

AĞ PROGRAMLAMA

DERS 3:

- Multi-Thread Programlama

Thread Senkronizasyonu

- Paylaşılan kaynakların thread'ler tarafından eş zamanlı okunması kaynaktaki güncelleme için hangi thread'in kullanımda olduğunu bilemeyeceğimiz durumlarda sıkıntı oluşturacaktır.
- Aynı kaynağa erişmeye çalışan iki thread birbirinden nasıl ayırtırılır?

Thread Senkronizasyonu

- Bir kaynağın belirli bir zaman diliminde (tekrar serbest bırakana kadar) sadece tek bir thread tarafından erişilmesine izin vermek bu problemi giderebilir.
- Bu yaklaşım thread senkronizasyonunu sağlar.

Monitor(ler)

- Java'da en yaygın kullanılan senkronizasyon yöntemidir.
- Paylaşılan objeler için bir monitor lock(kilit) kullanılır.
- Bu sayede bir thread aktif erişimde iken kaynağı kendisi için kilitlemiş olur.
- Kilitlenmiş bir kaynağa erişmeye çalışan diğer threadler **blocked** konumuna geçerler.
- Bir thread'in belirli bir kod bloğunu monitor lock kullanarak kilitlemesi için ilgili kod parçası **synchronized statement** içerisinde yazılır.

Monitor(ler)

```
synchronized ( object )
{
    statements
} // end synchronized statement
```

- **Object** : monitor lock'ın kilitleyeceği nesneyi ifade eder. Genelde bulunduğu nesneyi kilitleyeceği için **this** olarak kullanılır.
- Thread ilgili object ile işini bitirdiğinde blocked durumdaki diğer threadlerden bir tanesi çalıştırılır.
- **Synchronized method**'lar da kullanılabilir

Senkronize edilmeyen Thread Örneği

- Bölüm sayfasındaki örnek kodlar: Deitel ch26 Fig. 26-5 SimpleArray.java
- Satır 21,34 ve 38'deki işlemlere dikkat ediniz.
- Fig. 26-6 : ArrayWriter.java
- SimpleArray sınıfını işleyecek Thread türevimiz.
- Fig. 26-7: SharedArrayTest

Senkronize edilmeyen Thread Örneği

```
pool-1-thread-1 wrote 1 to element 0.  
Next write index: 1  
pool-1-thread-1 wrote 2 to element 1.  
Next write index: 2  
pool-1-thread-1 wrote 3 to element 2.  
Next write index: 3  
pool-1-thread-2 wrote 11 to element 0.  
Next write index: 4  
pool-1-thread-2 wrote 12 to element 4.  
Next write index: 5  
pool-1-thread-2 wrote 13 to element 5.  
Next write index: 6
```

First pool-1-thread-1 wrote the value 1 to element 0. Later pool-1-thread-2 wrote the value 11 to element 0, thus *overwriting* the previously stored value.

Contents of SimpleArray:
[11, 2, 3, 0, 12, 13]

Senkronize Thread Örneği

- Fig.26-8 : SimpleArray sınıfının senkronize çalışma özelliği eklenmiş hali.
- Satır 20. Add fonksiyonunun synchronized anahtar kelimesi ile eş zamanlı erişime kısıtlanması

Senkronize Thread Örneği

```
pool-1-thread-1 wrote 1 to element 0.  
Next write index: 1  
pool-1-thread-2 wrote 11 to element 1.  
Next write index: 2  
pool-1-thread-2 wrote 12 to element 2.  
Next write index: 3  
pool-1-thread-2 wrote 13 to element 3.  
Next write index: 4  
pool-1-thread-1 wrote 2 to element 4.  
Next write index: 5  
pool-1-thread-1 wrote 3 to element 5.  
Next write index: 6
```

Contents of SimpleArray:

1 11 12 13 2 3

Monitor(ler)

- Synchronized anahtar kelimesinin fonksiyona uygulanması sonucu fonksiyon içerisindeki tüm alt adımlar tek bir işlem şeklinde ele alınmış oldu.
- Synchronized blokları oluştururken dikkatli olunmalıdır.
 - Mمкнn olduğunca kыsa, iшlemlerin çabuk bitebileceği boyutlarda olmalı
 - Synchronized blou ierisinde sleep gibi komutlar yer almamalı (Bu ornekte sadece threadleri birbiri ile cakistirmak iin kullanılmıştir.)

Senkronizasyon olmaksızın Üretici(Producer)/Tüketici (Consumer) İlişkisi

- Örnek üretici tüketici ilişkisi durumu «print spooling»
- Multi-Thread uygulamalarda producer/consumer ilişkisi için tampon (buffer) bellek kullanılır.
- Üretici thread tampona veri yazar ve tüketici thread tampondaki veriyi okuyarak işler.
- Buffer üzerinde kayıplar oluşmaması için üretici / tüketici threadlerin senkronize edilmesi gereklidir.
- Fig 26. 9-13 Üretici /Tüketici yapısında senkronizasyon kullanılmayan örnek.

