A red text on a white background

Description automatically generated

MÜHENDİSLİK VE DOĞA BİLİMLERİ FAKÜLTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

**BİL-302 İŞLETİM SİSTEMLERİ ÖDEV-3**

**HAZIRLAYANLAR**

MERT TOSUN – 210601027

TESLİM TARİHİ

26.05.2024

ÖDEVİN AMACI

Bu ödevin amacı işletim sistemlerinde kaynak yönetimi ve eşzamanlılık konularını anlamak ve örnek bir sorunu çözerek pekiştirmektir.

İLK KISIM THREAD VE SEMAPHORE KULLANARAK

Thread: Java'da iş parçacıkları (threads), aynı program içinde eşzamanlı olarak çalışan bağımsız yürütme birimleridir. Birden fazla iş parçacığı aynı anda çalışarak programın farklı bölümlerini paralel olarak yürütebilir.

Semaphore: Semaforlar, eşzamanlı erişim kontrolü sağlamak için kullanılan senkronizasyon araçlarıdır. Belirli bir kaynağa erişimi sınırlamak için kullanılır.

Bir yazıcı kuyruğu simüle etmemiz isteniyor. Bu kuyruk belirli bir tampon belleğe sahiptir ve threadler yazdırılacak işleri bu kuyruğa ekleyip yazıcıya gönderebilir. Aynı anda birden fazla iş parçacığı bu kuyruğa erişmeye çalışabilir, bu yüzden erişimi kontrol etmek için semaforlar kullanılır.

* İşlerin tutulması için bir adet kuyruk yapısı(queue) kullanılır.



Burada final ifadesi bir değişkenin sadece tanımlandığı sınıftan erişilebilir ve değerinin değiştirilemez olduğunu belirtir.

* Eşzamanlı erişimi kontrol etmek, yazıcı biriktiricisinde bulunan işleri saymak ve yazıcı biriktiricisindeki boş alanı kontrol etmek için üç adet semaphore

metin, yazı tipi, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

Yeni iş eklenmeden önce, yazıcı biriktiricisinde yeterli boş alan olup olmadığını kontrol edilir. Eğer yeterli boş alan yoksa, bu satırda thread bekletilir. Yeterli alan olduğunda, işin boyutu kadar izin alınır ve thread devam eder. Anlık boş alan 100 KB ise ve eklenmek istenen iş 150 KB ise bu satırda thread bekletilir. (olmasını istediğiniz detay)



**Acquire ve Release Kavramları**

**Acquire:**

* Bir kaynağı **kullanım için** **elde etmek** anlamına gelir.
* Kaynak, bir bellek bloğu, bir donanım aygıtı, bir ağ bağlantısı veya başka bir sınırlı kaynak olabilir.
* Acquire işlemi genellikle bir **kilit** mekanizması veya **semafor** gibi bir senkronizasyon mekanizması aracılığıyla gerçekleştirilir.
* Bu, birden fazla kullanıcının aynı anda aynı kaynağa erişmesini önler ve veri tutarlılığını sağlar.

**Release:**

* Bir kaynağı **kullanımdan çıkarmak** ve **başkalarının kullanımına** **açmak** anlamına gelir.
* Kaynak artık ihtiyaç duyulmuyorsa veya başka bir işlem tarafından kullanılmaya hazır hale geldiğinde gerçekleştirilir.

**Senkronizasyonun Gerekliliği**

Yazıcı kuyruğu probleminde, birden fazla iş parçacığı (thread) aynı kaynakları paylaşarak eşzamanlı olarak çalışır. Burada kaynak, yazdırılacak işler için kullanılan tampon bellektir. İş parçacıkları, yazdırma işleri eklerken ve işler yazdırılırken tampon belleğe erişmek zorundadır.

**Neden Senkronizasyona İhtiyaç Var?**

Veri Bütünlüğü: Aynı anda birden fazla iş parçacığının tampon belleğe erişimi veri bütünlüğünü bozabilir. Örneğin, bir iş parçacığı iş eklerken diğeri iş alıyorsa, tampon belleğin durumu beklenmedik şekilde değişebilir.

Ölçeklenebilirlik ve Performans: Doğru senkronizasyon, iş parçacıklarının verimli bir şekilde çalışmasını ve sistemin performansının düşmemesini sağlar.

Deadlocks ve Race Condition: Senkronizasyon olmadan, iş parçacıkları arasında ölü kilitler (deadlocks) ve yarış koşulları (race conditions) gibi problemler ortaya çıkabilir.

**Job Ekleme (Producer) İşlemi:**

* İş parçacığı, yeni bir iş eklemek istediğinde önce space semaforunu kullanarak tampon bellekte yeterli alan olup olmadığını kontrol eder.
* Yeterli alan varsa mutex semaforunu alarak tampon belleğe erişir ve yeni işi ekler.
* İş eklendikten sonra items semaforu serbest bırakılarak yeni bir işin eklendiği belirtilir.

**Job Alma (Consumer) İşlemi:**

* Yazıcı iş parçacığı, yeni bir iş almak istediğinde items semaforunu kullanarak tampon bellekte mevcut iş olup olmadığını kontrol eder.
* Mevcut bir iş varsa mutex semaforunu alarak tampon belleğe erişir ve işi alır.
* İş alındıktan sonra space semaforu serbest bırakılarak tampon bellekte yer açıldığı belirtilir.

