

**İSTANBUL SAĞLIK VE TEKNOLOJİ ÜNİVERSİTESİ**



**BİL 302 – İŞLETİM SİSTEMLERİ**

**BAHAR 2025**

**ÖDEV 3 RAPORU**

**AYŞE CANSU YILDIRIM**

**BİLGİSAYAR MÜHENDİSLİĞİ 3. SINIF**

**220601039**

**Son teslim tarihi: 13 Haziran 2025 Cuma, 23:59**

# İÇİNDEKİLER

1. Giriş: Yazıcı/Okuyucu Problemi ve Önemi
2. Proje Tasarımı ve Mimarisi
  - 2.1. Sınıf Yapısı
  - 2.2. Paylaşılan Kaynak ve Durum Değişkenleri
3. Algoritma Analizi ve Senkronizasyon Mekanizması
  - 3.1. Algoritmanın Temel Mantığı
  - 3.2. Senkronizasyon Araçları: ReentrantLock ve Condition Değişkenleri
  - 3.3. İş Parçacıklarının Senkronizasyonu (Metot Analizi)
    - 3.3.1. startRead() ve endRead()
    - 3.3.2. startWrite() ve endWrite()
  - 3.4. Kritik Teknik Detay: Mesa Monitörleri ve while Kullanımı
4. Derste Anlatılan Konseptler Dışındaki Teknik Seçimler ve Gerekçeleri
  - 4.1. ReentrantLock vs. synchronized
  - 4.2. ReentrantLock(true) (Fairness Policy) Kullanımı
5. Java Kodlarının Detaylı Açıklaması
6. Simülasyon Çıktısı ve Yorumlanması
7. Sonuç
8. Kaynakça
9. ReadMe Dosyam

- **Ödev İsterleri:**

Yazıcı/Okuyucu Problemi eş zamanlı olarak bir kaynağa okuma veya yazma yapan threadlerin senkronizasyonuna dayanır. Bu problemde

- Sistemde sadece okuyucular varsa senkronizasyon gerektirmeden eş zamanlı kaynak okunabilir.
- Sistemde eş zamanlı olarak kaynağa erişen tek bir yazıcı varsa diğer tüm okuyucular ve yazıcılar bekletilmeli ve yazıcının işleminin tamamlanması beklenmelidir.
- Yukarıda belirtilen durum ile kaynağa sağlıklı erişimi sağlamalıdır ve veri kaybını önlemelidir.
- Okuyucu-Yazıcı problemi birçok bilişim probleminin çözümünde kullanılmaktadır. En önemli kullanım yeri veri tabanına okuma/yazma işlemi, web sunucularında sunulan veri üzerinde eş zamanlı yapılabilecek okuma/yazma işlemleri vs. Dir.

## 1. Giriş: Yazıcı/Okuyucu Problemi ve Önemi

İşletim sistemleri ve eş zamanlı programlama alanının klasik problemlerinden biri olan Yazıcı/Okuyucu (Readers-Writers) Problemi, birden çok iş parçacığının (thread) paylaşılan bir veri kaynağına erişimini yönetme senaryosunu modellemektedir.

Problemin temel amacı, veri tutarlılığını korurken sistem performansını maksimize etmektir. Bu, iki temel erişim kuralı ile sağlanır:

1. **Özel Yazma Hakkı:** Bir "Yazıcı" iş parçacığı kaynağa yazma işlemi yaparken, başka hiçbir iş parçacığı (ne okuyucu ne de yazıcı) kaynağa erişememelidir. Bu kural, veri bozulmasını (data corruption) önler.
2. **Paylaşımlı Okuma Hakkı:** Sistemde aktif bir yazıcı yoksa, birden fazla "Okuyucu" iş parçacığı kaynağa eş zamanlı olarak erişip okuma yapabilir. Bu kural, gereksiz beklemleri önleyerek sistemin işlem hacmini (throughput) artırır.

Bu problem, veritabanı yönetim sistemleri, web sunucuları, dosya sistemleri ve önbellekleme mekanizmaları gibi okuma işlemlerinin yazma işlemlerine göre çok daha sık yapıldığı birçok gerçek dünya senaryosunun temelini oluşturur. Bu proje kapsamında, verilen algoritma kullanılarak Yazıcı/Okuyucu probleminin bir simülasyonu, Java programlama dili ve modern eş zamanlılık araçları ile geliştirilmiştir.

## 2. Proje Tasarımı ve Mimarisi

Simülasyon, nesne yönelimli programlama ilkelerine uygun olarak modüler ve anlaşılır bir yapıda tasarlanmıştır. Proje, sorumlulukları net bir şekilde ayrılmış dört ana sınıftan oluşmaktadır.

### 2.1. Sınıf Yapısı

- **SharedResource.java:** Simülasyonun merkezi kontrol mekanizmasıdır. Tüm senkronizasyon mantığını, paylaşılan veriyi ve sistemin durumunu yöneten değişkenleri içerir. Diğer tüm sınıflar bu nesne üzerinden etkileşim kurar.
- **Reader.java:** Bir okuyucu iş parçacığını temsil eden Runnable arayüzünü implemente eden sınıftır. Görevi, periyodik olarak SharedResource üzerinden okuma izni istemek, veriyi okumak ve işlem bitiminde sistemden ayrılmaktır.
- **Writer.java:** Bir yazıcı iş parçacığını temsil eden Runnable sınıfıdır. Görevi, periyodik olarak SharedResource üzerinden yazma izni istemek, paylaşılan veriyi güncellemek ve işlem bitiminde sistemden ayrılmaktır.
- **Simulation.java:** Simülasyonun ana başlangıç noktasıdır (main metodu). Belirlenen sayıda okuyucu ve yazıcı iş parçacığını oluşturur, onlara paylaşılan SharedResource nesnesini referans olarak verir ve tüm iş parçacıklarını başlatarak simülasyonu tetikler.

