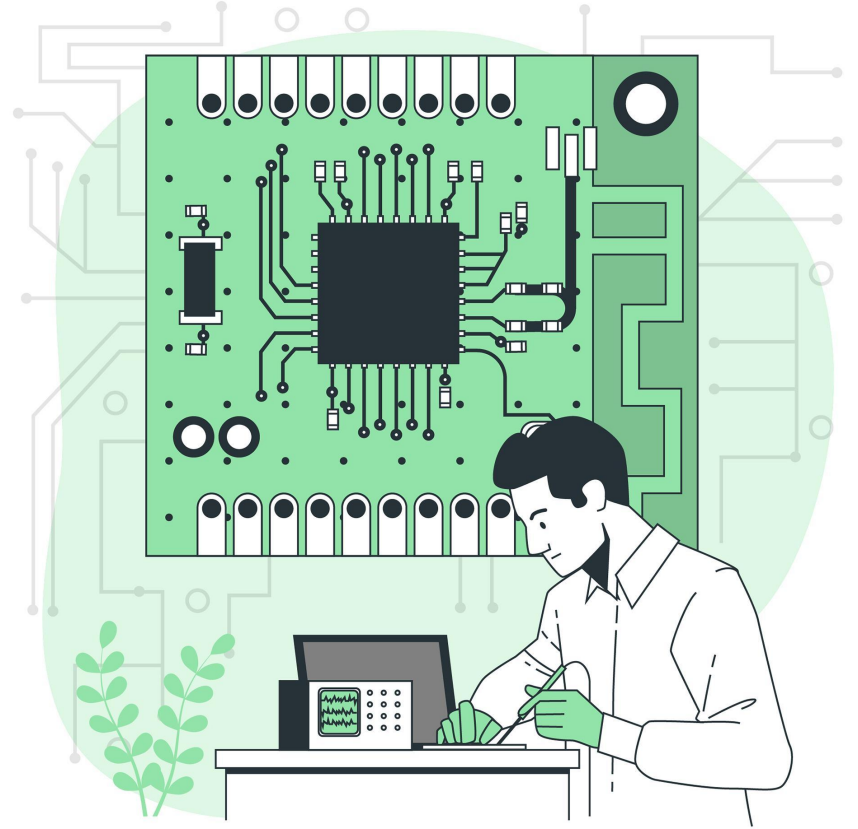


03 İş Parçacığı Threads

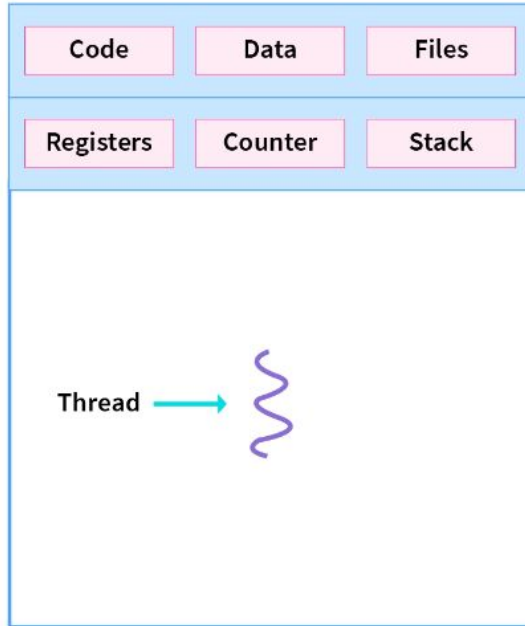


İçindekiler

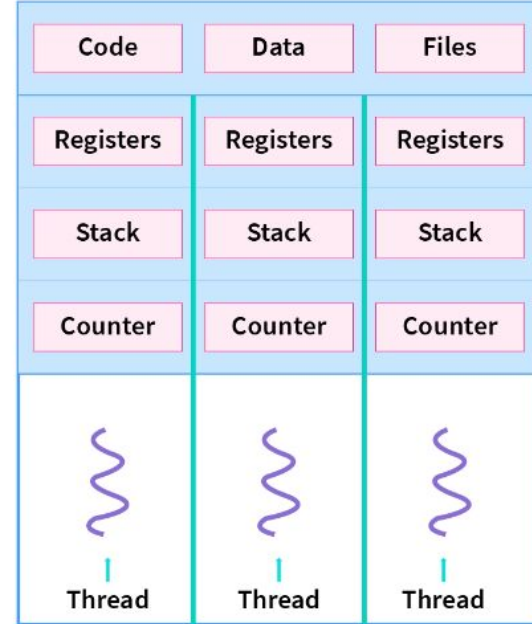
- Thread (İş Parçacığı)
 - Nedir
 - Faydaları
 - Örneği
 - Yaşam Döngüsü
- Klasik İş Parçacığı Modeli
- Süreç ile İş Parçacığı Farkları
- Çekirdek/Kullanıcı İş Parçacıkları
- Race Condition (Yarış Durumu)
- Critick Sections (Kritik Bölgeler)
- Semafor
- Mutex
- Barriers



İş Parçacığı Nedir?