**Producer (Job Ekleme) Algoritması**

1. Tampon bellekte yeterli alan olup olmadığını kontrol et
2. Tampon belleğe erişim için mutex'i al
3. Yeni işi tampon belleğe ekle
4. Mutex'i serbest bırak
5. Yeni bir iş eklendiğini belirt

**Consumer (Job Alma) Algoritması**

1. Tampon bellekte mevcut iş olup olmadığını kontrol et
2. Tampon belleğe erişim için mutex'i al
3. Mevcut işlerden birini tampon bellekten al
4. Tampon bellekte yer açıldığını belirt
5. Mutex'i serbest bırak
6. Alınan işi döndür

**Sonuç**

Bu programımızla, yazıcı kuyruğu problemini Thread ve Semaphore kullanarak çözüyoruz. Senkronizasyon, iş parçacıklarının tampon belleğe güvenli ve verimli bir şekilde erişimini sağlar. Semaphore kullanımı, eşzamanlı iş parçacıkları arasında koordinasyonu sağlayarak veri bütünlüğünü korur ve deadlocks ile race conditions önler. Bu şekilde, yazıcı kuyruk simülasyonunu başarılı bir şekilde gerçekleştirebiliriz.

EKRAN ÇIKTISI

metin, yazı tipi, ekran görüntüsü, tipografi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**İKİNCİ KISIM MONİTOR YAPISI(CONDİTİON VARİABLE LOCKS)**

Monitor yapısı, senkronizasyonu daha kolay ve güvenli hale getirmek için condition variables ve locks kullanır.

**Neden Senkronizasyona İhtiyaç Var?**

* Veri Bütünlüğü: Aynı anda birden fazla iş parçacığının tampon belleğe erişimi veri bütünlüğünü bozabilir. Monitor yapısı, bu erişimi güvenli hale getirir.
* Ölçeklenebilirlik ve Performans: Doğru senkronizasyon, iş parçacıklarının verimli bir şekilde çalışmasını ve sistemin performansının düşmemesini sağlar.
* Ölü Kilitler ve Yarış Koşulları: Monitor yapısı, iş parçacıkları arasında ölü kilitler (deadlocks) ve yarış koşulları (race conditions) gibi problemleri önler.

**Senkronizasyon Geliştirme Adımları**

Monitor ve Condition Variables Tanımlanması:

* lock: Tampon belleğe erişimi kontrol eden kilit.
* notFull: Tampon belleğin dolu olup olmadığını kontrol eden condition variable.
* notEmpty: Tampon bellekte iş olup olmadığını kontrol eden condition variable.

**Job Ekleme (Producer) İşlemi:**

* İş parçacığı, yeni bir iş eklemek istediğinde önce kilidi alır.
* Eğer tampon bellek doluysa, notFull condition variable ile bekler.
* Yeterli alan olduğunda yeni iş tampon belleğe eklenir.
* İş eklendikten sonra notEmpty condition variable sinyal göndererek yeni bir işin eklendiğini belirtir.
* Kilit serbest bırakılır.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Job Alma (Consumer) İşlemi:**

* Yazıcı iş parçacığı, yeni bir iş almak istediğinde kilidi alır.
* Eğer tampon bellek boşsa, notEmpty condition variable ile bekler.
* Mevcut bir iş olduğunda iş tampon bellekten alınır.
* İş alındıktan sonra notFull condition variable sinyal göndererek tampon bellekte yer açıldığını belirtir.
* Kilit serbest bırakılır.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**ReentrantLock Nedir?**

Javanın java.util.concurrent.locks kütüphanesinde yer alan bir senkronizasyon anahtar kelimesi denilebilir. Kilitle alakalı durumları kontrol eder. Bir threadin bir kilidi birden fazla kez almasını sağlar.



**A VE B KISIMLARINDAKİ ÇÖZÜMLERİN FARKI**

A kısmında, senkronizasyon için semaforlar kullanılır. Semaforlar, belirli bir kaynağa erişim izni verir ve belirli sayıda iş parçacığının aynı anda kaynağa erişimini kontrol eder. Bu yöntemde, mutex semaforu kullanılarak tampon belleğe erişim kontrol edilir. İşlerin eklenip çıkarılmasına izin vermek için items ve space semaforları kullanılır. Semafor kullanımı, daha düşük seviyeli bir senkronizasyon kontrolü sağlar ve doğru kullanım için dikkat gerektirir. Yanlış kullanımı deadlocks ve race conditions yol açabilir.

B kısmında ise, senkronizasyon için monitor yapısı kullanılır. Monitor yapısı, senkronizasyonu daha üst seviyeli ve kolay anlaşılır hale getirir. Bu yöntemde, ReentrantLock kullanılarak tampon belleğe erişim kontrol edilir. Condition variables (notFull ve notEmpty), belirli koşulların sağlanmasını beklemek ve bu koşullar gerçekleştiğinde iş parçacıklarını uyandırmak için kullanılır. Monitor yapısı, daha yüksek seviyeli bir senkronizasyon kontrolü sağlar ve kodun daha okunabilir ve anlaşılır olmasını sağlar. Bu yapı, deadlocks ve race conditions gibi problemleri önlemeye yardımcı olur.

Kısacası, semaforlar düşük seviyeli kontrol sağlarken, monitor yapısı daha yüksek seviyeli ve kullanımı kolay bir kontrol sunar. Monitor yapısı, belirli bir koşulun oluşmasını beklemek için daha doğal bir yol sunar ve kodun okunabilirliğini artırır. Monitor yapısı, senkronizasyonun karmaşıklığını azaltmak ve hataya açık yapıları minimize etmek için daha iyi bir seçim olabilir. Her iki yapı da performans ve ölçeklenebilirlik açısından etkilidir, ancak monitor yapısı genellikle daha güvenli ve anlaşılırdır.

EKRAN ÇIKTISI