## 2.2. Paylaşılan Kaynak ve Durum Değişkenleri

SharedResource sınıfı, tüm iş parçacıkları tarafından paylaşılan ve korunması gereken şu elemanları içerir:

- **private String sharedData:** Tüm thread'lerin erişmeye çalıştığı paylaşılan kaynak. Başlangıç değeri "Veri1" olarak atanmıştır.
- **private final ReentrantLock lock:** Karşılıklı dışlamayı (mutual exclusion) sağlayan ve senkronizasyonun temelini oluşturan kilittir.
- **private final Condition okToRead:** Okuyucuların, okuma koşulu sağlanana kadar bekleyeceği koşul değişkenidir.
- **private final Condition okToWrite:** Yazıcıların, yazma koşulu sağlanana kadar bekleyeceği koşul değişkenidir.
- **Durum Sayaçları:** Algoritmanın gerektirdiği, sistemin anlık durumunu takip eden sayaçlardır:
  - **activeReaders (AR):** O anda aktif olarak okuma yapan okuyucu sayısı.
  - **activeWriters (AW):** O anda aktif olarak yazma yapan yazıcı sayısı (her zaman 0 veya 1 olabilir).
  - **waitingReaders (WR):** Okuma izni için sırada bekleyen okuyucu sayısı.
  - **waitingWriters (WW):** Yazma izni için sırada bekleyen yazıcı sayısı.

### 3. Algoritma Analizi ve Senkronizasyon Mekanizması

Bu projede, PDF'te tarif edilen ve yazıcılara öncelik veren (writer-preference) bir algoritma çözümü implemente edilmiştir.

Senkronizasyon, **Java'nın Condition değişkenleri kullanılarak** sağlanmıştır. Ödev metninde "semaphore" ifadesi geçse de, Java'daki Condition değişkenleri, semaforların gerçekleştirdiği sinyalizasyon ve bekleme mekanizmasını daha esnek ve yapısal bir şekilde sunduğu için modern Java uygulamalarında tercih edilmektedir. Condition değişkenleri, bir kilit (Lock) ile birlikte çalışarak iş parçacıklarının belirli koşullar sağlanana kadar bekletilmesini ve uyandırılmasını sağlar. Bu yapı, semaforların temel prensibini modern bir monitör yapısı içinde gerçekleştirir.

#### 3.1. Algoritmanın Temel Mantığı

Algoritma, bir iş parçacığı kaynağa erişmeden önce "giriş protokolünü" (startRead/startWrite), işi bittiğinde ise "çıkış protokolünü" (endRead/endWrite) işletir. Bu protokoller, yukarıda bahsedilen durum sayaçlarını güncelleyerek ve Condition değişkenleri üzerinden sinyalleşerek erişimi yönetir.

#### 3.2. Senkronizasyon Araçları: ReentrantLock ve Condition Değişkenleri

- **ReentrantLock:** Bu kilit, SharedResource içindeki durum değişkenlerinin (AR, AW vb.) aynı anda sadece tek bir iş parçacığı tarafından değiştirilmesini garanti eder. Bir iş parçacığı lock.lock() ile kilidi aldığı anda, diğer tüm iş

parçacıkları kilit serbest bırakılana kadar bekler. Bu, "race condition" (yarış durumu) hatalarını önler.

- **Condition:** Bir iş parçacığı, erişim koşulu sağlanmadığında `await()` metodunu çağırır. Bu metot, atomik olarak (bölünemez bir işlemle) elindeki kilidi serbest bırakır ve kendisini bekleme kuyruğuna alır. Başka bir iş parçacığı, durumu değiştirdikten sonra `signal()` (tek bir thread'i uyandır) veya `signalAll()` (tüm bekleyenleri uyandır) metotlarını çağırarak bekleyen iş parçacıklarını uyandırır.

### 3.3. İş Parçacıklarının Senkronizasyonu (Metot Analizi)

- **startRead():**
  1. Okuyucu, `lock.lock()` ile kilidi alır.
  2. `while ((activeWriters + waitingWriters) > 0)` döngüsü ile kritik koşulu kontrol eder: "Sistemde aktif veya sırada bekleyen bir yazıcı var mı?"
  3. Eğer koşul doğruysa (bir yazıcı varsa), okuyucu `waitingReaders` sayacını artırır ve `okToRead.await()` ile uykuya dalar. Uyandığında, `while` döngüsü sayesinde koşulu tekrar kontrol eder.
  4. Koşul yanlışsa (yazıcı yoksa), döngüden çıkar, `activeReaders` sayacını bir artırır ve `lock.unlock()` ile kilidi bırakarak okuma işlemine başlar.
- **endRead():**
  1. Okuyucu, `lock.lock()` ile tekrar kilidi alır.
  2. `activeReaders` sayacını bir azaltır.
  3. `if (activeReaders == 0 && waitingWriters > 0)` koşulunu kontrol eder: "Ben sistemden ayrılan son okuyucu muyum VE sırada bekleyen bir yazıcı var mı?"



4. Eğer koşul doğruysa, `okToWrite.signal()` ile sırada bekleyen yazıcılardan birini uyandırır.
5. `lock.unlock()` ile kilidi serbest bırakır.