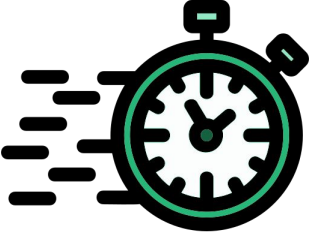


Single-threaded process

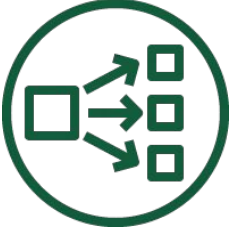


Multithreaded process

İş Parçacığının Faydaları



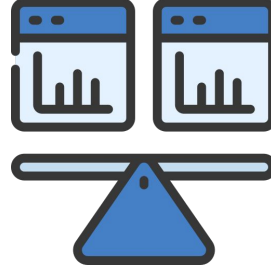
Hızlı yanıt verme
kapasitesi



Parallel Çalışma



Daha az
hafıza
gereksinimi



İşlemciler arasında
yük dengelemesi



Oluşturma ve yok
etme kolaylığı

İş Parçacığı Örneği



HayaliRoman.docx



Page: 600



Bolum1.docx



BolumN.docx

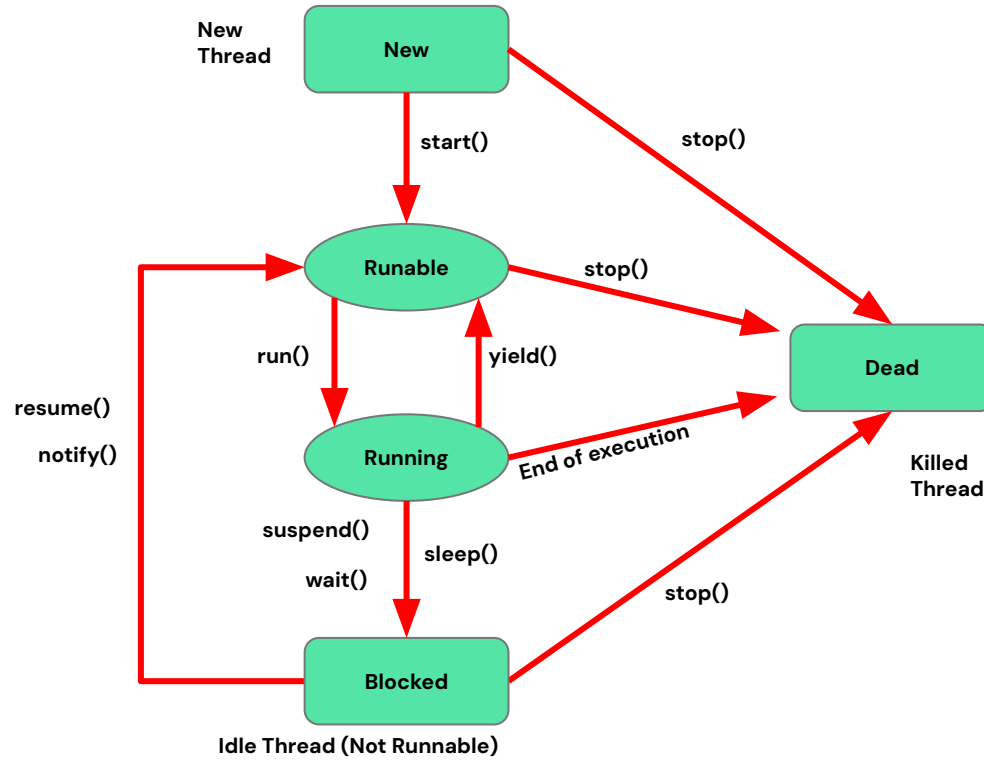


.....

