔒
    if shared_resource:  # [] - BOŞ!
        item = shared_resource.pop(0)
        print(f"{name}: Consumed {item}")
    else:
        print(f"{name}: No item available")
        # Çıktı: "Consumer 2: No item available"
# Lock serbest 🔓
```