Senkronizasyon olmaksızın Üretici(Producer)/Tüketicisi (Consumer) İlişkisi

Action	Value	Sum of Produced	Sum of Consumed	
Producer writes	1	1		
Producer writes	2	3		— 1 is lost
Producer writes	3	6		— 2 is lost
Consumer reads	3		3	
Producer writes	4	10		
Consumer reads	4		7	
Producer writes	5	15		
Producer writes	6	21		— 5 is lost
Producer writes	7	28		— 6 is lost
Consumer reads	7		14	
Consumer reads	7		21	— 7 read again
Producer writes	8	36		
Consumer reads	8		29	
Consumer reads	8		37	— 8 read again
Producer writes	9	45		
Producer writes	10	55		— 9 is lost
Producer done producing				
Terminating Producer				
Consumer reads	10		47	
Consumer reads	10		57	— 10 read again
Consumer reads	10		67	— 10 read again
Consumer reads	10		77	— 10 read again
Consumer read values totaling 77				
Terminating Consumer				

ArrayBlockingQueue ile Üretici/Tüketici İlişkisi

- Package: `java.util.concurrent`
- Thread-safe buffer class
- Implements : `BlockingQueue`
- Multi-thread uygulamalarında ortak buffer kullanımları için en iyi çözümlerden biridir.
- Put ve take komutları. İlgili thread için kuyruğun boş veya dolu olma durumunu kontrol ederek işlemeyi sağlar.
- Fig 26.14–26.15 örneklerini inceleyelim.

ArrayBlockingQueue ile Üretici/Tüketicisi İlişkisi

```
Producer writes 1      Buffer cells occupied: 1
Consumer reads 1      Buffer cells occupied: 0
Producer writes 2      Buffer cells occupied: 1
Consumer reads 2      Buffer cells occupied: 0
Producer writes 3      Buffer cells occupied: 1
Consumer reads 3      Buffer cells occupied: 0
Producer writes 4      Buffer cells occupied: 1
Consumer reads 4      Buffer cells occupied: 0
Producer writes 5      Buffer cells occupied: 1
Consumer reads 5      Buffer cells occupied: 0
Producer writes 6      Buffer cells occupied: 1
Consumer reads 6      Buffer cells occupied: 0
Producer writes 7      Buffer cells occupied: 1
Consumer reads 7      Buffer cells occupied: 0
Producer writes 8      Buffer cells occupied: 1
Consumer reads 8      Buffer cells occupied: 0
Producer writes 9      Buffer cells occupied: 1
Consumer reads 9      Buffer cells occupied: 0
Producer writes 10     Buffer cells occupied: 1
```

Producer done producing

Terminating Producer

Consumer reads 10 Buffer cells occupied: 0

Consumer read values totaling 55

Terminating Consumer

Senkronizasyonlu Üretici/Tüketici İlişkisi

- ArrayBlockingQueue benzeri bir yapıyı synchronized kullanarak nasıl tasarılayabiliriz?
- Buffer için set ve get methodları synchronized ile kapsanarak yazılır.
- Buffer doluluk ve boş olma kontrollerinin ayrıca eklenmesi gereklidir.
- Beklemesi gereken bir thread ihtiyaç duyulduğunda waiting state'ine geçilmelidir.

Wait , Notify ve Notify All methodları

- Bir threadin monitor lock ile kilitlediği bir nesne ile işinin bitmesi belirli bazı durumlar gerektiriyorsa **wait** komutu ile ilgili nesneyi kilitli bırakabilir.
- Eğer thread işlemini bitirmiş monitor lock ile elinde olan nesneyi başka bir thread'in kullanımına bırakacak ise bunu **notify** ile sağlayabilir.
- Eğer monitor lock ile kilitlenmiş nesne tüm diğer threadlerin kullanımı için serbest bırakılmak istenirse bunu **notifyAll** ile sağlayabiliriz.
- Aynı anda bir nesneye sadece bir thread monitor lock uygulayabilir. Bu arada erişmek isteyen diğer threadlerin blocked konumda kalacaklarını unutmamak gereklidir.
- Fig. 26.16 ve Fig. 26.17'de örnek uygulamayı inceleyelim.

Wait , Notify ve Notify All methodları

Operation	Buffer	Occupied
Consumer tries to read. Buffer empty. Consumer waits.	-1	false
Producer writes 1	1	true
Consumer reads 1	1	false
Consumer tries to read. Buffer empty. Consumer waits.	1	false
Producer writes 2	2	true
Consumer reads 2	2	false
Producer writes 3	3	true
Consumer reads 3	3	false
Producer writes 4	4	true

Bounded Buffers ile Üretici/Tüketici İlişkisi

- İki threadin üretim ve tüketim hızları daima aynı olmayabilir.
- «outofsync» durumunda çoğunlukla threadler wait durumunda kalacaktır.
- Wait durumunda kalan threadler uygulamanın verimliliğini düşürür.
- Bu durum nasıl en iyileştirilebilir?