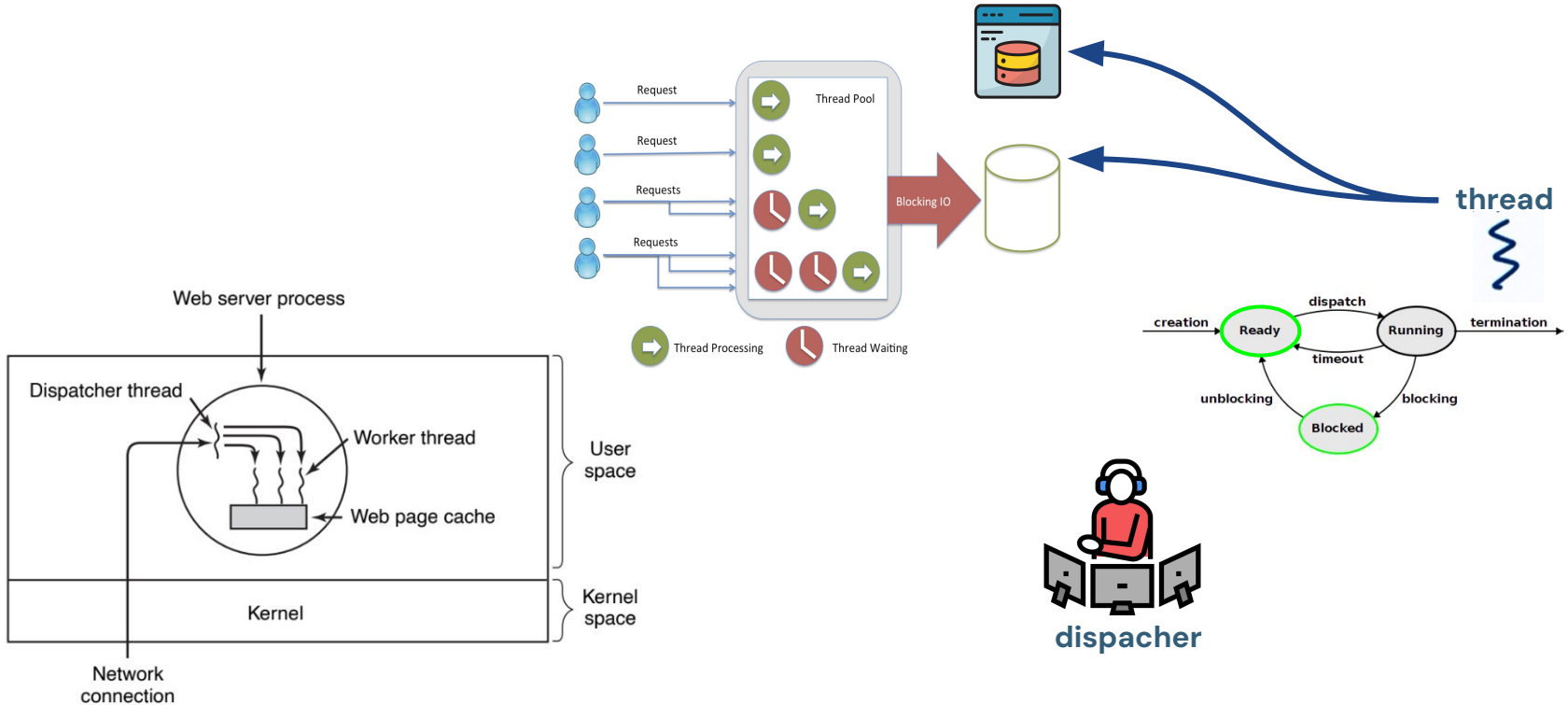
İş Parçacığı Örneği



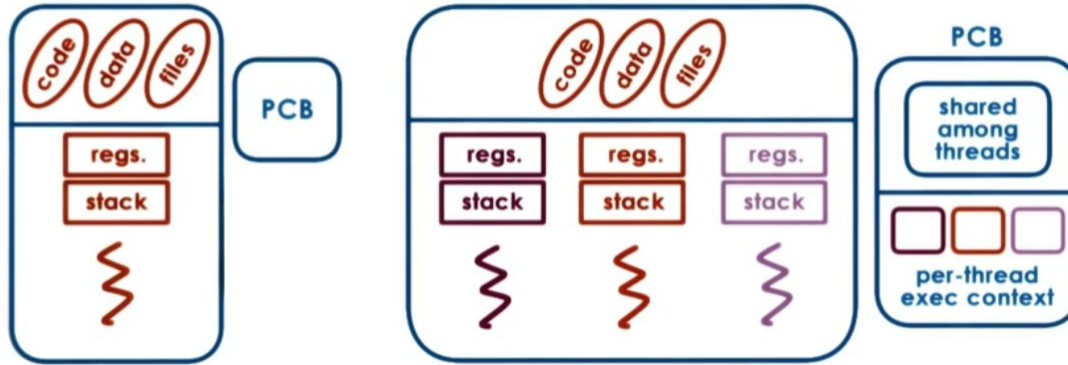
İş Parçacığı Yaşam Döngüsü



İş Parçacığı Yaşam Döngüsü Örneği



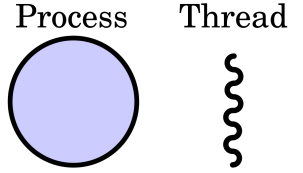
Klasik İş Parçacığı Modeli



Süreç ile İş Parçacığı Farkları

Süreç

İş Parçacığı

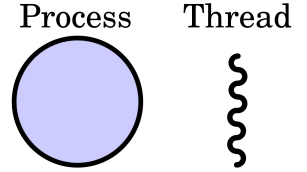


Süreç ile İş Parçacığı Farkları

	Süreç	İş Parçacığı
İletişim	Uzun	Kısa
		

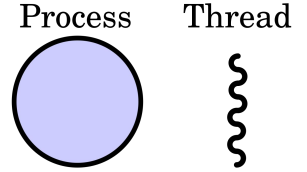
Süreç ile İş Parçacığı Farkları

	Süreç	İş Parçacığı
İletişim	Uzun	Kısa
Kaynak Kullanımı	Fazla	Az



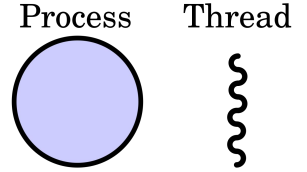
Süreç ile İş Parçacığı Farkları

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



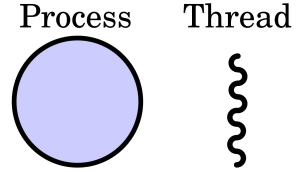
Süreç ile İş Parçacığı Farkları

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



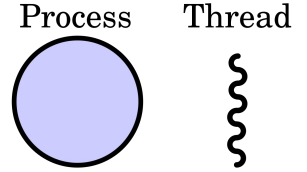
Süreç ile İş Parçacığı Farkları

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



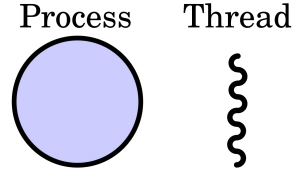
Süreç ile İş Parçacığı Farkları

	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

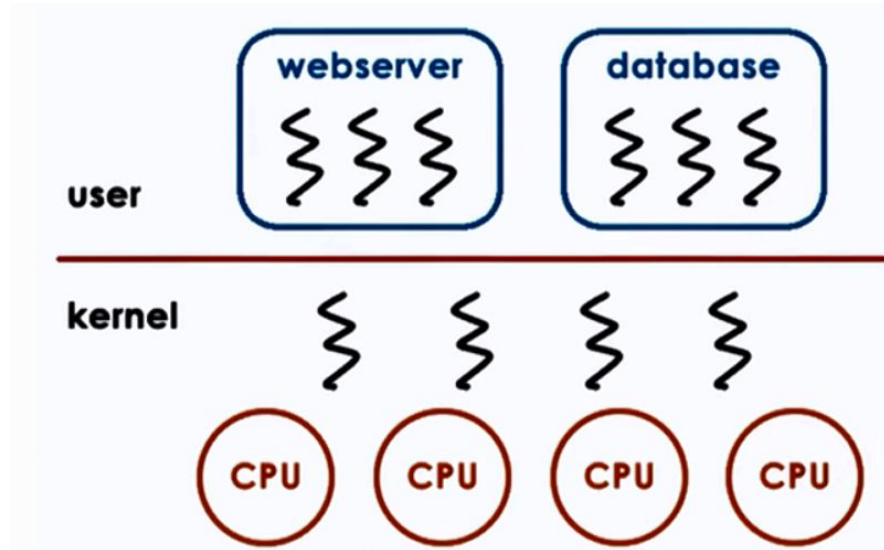


Süreç ile İş Parçacığı Farkları

	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

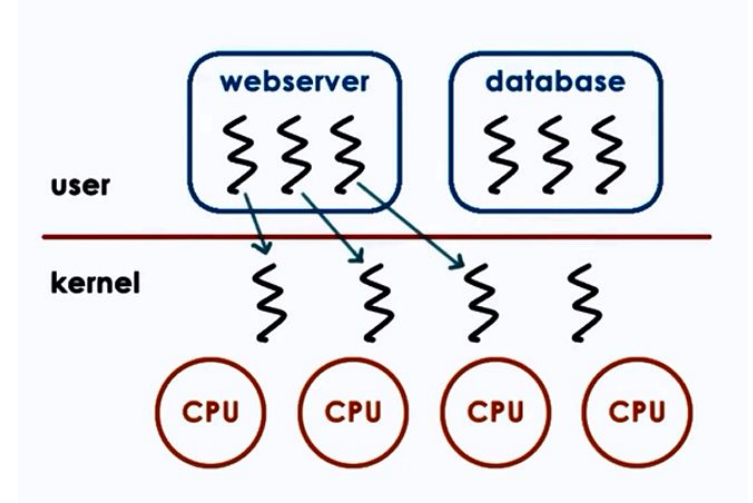
One to One model:

Avantajları:

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

Dezavantajları:

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



Çekirdek/Kullanıcı İş Parçacıkları

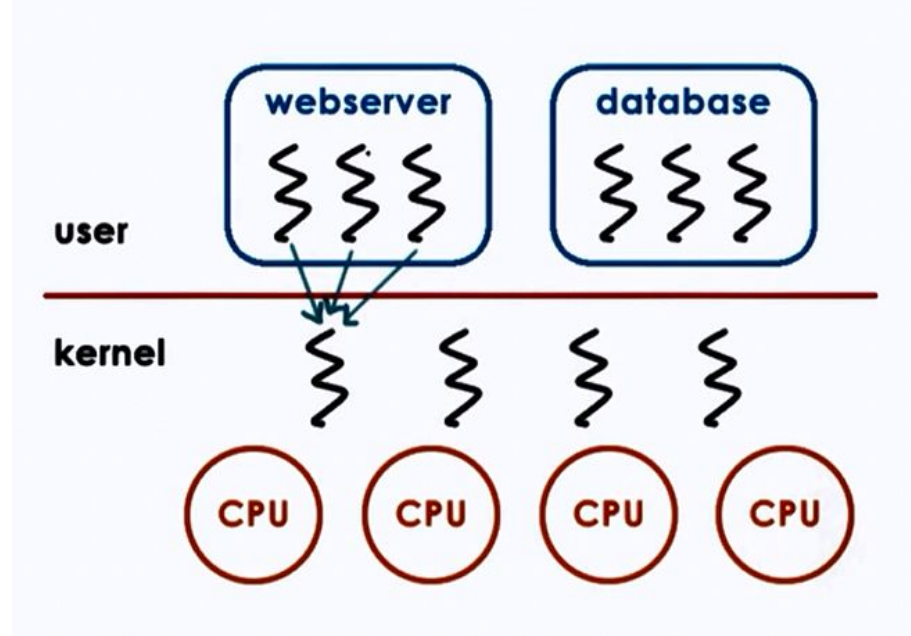
Many to One model:

Avantajları:

- Taşınabilir
- İş parçacıklarında OS sınırlaması yoktur

Dezavantajları:

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



Çekirdek/Kullanıcı İş Parçacıkları

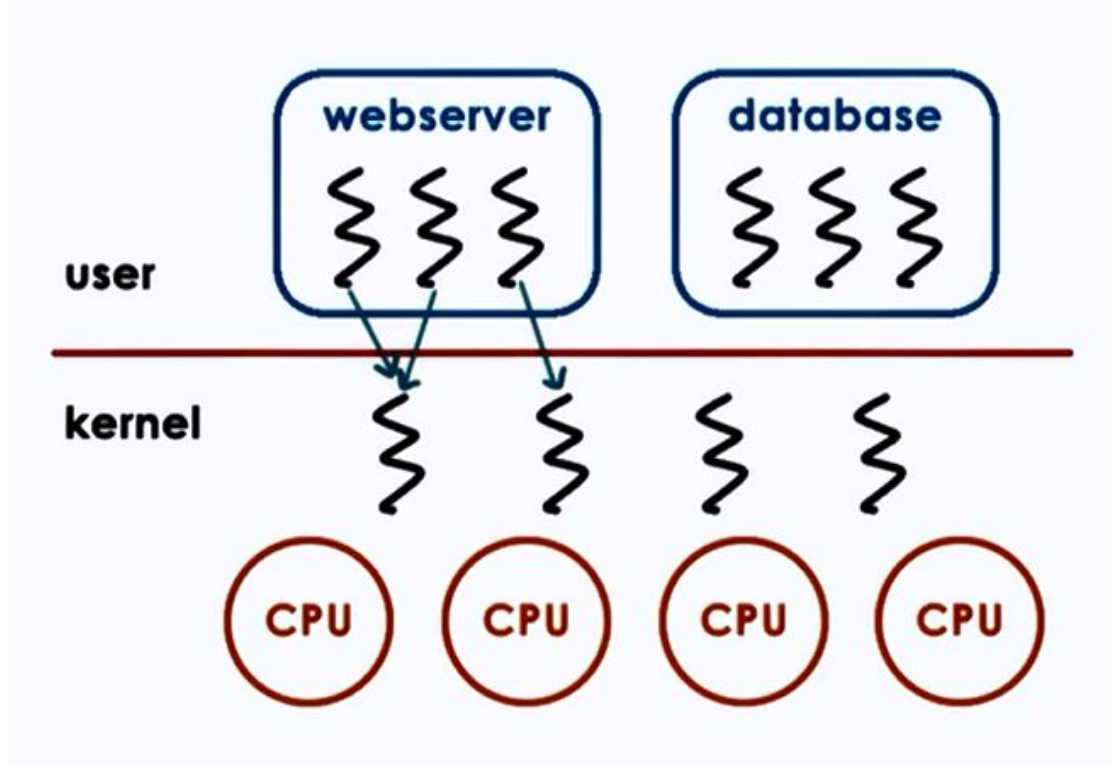
Many to Many model:

Avantajları:

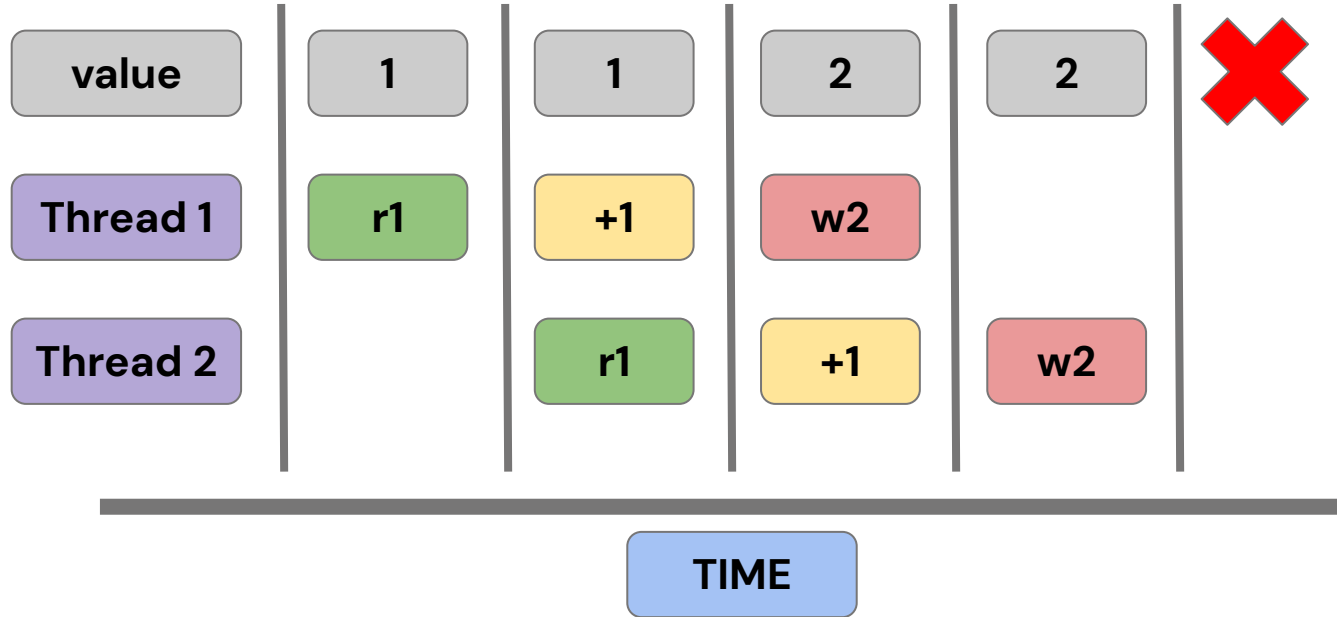
- En iyi modeldir
- Bağlı veya ilişkisiz iş parçacıkları olabilir

Dezavantajları:

- Koordinasyon gerektirir



Race Condition (Yarış Durumu)



Race Condition Neden Olur?