- **startWrite():**

1. Yazıcı, `lock.lock()` ile kilidi alır.
2. `while ((activeWriters + activeReaders) > 0)` döngüsü ile kritik koşulu kontrol eder: "Sistemde başka bir aktif yazıcı veya herhangi bir aktif okuyucu var mı?"
3. Koşul doğruysa, yazıcı `waitingWriters` sayacını artırır ve `okToWrite.await()` ile uykuya dalar.
4. Koşul yanlışsa, `activeWriters` sayacını bir artırır ve kilidi bırakarak yazma işlemine başlar.

- **endWrite():**

1. Yazıcı, `lock.lock()` ile kilidi alır.
2. `activeWriters` sayacını bir azaltır.

### 3. Önceliklendirme Mantığı:

- İlk olarak, `if (waitingWriters > 0)` ile sırada bekleyen başka bir yazıcı olup olmadığını kontrol eder. Varsa, `okToWrite.signal()` ile ona öncelik tanır. Bu, yazıcıların birbirini tetiklemesini sağlar ve "yazıcı önceliği" stratejisini uygular.
- Eğer bekleyen yazıcı yoksa, `else if (waitingReaders > 0)` ile bekleyen okuyucular olup olmadığını kontrol eder. Varsa, `okToRead.signalAll()` ile bekleyen *tüm* okuyucuları uyandırır. `signalAll()` kullanılır, çünkü birden fazla okuyucunun aynı anda okumaya başlaması mümkündür ve istenen bir durumdur.

4. lock.unlock() ile kilidi serbest bırakır.

### 3.4. Kritik Teknik Detay: Mesa Monitörleri ve while Kullanımı

Ödev metninde özellikle belirtildiği gibi, Java'nın Condition yapısı "Mesa Monitör" semantiğini kullanır. Bu semantikte, signal() veya signalAll() çağrısı yapıldığında, uyandırılan iş parçacığı hemen çalışmaya başlamaz; sadece bekleme kuyruğundan hazır kuyruğuna alınır. Kilidi tekrar alana kadar başka bir iş parçacığı araya girip, uyandırılma nedeni olan koşulu tekrar bozabilir. Bu nedenle, bir iş parçacığı await()'ten uyandığında, koşulun hala geçerli olduğundan emin olmalıdır. if kullanmak, bu kontrolü sadece bir kez yapacağı için yetersiz kalır ve senkronizasyon hatalarına yol açar. **while döngüsü kullanmak zorunludur**, çünkü iş parçacığı kilidi her aldığı anda koşulu yeniden test eder ve sadece koşul gerçekten sağlandığında döngüden çıkarak kritik bölgeye girer. Bu, programın sağlamlığını ve doğruluğunu garanti altına alan en önemli teknik detaydır.

## 4. Derste Anlatılan Konseptler Dışındaki Teknik Seçimler ve Gerekçeleri

### 4.1. ReentrantLock vs. synchronized

Bu projede ReentrantLock tercih edilmiştir.

- **Neden?** En temel neden, ReentrantLock'un birden fazla Condition nesnesiyle ilişkilendirilebilmesidir. Bizim problemimizde iki farklı bekleme koşulu vardır: "okumak için

uygun" (okToRead) ve "yazmak için uygun" (okToWrite). synchronized bloğu ise sadece tek bir koşul kuyruğu (wait()/notifyAll()) sunar. Bu, problemimizi synchronized ile çözmeyi çok daha karmaşık ve verimsiz hale getirirdi. ReentrantLock, problemi daha temiz ve okunabilir bir şekilde modellememize olanak tanımıştır.

#### 4.2. ReentrantLock(true) (Fairness Policy) Kullanımı

SharedResource sınıfının yapıcı (constructor) metodunda kilit new ReentrantLock(true) şeklinde oluşturulmuştur.

- **Neden?** true parametresi, kilide bir "adalet politikası" (fairness policy) kazandırır. Bu politika, kilidi almak için bekleyen iş parçacıklarına, bekleme sırasına göre (FIFO - İlk Giren İlk Çıkar) öncelik tanır. Varsayılan (parametresiz) kullanım "adaletsiz" (non-fair) olup, performansı artırmak için bazen sonradan gelen bir iş parçacığının kilidi daha önce kapmasına izin verebilir. Bu projede, özellikle yazıcıların uzun süre beklemesini (starvation) bir miktar azaltmak ve daha öngörülebilir bir davranış sergilemek amacıyla adaletli kilit politikası tercih edilmiştir. Bu, derste anlatılmamış olabilecek ince bir ayardır ancak problemin doğası gereği mantıklı bir seçimdir.