1. Paylaşılan Kaynaklara Eş Zamanlı Erişim

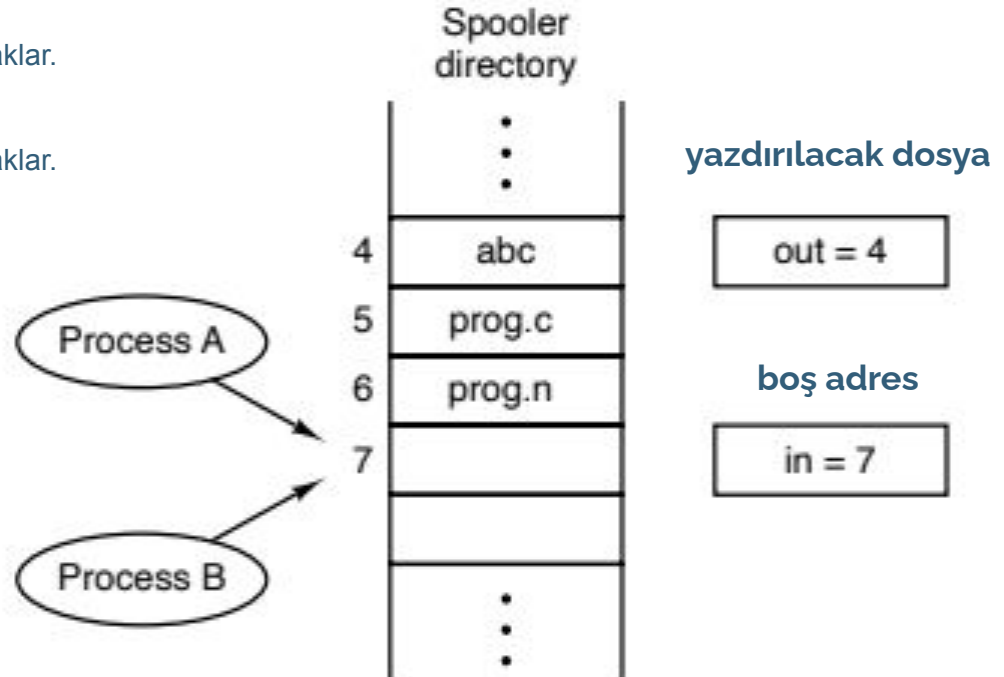
Birden fazla süreç veya iş parçacığı, aynı veriye veya kaynağa eş zamanlı olarak erişmeye çalışır.

2. Yetersiz Senkronizasyon

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

Race Condition: Yazdırma Kuyruğu

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.

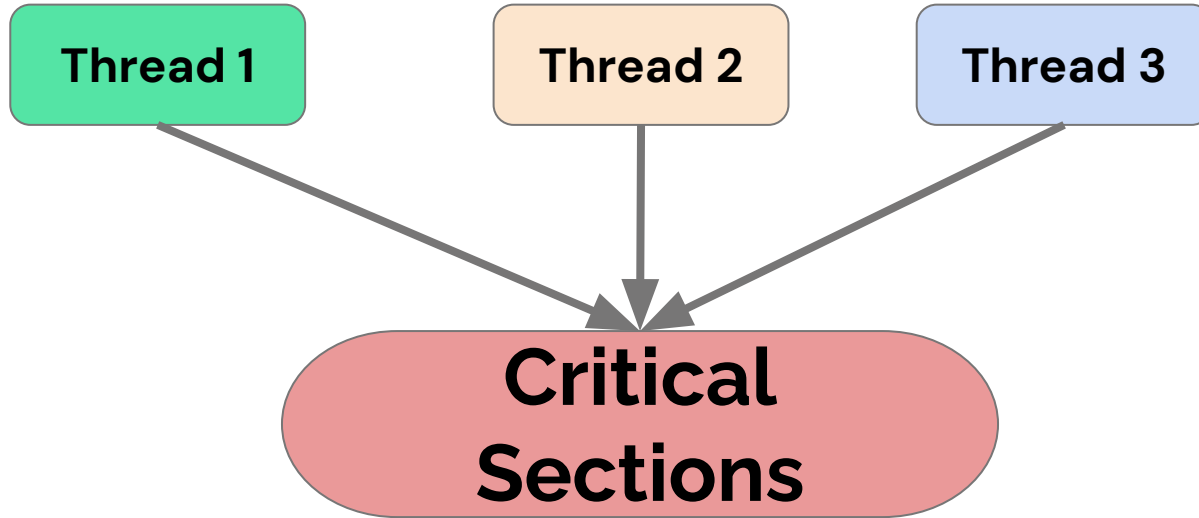
Atomik İşlemler

Birtakım işlemler, bölünemez ve kesintisiz olarak atomik olarak işaretlenebilir.

Transactionlar

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

Critical Sections (Kritik Bölgeleler)



Critical Sections Neden Önemlidir?

1. Veri Tutarsızlığının Önlenmesi

Birden fazla thread aynı veri üzerinde eş zamanlı işlem yaptığında, veri tutarsızlıkları oluşabilir.

2. Race Conditionların Önlenmesi

Kritik bölgelerin uygun şekilde yönetilmesi, race conditionların önlenmesine yardımcı olur.

Critical Sections Örnek 1: Banka Hesabı Güncellemesi

Birden fazla kullanıcının aynı banka hesabına para yatırması veya çekmesi durumunda, hesap bakiyesi bir kritik bölge oluşturur.

Kritik Bölge

Hesap bakiyesini güncelleyen kod bölümü.

Risk

Eş zamanlı güncellemeler, yanlış hesap bakiyesi hesaplamalarına yol açabilir.

Yönetim

Mutex veya semafor kullanarak, bir seferde yalnızca bir iş parçasığı veya sürecin bakiyeyi güncellemesini sağlamak.

Critical Sections Örnek 2: Log Dosyasına Yazma

Birden fazla süreç veya iş parçacığı, aynı log dosyasına yazmak istediğinde, dosyaya erişim bir kritik bölgedir.

Kritik Bölge

Dosyaya yazma işlemi yapan kod bölümü.

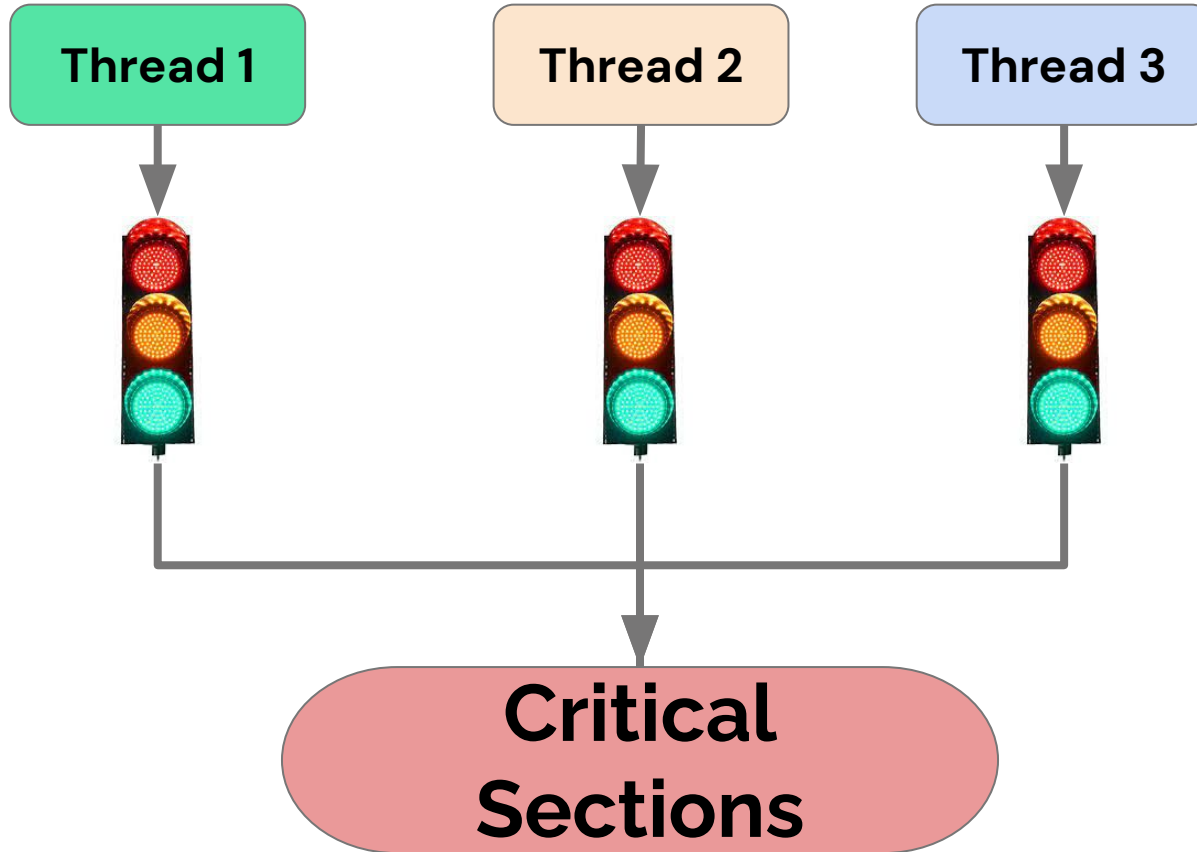
Risk

Eş zamanlı yazma işlemleri, log kayıtlarının karışmasına veya kaybolmasına neden olabilir.

Yönetim

Kilit mekanizmaları kullanarak dosyaya sıralı erişim sağlamak.

Semafor



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 parçasığı/süreç tarafından kullanılabilirliğini 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 iş 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

```
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} finished 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()
```

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

python

```
from threading import Semaphore, Thread
import time

pool_semaphore = Semaphore(2)

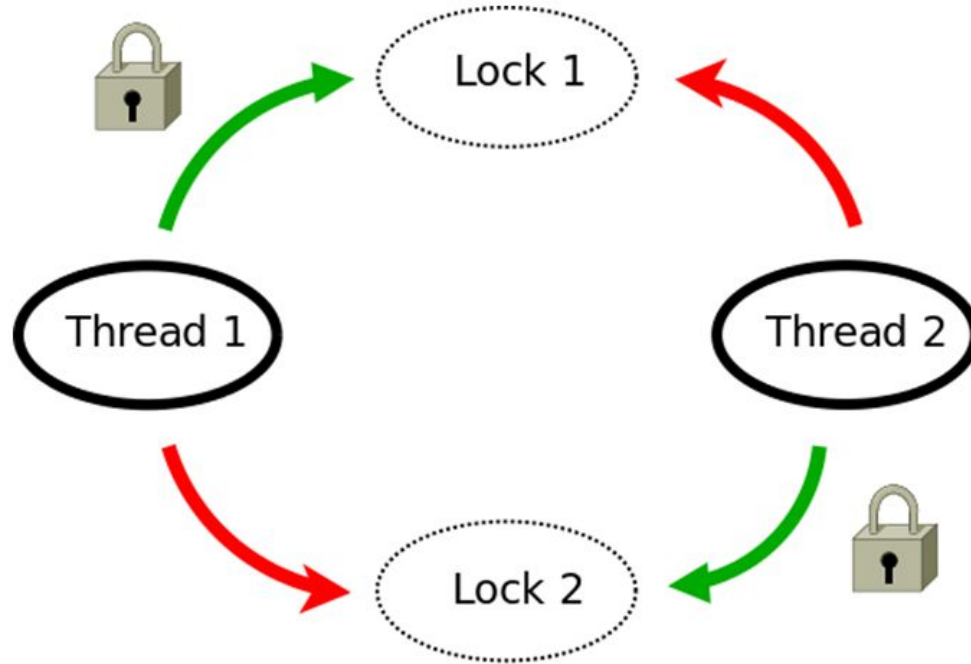
def swimmer(name):
    pool_semaphore.acquire()
    print(f"{name} is swimming in the pool.")
    time.sleep(2)
    print(f"{name} finished swimming.")
    pool_semaphore.release()

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()
```