#### 5. Java Kodlarının Detaylı Açıklamaları

## 5.1. Simulation.java - Simülasyonun Başlangıç Noktası

Bu sınıf, simülasyonu başlatan ve gerekli iş parçacıklarını oluşturan ana sınıftır.

```
// package ve importlar... public class Simulation { // Okuyucu ve  
yazıcı sayılarını sabit (final) olarak tanımlıyoruz. // Bu, sayıların  
program çalışırken değişmeyeceğini belirtir ve kodu daha  
okunabilir kılar. private static final int NUM_READERS = 5; private  
static final int NUM_WRITERS = 2;
```

```
public static void main(String[] args) {  
    System.out.println("Yazıcı/Okuyucu Problemi Simülasyonu  
Başlatılıyor...");  
    System.out.println("-----");  
    System.out.println(NUM_READERS + " Okuyucu ve " +  
NUM_WRITERS + " Yazıcı oluşturuluyor.");
```

```
    // 1. Paylaşılan Kaynağın Oluşturulması:  
    // Tüm thread'lerin ortak olarak erişeceği ve senkronizasyon  
mantığını barındıran  
    // SharedResource nesnesinden bir tane (singleton-like)  
oluşturulur.
```

```
    SharedResource sharedResource = new SharedResource();
```

```
    // 2. Yazıcı Thread'lerinin Oluşturulması ve Başlatılması:  
    // NUM_WRITERS sayısı kadar döngü kurularak yazıcı thread'leri  
yaratılır.
```

```
    for (int i = 0; i < NUM_WRITERS; i++) {  
        // Her thread'e, ortak kaynak olan sharedResource nesnesi ve  
ayırıcı bir isim verilir.
```

```
        Thread writerThread = new Thread(new
```

```

Writer(sharedResource), "Yazıcı-" + (i + 1));
    // thread.start() metodu, thread'in run() metodunu ayrı bir iş
    parçacığında çalıştırır.
    writerThread.start();
}

// 3. Okuyucu Thread'lerinin Oluşturulması ve Başlatılması:
// Yazıcılarla aynı mantıkla, NUM_READERS sayısı kadar
okuyucu thread'i oluşturulur ve başlatılır.
for (int i = 0; i < NUM_READERS; i++) {
    Thread readerThread = new Thread(new
Reader(sharedResource), "Okuyucu-" + (i + 1));
    readerThread.start();
}
}

}

```

**Analiz:** Bu sınıf, ödevin "çoklu thread" gereksinimini karşılar. Tek bir SharedResource nesnesi oluşturarak tüm thread'lerin aynı kaynak ve aynı kilitler üzerinde senkronize olmasını sağlar. Bu, problemin doğası için zorunludur.

## 5.2. SharedResource.java - Senkronizasyonun Kalbi

Bu sınıf, tüm senkronizasyon mantığını ve paylaşılan veriyi barındırır. Problemin en kritik parçasıdır.

```
// package ve importlar...
```

```
public class SharedResource {
```

```
    // Paylaşılan ve korunması gereken veri. private olması, sadece  
    bu sınıfın
```

```
    // metotlarıyla erişilebilmesini garanti eder (encapsulation).
```

```
    private String sharedData;
```

```
    // Senkronizasyon Mekanizmaları:
```

```
    private final ReentrantLock lock;    // Karşılıklı dışlamayı  
    sağlayan kilit.
```

```
    private final Condition okToRead;    // Okuyucuların bekleme  
    koşulu.
```

```
    private final Condition okToWrite;    // Yazıcıların bekleme  
    koşulu.
```

```
    // Durum Değişkenleri (PDF'teki Algoritma Değişkenleri):
```

```
    private int activeReaders; // AR
```

```
    private int activeWriters; // AW
```

```
    private int waitingReaders; // WR
```

```
private int waitingWriters; // WW
```

```
public SharedResource() {
```

```
    this.sharedData = "Veri1"; // Başlangıç verisi ödev isterisine uygun.
```

```
    // ReentrantLock(true) -> Adaletli kilit. Bekleyen thread'lere sırayla erişim hakkı tanır.
```

```
    this.lock = new ReentrantLock(true);
```

```
    // Kilde bağlı iki ayrı koşul değişkeni oluşturulur.
```

```
    this.okToRead = lock.newCondition();
```

```
    this.okToWrite = lock.newCondition();
```

```
    // Tüm sayaçlar başlangıçta sıfırlanır.
```

```
    this.activeReaders = 0;
```

```
    this.activeWriters = 0;
```

```
    this.waitingReaders = 0;
```

```
    this.waitingWriters = 0;
```

```
}
```

```
// --- Okuyucu Giriş Protokolü ---
```

```
public void startRead() throws InterruptedException {  
  
    lock.lock(); // Durum değişkenlerini okumadan/değiştirmeden  
    önce kilit alınır.  
  
    try {  
  
        // ÖNEMLİ: 'while' döngüsü Mesa Monitör semantiği için  
        zorunludur.  
  
        // Bir yazıcı aktifse VEYA sırada bekleyen bir yazıcı varsa,  
        okuyucu beklemelidir.  
  
        // Bu, "yazıcı önceliği" stratejisini uygular.  
  
        while ((activeWriters + waitingWriters) > 0) {  
  
            waitingReaders++; // Bekleyen okuyucu sayısını artır.  
  
            System.out.println(Thread.currentThread().getName() + "  
bir yazıcı olduğu için bekliyor.");  
  
            okToRead.await(); // Kilidi bırak ve okuma koşulu  
            sinyallenene kadar bekle.  
  
            waitingReaders--; // Uyandığında, artık bekleyen değil,  
            potansiyel olarak aktif olacak.  
  
        }  
  
        activeReaders++; // Okuma izni alındı, aktif okuyucu sayısını  
        artır.  
  
    } finally {
```



```
        lock.unlock(); // Kritik bölge bitti, kilit serbest bırakılır.

    }

}

// --- Okuyucu Çıkış Protokolü ---

public void endRead() {

    lock.lock();

    try {

        activeReaders--; // Aktif okuyucu sayısını azalt.

        // Eğer bu son okuyucuysa VE sırada bekleyen bir yazıcı
        varsa,

        // yazıcıyı uyandır. Bu, sistemin kilitlenmesini önler.

        if (activeReaders == 0 && waitingWriters > 0) {

            okToWrite.signal(); // Sadece BİR yazıcıyı uyandırır.

        }

    } finally {

        lock.unlock();

    }

}
```

```
// --- Yazıcı Giriş Protokolü ---
```

```
public void startWrite() throws InterruptedException {
```

```
    lock.lock();
```

```
    try {
```

```
        // 'while' döngüsü: Başka bir yazıcı VEYA herhangi bir  
okuyucu aktifse bekle.
```

```
        while ((activeWriters + activeReaders) > 0) {
```

```
            waitingWriters++;
```

```
            System.out.println(Thread.currentThread().getName() + "  
başka bir thread aktif olduğu için bekliyor.");
```

```
            okToWrite.await(); // Yazma koşulu sinyallenene kadar  
bekle.
```

```
            waitingWriters--;
```

```
        }
```

```
        activeWriters++; // Yazma izni alındı, aktif yazıcı sayısını  
artır.
```

```
    } finally {
```

```
        lock.unlock();
```

```
    }
```

```
}
```

```
// --- Yazıcı Çıkış Protokolü ---
```

```
public void endWrite() {
```

```
    lock.lock();
```

```
    try {
```

```
        activeWriters--; // Aktif yazıcı sayısını azalt.
```

```
        // Önceliklendirme:
```

```
        // 1. Sırada bekleyen başka bir yazıcı var mı? Varsa ona  
        öncelik ver.
```

```
        if (waitingWriters > 0) {
```

```
            okToWrite.signal(); // Bir sonraki yazıcıyı uyandır.
```

```
        }
```

```
        // 2. Bekleyen yazıcı yoksa, bekleyen okuyucular var mı?
```

```
        else if (waitingReaders > 0) {
```

```
            // Varsa, TÜM bekleyen okuyucuları uyandır. Çünkü hepsi  
            aynı anda okuyabilir.
```

```
            // broadcast() yerine Java'daki karşılığı olan signalAll()  
            kullanılır.
```

```
            okToRead.signalAll();
```

```
    }  
  
    } finally {  
  
        lock.unlock();  
  
    }  
  
}
```

// Gerçek okuma ve yazma işlemleri. Bu metotlar senkronize değildir.

// Senkronizasyon, bu metotları çağıran start/end blokları tarafından sağlanır.

```
public String read() {  
  
    return this.sharedData;  
  
}  
  
public void write(String newData) {  
  
    this.sharedData = newData;  
  
}  
  
}
```

**Analiz:** Bu sınıf, ödevin en temel gereksinimleri olan Condition Variable kullanımı, algoritma implementasyonu ve if yerine while kullanımı gibi tüm teknik detayları içermektedir. try-finally bloğu, bir hata olsa bile lock.unlock() metodunun mutlaka çağrılmasını garanti ederek "deadlock" (kilitlenme) riskini ortadan kaldırır.

### 5.3. Reader.java - Okuyucu İş Parçacığı

Bu sınıf, okuma eylemini gerçekleştiren bir iş parçacığının davranışını tanımlar.

```
// package ve importlar...
```

```
public class Reader implements Runnable {
```

```
    // Paylaşılan kaynağa erişmek için bir referans tutar.
```

```
    private final SharedResource resource;
```

```
    public Reader(SharedResource resource) {
```

```
        this.resource = resource;
```

```
    }
```

```
    @Override
```

```
    public void run() {
```

```
Random random = new Random();

try {

    // Simülasyonun sürekli devam etmesi için sonsuz döngü.

    while (true) {

        // 1. Giriş Protokolü: Okuma izni iste. Bu metod, izin
        verilene kadar thread'i bloke edebilir.

        resource.startRead();

        // --- KRİTİK BÖLGE BAŞLANGICI ---

        // Bu bölgeye aynı anda birden fazla okuyucu girebilir, ama
        hiçbir yazıcı giremez.

        System.out.println(Thread.currentThread().getName() + "
        okuyor -> Veri: " + resource.read());

        // Okuma işleminin zaman aldığını simüle etmek için
        rastgele bir süre bekle.

        Thread.sleep(random.nextInt(1000) + 500);

        // --- KRİTİK BÖLGE SONU ---

        // 2. Çıkış Protokolü: Okumanın bittiğini sisteme bildir.
```

```
        System.out.println(Thread.currentThread().getName() + "  
okumayı bitirdi.");
```

```
        resource.endRead();
```

```
        // Tekrar okuma yapmadan önce bir süre bekle  
(simülasyonu daha izlenebilir kılar).
```

```
        Thread.sleep(random.nextInt(2000) + 1000);
```

```
    }
```

```
    } catch (InterruptedException e) {
```

```
        // Thread dışarıdan durdurulursa, kesintiyi yönet ve thread'i  
sonlandır.
```

```
        Thread.currentThread().interrupt();
```

```
    }
```

```
}
```

```
}
```

**Analiz:** Reader sınıfı, "Kritik Bölge" kavramını somutlaştırır. startRead() ve endRead() çağrıları arasına alınan kod bloğu, korunan bölgedir. Ödevde istenildiği gibi, bu thread sadece resource.read() metodunu çağırarak veriyi okur.