Mutex



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

Birden fazla iş parçacığı/süreç aynı dosyayı okuma/yazma işlemi yapmak istediğinde, mutex kullanabilirsiniz.

python

```
import threading

file_mutex = threading.Lock()
file_contents = []

def write_to_file(data):
    with file_mutex:
        file_contents.append(data)

def read_from_file():
    with file_mutex:
        return file_contents

# İki thread, aynı anda dosyaya yazma işlemi yapabilir.
thread1 = threading.Thread(target=write_to_file, args=("Data from Thread 1",))
thread2 = threading.Thread(target=write_to_file, args=("Data from Thread 2",))
```

```
thread1.start()
thread2.start()

thread1.join()
thread2.join()

# Dosyadan okuma işlemi de mutex ile korunur.
def read_thread():
    data = read_from_file()
    print(data)

read_thread1 = threading.Thread(target=read_thread)
read_thread2 = threading.Thread(target=read_thread)

read_thread1.start()
read_thread2.start()

read_thread1.join()
read_thread2.join()
```

Mutex Örnek 1: Kaynak Paylaşımı

Birden fazla iş parçacığı/süreç, bir kaynağı paylaşmak istediğinde, mutex kullanarak kaynağa sıralı erişim sağlayabilirsiniz.

```
python

import threading

shared_resource = []
mutex = threading.Lock()

def append_to_shared(data):
    with mutex:
        shared_resource.append(data)

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

# İki thread, aynı anda kaynağı paylaşabilir.
thread1 = threading.Thread(target=append_to_shared, args=("Data 1",))
thread2 = threading.Thread(target=append_to_shared, args=("Data 2",))
```

```
thread1.start()
thread2.start()

thread1.join()
thread2.join()

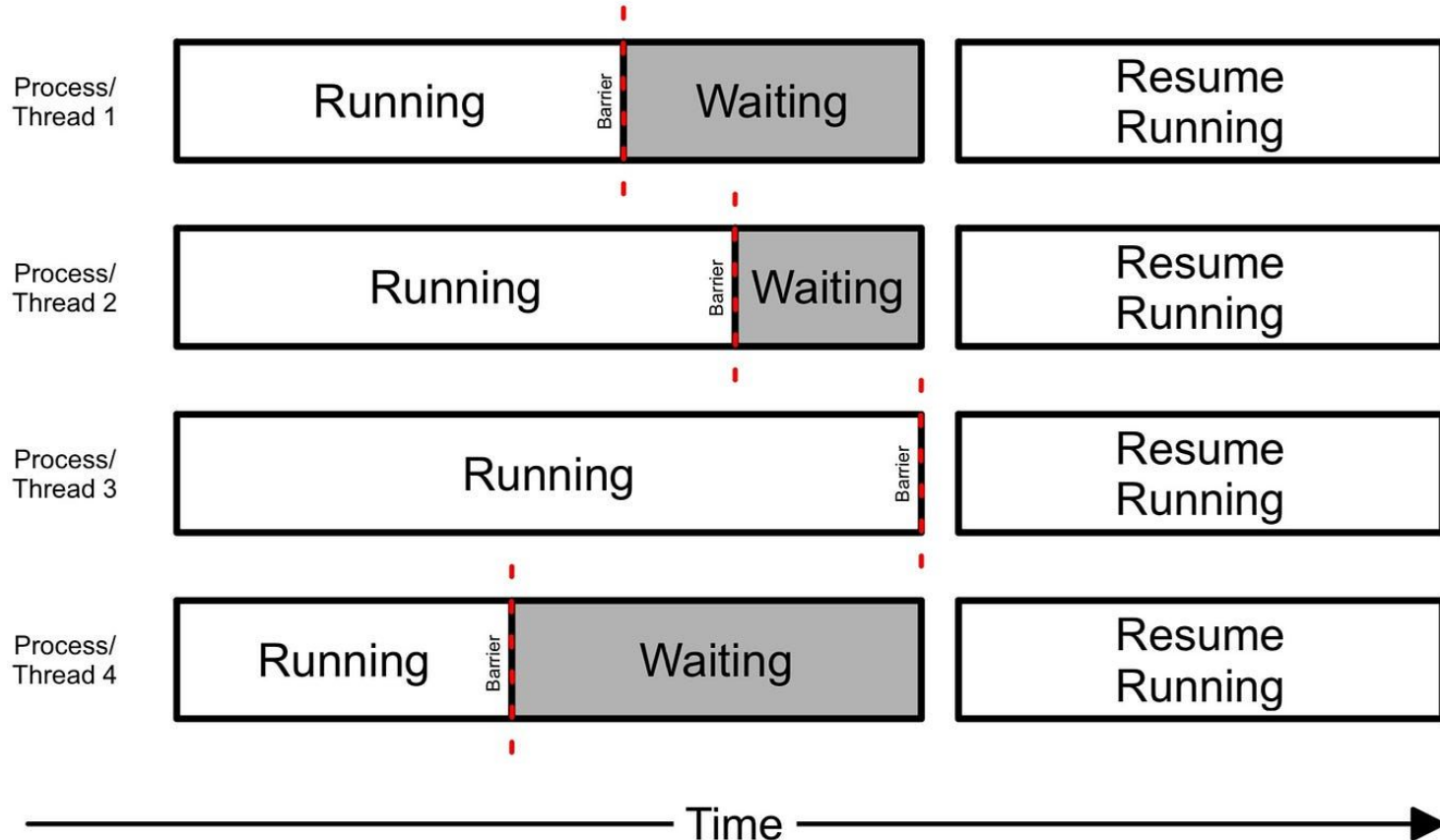
# Kaynağı kullanmak ve çıkarmak da mutex ile korunur.
def use_shared():
    data = remove_from_shared()
    if data:
        print(f"Used: {data}")
    else:
        print("No data available.")

use_thread1 = threading.Thread(target=use_shared)
use_thread2 = threading.Thread(target=use_shared)

use_thread1.start()
use_thread2.start()

use_thread1.join()
use_thread2.join()
```

Barriers (Engelleyciler)