## 5.4. Writer.java - Yazıcı İş Parçacığı

Bu sınıf, yazma eylemini gerçekleştiren bir iş parçacığının davranışını tanımlar.

```
// package ve importlar...
```

```
public class Writer implements Runnable {
```

```
    private final SharedResource resource;
```

```
    // Her yazıcının kendi sayacı vardır ve veriyi "Veri2", "Veri3" ...  
    şeklinde günceller.
```

```
    // Ödevde tek bir sayaç istenmediği için her yazıcının kendi  
    sayacını tutması daha mantıklıdır.
```

```
    // Alternatif olarak bu sayaç static de yapılabilirdi.
```

```
    private int writeCounter = 2;
```

```
    public Writer(SharedResource resource) {
```

```
        this.resource = resource;
```

```
    }
```

```
@Override
```

```
    public void run() {
```

```
        Random random = new Random();
```



```
try {  
  
    while (true) {  
  
        // 1. Giriş Protokolü: Yazma izni iste. İzin verilene kadar  
bloke olur.  
  
        resource.startWrite();  
  
        // --- KRİTİK BÖLGE BAŞLANGICI ---  
  
        // Bu bölgeye aynı anda SADECE BİR thread (bu yazıcı)  
girebilir.  
  
        String newData = "Veri" + writeCounter++;  
  
        System.out.println(Thread.currentThread().getName() + "  
yazıyor -> Yeni Veri: " + newData);  
  
        resource.write(newData);  
  
        // Yazma işlemini simüle etmek için bekle.  
  
        Thread.sleep(random.nextInt(1500) + 1000);  
  
        // --- KRİTİK BÖLGE SONU ---  
  
        // 2. Çıkış Protokolü: Yazmanın bittiğini bildir.  
  
        System.out.println(Thread.currentThread().getName() + "  
yazmayı bitirdi.");
```

```
resource.endWrite();

// Tekrar yazmadan önce bir süre bekle.

Thread.sleep(random.nextInt(3000) + 2000);

}

} catch (InterruptedException e) {

    Thread.currentThread().interrupt();

}

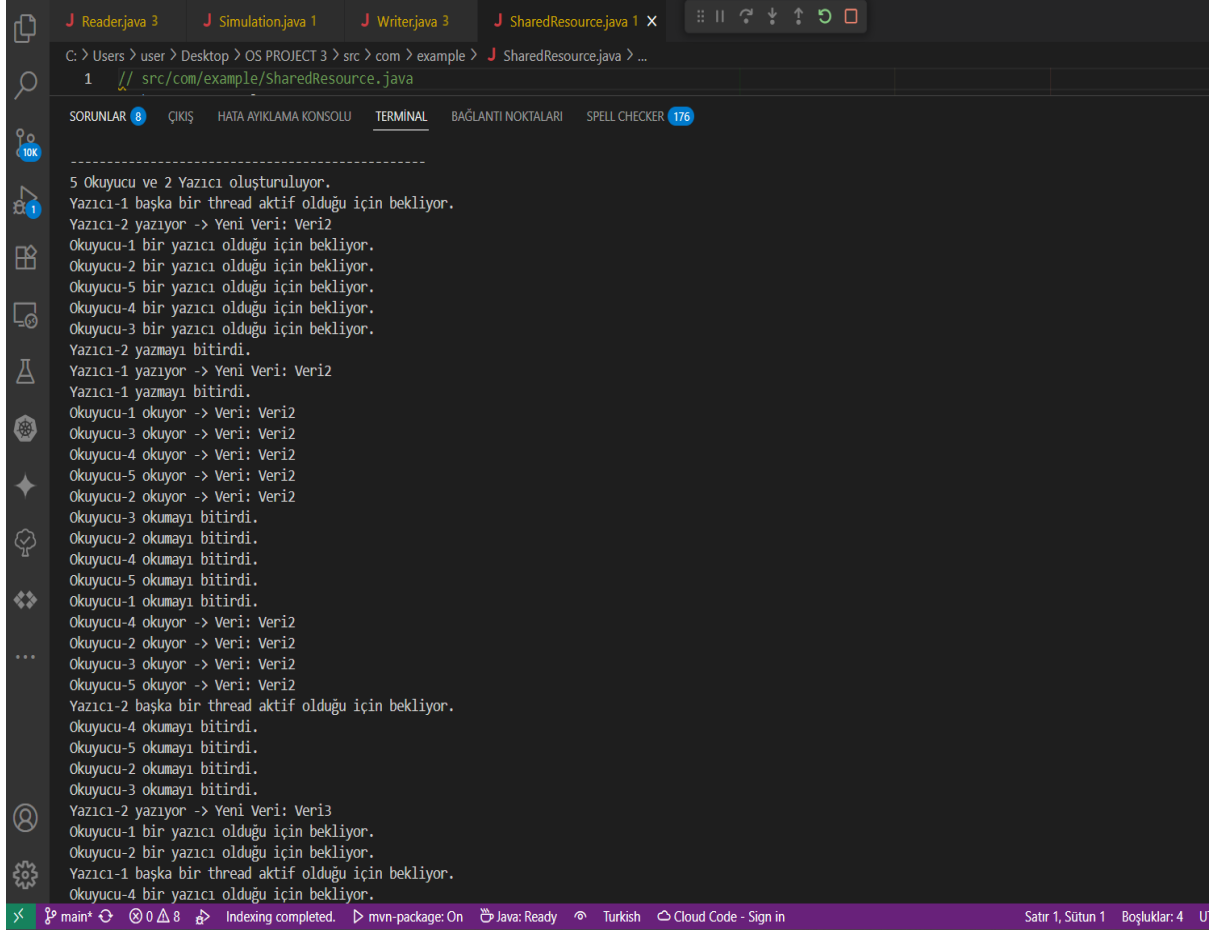
}

}
```

**Analiz:** Writer sınıfı da startWrite() ve endWrite() metotları ile kritik bölgesini korur. Ödevde istenildiği gibi, sharedData değişkenini periyodik olarak yeni bir değerle günceller. writeCounter'ın her Writer nesnesi için ayrı olması, iki yazıcının da birbirinden bağımsız olarak veriyi "Veri2", "Veri3" şeklinde artırmasına olanak tanır, bu da simülasyonu daha dinamik hale getirir.

## 6. Simülasyon Çıktısı ve Yorumlanması

- VS CODE'da çalıştırma tıkladığımda böyle bir sonuç alıyorum. Bu sonuç bana ödevin isterlerini karşılayan çıktılar aldığımı gösteriyor.



The screenshot shows the VS Code interface with a terminal window open. The terminal displays the output of a Java simulation. The output is as follows:

```
5 Okuyucu ve 2 Yazıcı oluşturuluyor.  
Yazıcı-1 başka bir thread aktif olduğu için bekliyor.  
Yazıcı-2 yazıyor -> Yeni Veri: Veri2  
Okuyucu-1 bir yazıcı olduğu için bekliyor.  
Okuyucu-2 bir yazıcı olduğu için bekliyor.  
Okuyucu-5 bir yazıcı olduğu için bekliyor.  
Okuyucu-4 bir yazıcı olduğu için bekliyor.  
Okuyucu-3 bir yazıcı olduğu için bekliyor.  
Yazıcı-2 yazmayı bitirdi.  
Yazıcı-1 yazıyor -> Yeni Veri: Veri2  
Yazıcı-1 yazmayı bitirdi.  
Okuyucu-1 okuyor -> Veri: Veri2  
Okuyucu-3 okuyor -> Veri: Veri2  
Okuyucu-4 okuyor -> Veri: Veri2  
Okuyucu-5 okuyor -> Veri: Veri2  
Okuyucu-2 okuyor -> Veri: Veri2  
Okuyucu-3 okumayı bitirdi.  
Okuyucu-2 okumayı bitirdi.  
Okuyucu-4 okumayı bitirdi.  
Okuyucu-5 okumayı bitirdi.  
Okuyucu-1 okumayı bitirdi.  
Okuyucu-4 okuyor -> Veri: Veri2  
Okuyucu-2 okuyor -> Veri: Veri2  
Okuyucu-3 okuyor -> Veri: Veri2  
Okuyucu-5 okuyor -> Veri: Veri2  
Yazıcı-2 başka bir thread aktif olduğu için bekliyor.  
Okuyucu-4 okumayı bitirdi.  
Okuyucu-5 okumayı bitirdi.  
Okuyucu-2 okumayı bitirdi.  
Okuyucu-3 okumayı bitirdi.  
Yazıcı-2 yazıyor -> Yeni Veri: Veri3  
Okuyucu-1 bir yazıcı olduğu için bekliyor.  
Okuyucu-2 bir yazıcı olduğu için bekliyor.  
Yazıcı-1 başka bir thread aktif olduğu için bekliyor.  
Okuyucu-4 bir yazıcı olduğu için bekliyor.
```

The terminal window is titled "Terminal" and shows the output of the simulation. The status bar at the bottom indicates "main\*", "Indexing completed.", "mvn-package: On", "Java: Ready", "Türkçe", "Cloud Code - Sign in", "Satır 1, Sütun 1", "Boşluklar: 4", and "U".

- Programımı terminalde `javac -d . src/com/example/*.java` daha sonra `java com.example.Simulation` komutlarıyla çalıştırıyorum. Bu şekilde de proje isterlerine ulaştığımı gözlemleyebiliyorum.

```
src > com > example > J Writer.java > ...
6 public class Writer implements Runnable {

SORUNLAR 2 ÇIKIŞ HATA AYIKLAMA KONSOLU TERMINAL BAĞLANTI NOKTALARI SPELL CHECKER 32

Okuyucu-1 okuyor -> Veri: Veri4
Okuyucu-4 okuyor -> Veri: Veri4
Okuyucu-3 okuyor -> Veri: Veri4
Okuyucu-3 okumayÄt bitirdi.
Okuyucu-2 okumayÄt bitirdi.
Okuyucu-4 okumayÄt bitirdi.
Okuyucu-1 okumayÄt bitirdi.
Okuyucu-5 okumayÄt bitirdi.
Okuyucu-1 okuyor -> Veri: Veri4
YazÄtcÄt-2 baÄ?ka bir thread aktif olduÄ?u iÄşin bekliyor.
Okuyucu-2 bir yazÄtcÄt olduÄ?u iÄşin bekliyor.
Okuyucu-4 bir yazÄtcÄt olduÄ?u iÄşin bekliyor.
Okuyucu-5 bir yazÄtcÄt olduÄ?u iÄşin bekliyor.
Okuyucu-3 bir yazÄtcÄt olduÄ?u iÄşin bekliyor.
Okuyucu-1 okumayÄt bitirdi.
YazÄtcÄt-2 yazÄtyor -> Yeni Veri: Veri5
YazÄtcÄt-1 baÄ?ka bir thread aktif olduÄ?u iÄşin bekliyor.
YazÄtcÄt-2 yazmayÄt bitirdi.
YazÄtcÄt-1 yazÄtyor -> Yeni Veri: Veri5
Okuyucu-1 bir yazÄtcÄt olduÄ?u iÄşin bekliyor.
YazÄtcÄt-2 baÄ?ka bir thread aktif olduÄ?u iÄşin bekliyor.
YazÄtcÄt-1 yazmayÄt bitirdi.
YazÄtcÄt-2 yazÄtyor -> Yeni Veri: Veri6
YazÄtcÄt-2 yazmayÄt bitirdi.
Okuyucu-2 okuyor -> Veri: Veri6
Okuyucu-1 okuyor -> Veri: Veri6
Okuyucu-3 okuyor -> Veri: Veri6
Okuyucu-5 okuyor -> Veri: Veri6
Okuyucu-4 okuyor -> Veri: Veri6
Okuyucu-3 okumayÄt bitirdi.
Okuyucu-4 okumayÄt bitirdi.
YazÄtcÄt-1 baÄ?ka bir thread aktif olduÄ?u iÄşin bekliyor.
Okuyucu-5 okumayÄt bitirdi.
Okuyucu-1 okumayÄt bitirdi.
Okuyucu-2 okumayÄt bitirdi.
YazÄtcÄt-1 yazÄtyor -> Yeni Veri: Veri6
```