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

Bir yarışta, bir grup koşucu aynı anda başlamak istiyor. Barrier, bu senaryoyu modellemek için kullanılabilir.

```
python

import threading

barrier = threading.Barrier(5) # 5 katılımcı bekleniyor.

def runner(name):
    print(f"{name} is ready.")
    barrier.wait() # Tüm koşucuların hazır olduğunu bekler.
    print(f"{name} started running.")

# 5 koşucu aynı anda başlar.
runner1 = threading.Thread(target=runner, args=("Runner 1",))
runner2 = threading.Thread(target=runner, args=("Runner 2",))
runner3 = threading.Thread(target=runner, args=("Runner 3",))
runner4 = threading.Thread(target=runner, args=("Runner 4",))
runner5 = threading.Thread(target=runner, args=("Runner 5",))

runner1.start()
runner2.start()
runner3.start()
runner4.start()
runner5.start()

runner1.join()
runner2.join()
runner3.join()
runner4.join()
runner5.join()
```

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

import threading

barrier = threading.Barrier(3) # Üretici ve iki tüketici bekleniyor.
shared_resource = []

def producer():
    for i in range(5):
        shared_resource.append(f"Item {i}")
        print(f"Produced Item {i}")
        barrier.wait()

def consumer(name):
    barrier.wait() # Üreticiyi bekler.
    while shared_resource:
        item = shared_resource.pop()
        print(f"{name} consumed {item}")

producer_thread = threading.Thread(target=producer)
consumer_thread1 = threading.Thread(target=consumer, args=("Consumer 1",))
consumer_thread2 = threading.Thread(target=consumer, args=("Consumer 2",))

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

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

POSIX Thread

İş Parçacığı Çağrısı (Thread Call)

Açıklaması

pthread create

Yeni bir iş parçacığı oluşturma

pthread exit

Çağrılan iş parçacığının sonlandırılması

pthread join

Belli bir iş parçacığının çıkmasını bekleme

Pthreads iş parçacığı çağrılarından bazıları

POSIX Thread

İş Parçacığı Çağrısı (Thread Call)

Açıklaması

pthread yield

Başka bir iş parçacığının çalışması için
CPU yu serbest bırakma

pthread attr init

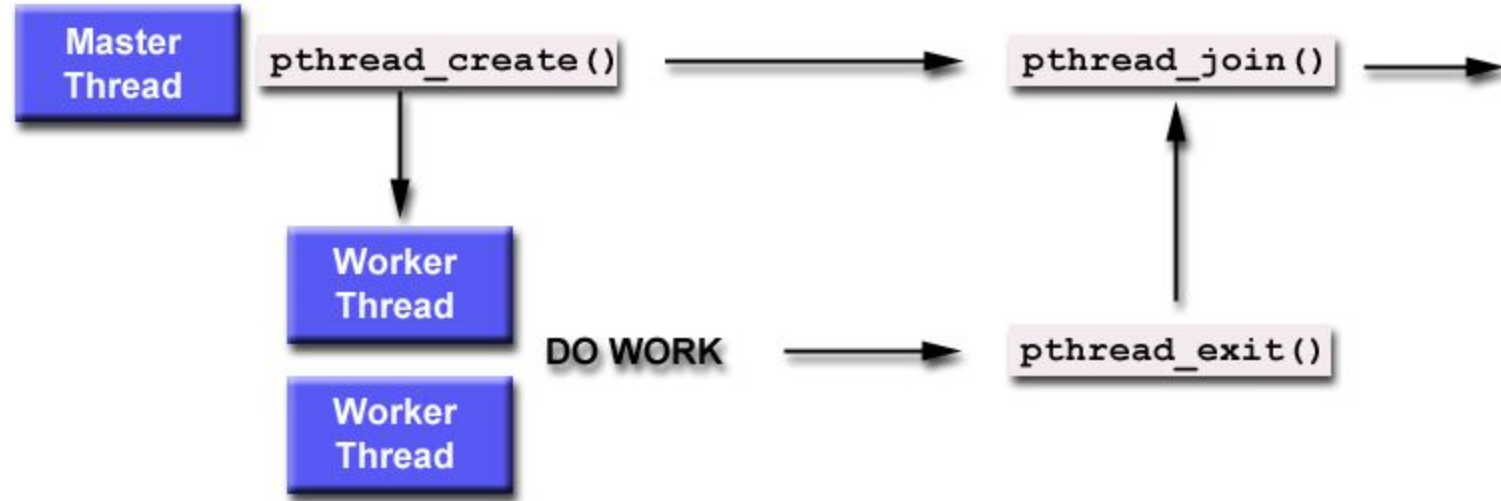
İş parçacığını öznitelikleri ile oluşturup
başlatma

pthread attr destroy

İş parçacığının öznitelik yapısını kaldırma

Pthreads iş parçacığı çağrılarından bazıları

POSIX Thread



User Space Thread Kullanımı - Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define MAX_THREADS 50

pthread_t thread_id[MAX_THREADS];

void * PrintHello(void * data)
{
    printf("Hello from thread %u - I was created in iteration %d !\n", (int)pthread_self(), (int)data);
    pthread_exit(NULL);
}
```

User Space Thread Kullanımı - Pthread

```
int main(int argc, char * argv[])
{
    int rc, i, n;

    if(argc < 2)
    {
        printf("Please add the number of threads to the command line\n");
        exit(1);
    }
    n = atoi(argv[1]);
    if(n > MAX_THREADS) n = MAX_THREADS;

    for(i = 0; i < n; i++)
    {
        rc = pthread_create(&thread_id[i], NULL, PrintHello, (void*)i);
        if(rc)
        {
            printf("\n ERROR: return code from pthread_create is %d \n", rc);
            exit(1);
        }
        printf("\n I am thread %u. Created new thread (%u) in iteration %d ... \n",
            (int)pthread_self(), (int)thread_id[i], i);
        if(i % 5 == 0) sleep(1);
    }

    pthread_exit(NULL);
}
```

User Space Thread Kullanımı - Pthread

```
I am thread 1. Created new thread (4) in iteration 0...
Hello from thread 4 - I was created in iteration 0
I am thread 1. Created new thread (6) in iteration 1...
I am thread 1. Created new thread (7) in iteration 2...
I am thread 1. Created new thread (8) in iteration 3...
I am thread 1. Created new thread (9) in iteration 4...
I am thread 1. Created new thread (10) in iteration 5...
Hello from thread 6 - I was created in iteration 1
Hello from thread 7 - I was created in iteration 2
Hello from thread 8 - I was created in iteration 3
Hello from thread 9 - I was created in iteration 4
Hello from thread 10 - I was created in iteration 5
I am thread 1. Created new thread (11) in iteration 6...
I am thread 1. Created new thread (12) in iteration 7...
Hello from thread 11 - I was created in iteration 6
Hello from thread 12 - I was created in iteration 7
```