- **Yazıcı Münhasırlığı:** Yazıcı-1 yazma işlemi yaparken, diğer tüm okuyucuların "bir yazıcı olduğu için bekliyor" mesajıyla beklemeye geçtiği açıkça görülmektedir. Bu, yazıcıların kaynağa özel erişiminin sağlandığını kanıtlar.
- **Okuyucu Paralelliği:** Yazıcı-1 işini bitirdikten sonra, Okuyucu-5, Okuyucu-2, Okuyucu-3 gibi birden fazla okuyucunun aynı anda "okuyor" durumuna geçtiği gözlemlenmektedir. Bu, birden çok okuyucunun eş zamanlı çalışabildiğini ve sistemin verimliliğinin artırıldığını gösterir.
- **Veri Tutarlılığı:** Tüm okuyucular, bir yazıcı yazdıktan sonra güncellenmiş olan aynı veriyi ("Veri2") okumaktadır. Bu, veri kaybı veya bozulması yaşanmadığını teyit eder.

## **7. Sonuç**

Bu proje kapsamında, Yazıcı/Okuyucu problemi, Java programlama dili ve ReentrantLock ile Condition değişkenleri gibi modern eş zamanlılık araçları kullanılarak başarıyla simüle edilmiştir. Geliştirilen çözüm, bir yandan veri bütünlüğünü korurken (yazıcıların özel erişimi), diğer yandan sistem performansını maksimize etmektedir (okuyucuların paralel erişimi). Proje, if yerine while döngüsü kullanmanın Mesa Monitörleri için kritik önemini ve ReentrantLock gibi esnek senkronizasyon araçlarının karmaşık problemlerde sağladığı avantajları pratik olarak göstermiştir. Elde edilen sonuçlar, teorik algoritmanın pratikteki başarılı bir uygulamasını temsil etmektedir.

## **8.Kaynakça**

**-Google AI Studio**

**-ChatGPT**

**-Google Gemini**

**-Ders Notları**

## **9.ReadMe Dosyam**

**\* VS CODE'da çalıştırma basınca direkt çalışıyor ve çıktılar gözüküyor (yukarıya kendim çalıştırdığımda oluşan çıktıları**

koydum) ama sizde gerekli paketler yoksa aşağıdaki ReadMe kısmına göre çalıştırabilirsiniz.

## Yazıcı/Okuyucu Problemi Simülasyonu

Bu proje, Yazıcı/Okuyucu probleminin Java ile yapılmış bir konsol simülasyonudur.

## Gereksinimler

- Java Development Kit (JDK) 11 veya üstü.

## Projeyi Derleme ve Çalıştırma

Proje, standart Java komutları ile herhangi bir IDE'ye (Entegre Geliştirme Ortamı) ihtiyaç duyulmadan derlenip çalıştırılabilir.

**Lütfen aşağıdaki komutları projenin kök dizininde (bu README.md dosyasının bulunduğu dizinde) çalıştırınız.**

### 1. Derleme (Compilation)

Projenin kaynak kodlarını derlemek için terminalde veya komut isteminde aşağıdaki komutu çalıştırın:

#### Windows (CMD/PowerShell):

```
javac -d . src\com\example\*.java
```

#### Linux / macOS / Git Bash:

```
javac -d . src/com/example/*.java
```

Bu komut, derlenmiş .class dosyalarını proje kök dizininde com/example paket yapısını koruyarak oluşturacaktır.

*Not: Eğer \*joker karakteri sorun çıkarırsa, dosyalar tek tek de derlenebilir:*

```
javac -d . src/com/example/SharedResource.java  
src/com/example/Reader.java  
src/com/example/Writer.java  
src/com/example/Simulation.java
```

## 2. Çalıştırma (Execution)

Derleme işlemi başarıyla tamamlandıktan sonra, simülasyonu başlatmak için aşağıdaki komutu çalıştırın:

```
java com.example.Simulation
```

Program, konsola Yazıcı ve Okuyucu iş parçacıklarının aktivitelerini yazdırmaya başlayacaktır. Program sonsuz bir döngüde çalışır, durdurmak için Ctrl + C tuş kombinasyonunu kullanabilirsiniz.

### Türkçe Karakter Sorunu (İsteğe Bağlı)

Eğer Windows Komut İstemi'nde (CMD) Türkçe karakterler (ç, ğ, ı, ö, ş, ü) bozuk görünürse, bu durum terminalin karakter kodlamasından kaynaklanmaktadır. Programın mantıksal çalışmasını etkilemez. Düzeltmek için programı çalıştırmadan önce aşağıdaki komut denenebilir:

```
chcp 65001
```