Bounded Buffers ile Üretici/Tüketici İlişkisi

- Çözüm: Bir işlem için belirli sınırları olan **bounded buffer** kullanımıdır.
- Eğer üretici ve tüketici hızları birbirine göre aşırı farklılık gösteriyorsa sadece bounded buffer kullanımı yeterli olmayacağındır.
- Bounded buffer oluşturmak için iyi bir alternatif `ArrayBlockingQueue`'yu constructor'ında belirli sınırlarla tanımlamaktır.
- Sıfırdan kendimiz tanımladığımız circular buffer örneğini inceleyelim. Fig. 26.18 ve Fig. 26.19

Üretici / Tüketici İlişkisi Lock ve Condition Interface(leri)

- synchronized anahtar kelimesi dışında da thread yönetim yapıları mevcuttur. Bunun için geliştirilmiş iki interface'i inceleyeceğiz.

Interface Lock ve Class ReentrantLock

- package java.util.concurrent.locks
- Lock mekanizmasını ilgili interface’i implemente eden bütün yapılar kullanabilir. Daha önce de bahsettiğimiz gibi bir thread bir nesne için lock uyguladıysa unlock uygulayana kadar buna diğer threadler erişemez ve bu lock için waiting state de beklerler.

Interface Lock ve Class ReentrantLock

- Class ReentrantLock lock interface'ini implemente etmiş bir örnektir.
- Constructor'ı fairness policy için kullanılan boolean bir değişken alır.
- True olursa en uzun bekleyen thread lock'u kapar.

Condition Objects ve Interface Condition

- Eğer lock'un sahibi olan thread işlemi belli bir durum gerçekleşene kadar bitiremeyeceğini tespit ederse bir **condition object** üzerinde bekleme yapabilir.
- Üretici /Tüketicisi yapsında üretici bir nesnede beklerken tüketici başka bir nesnede bekleyebilir.
- Condition objeleri sadece bir lock ile ilişkili olabilir.
- Lock'un **newCondition** methodu çağrılarak üretilirler.
- Bir condition objesini beklemek için thread condition'ın **await** methodunu çağırır.
- Bu işlem ilgili Lock'u iptal eder ve thread'i ilgili condition için waiting konumuna sokar.
- Runnable durumdaki başka bir thread işlemini bitirdiğinde waiting durumundaki bir threadin çalışabileceğini görürse condition'ın **signal** methodunu çağrıarak diğer threadyi aktif hale getirir.
- Eğer thread hala işini bitiremeyecek durumda ise lock'ı tekrar serbest bırakmak için await methodunu çağrıabilir.
- Aynı anda birden fazla thread waiting modda ilgili condition'ı bekliyorsa **signal** sonucu en uzun bekleyen lock'ı devralır. (**signalAll**)

Lock, Condition v.s. Synchronized

- Lock çalışan bir thread'i interrupt etme, veya bir lock için belli bir timeout verme gibi özellikler sağlar. (synchronized ile mümkün olmayan)
- Lock'in kilitleme ve serbest bırakma kodları aynı block içerisinde olmak zorunda değildir.
- Condition, bekleyen bir thread için çoklu koşul koymamıza imkan verir. Synchronized ile hangi threadlerin waiting durumunda olduğunu takip etmemiz ve ilgili bir thread için notify kullanarak condition'ın uygun olduğu gibi bir bildirim yapma şansımız yoktur.

Lock ve Condition ile Synchronization Gerçeklemesi

- Fig 26.20 ve Fig 26.21 örneklerini inceleyelim

Concurrent Collections

- [download.oracle.com/javase/6/docs/api/
java/util/concurrent/package-summary.html](http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html)
- [download.java.net/jdk7/docs/api/java/util/
concurrent/package-summary.html](http://download.java.net/jdk7/docs/api/java/util/concurrent/package-summary.html)

Concurrent Collections

Collection	Description
<code>ArrayBlockingQueue</code>	A fixed-size queue that supports the producer/consumer relationship—possibly with many producers and consumers.
<code>ConcurrentHashMap</code>	A hash-based map that allows an arbitrary number of reader threads and a limited number of writer threads.
<code>ConcurrentLinkedQueue</code>	A concurrent linked-list implementation of a queue that can grow dynamically.
<code>ConcurrentSkipListMap</code>	A concurrent map that is sorted by its keys.
<code>ConcurrentSkipListSet</code>	A sorted concurrent set.
<code>CopyOnWriteArrayList</code>	A thread-safe <code>ArrayList</code> . Each operation that modifies the collection first creates a new copy of the contents. Used when the collection is traversed much more frequently than the collection's contents are modified.
<code>CopyOnWriteArrayList</code>	A set that's implemented using <code>CopyOnWriteArrayList</code> .
<code>DelayQueue</code>	A variable-size queue containing <code>Delayed</code> objects. An object can be removed only after its delay has expired.
<code>LinkedBlockingDeque</code>	A double-ended blocking queue implemented as a linked list that can optionally be fixed in size.
<code>LinkedBlockingQueue</code>	A blocking queue implemented as a linked list that can optionally be fixed in size.
<code>PriorityBlockingQueue</code>	A variable-length priority-based blocking queue (like a <code>PriorityQueue</code>).
<code>SynchronousQueue</code>	A blocking queue implementation that does not have an internal capacity. Each insert operation by one thread must wait for a remove operation from another thread and vice versa.

GUI ile Multi-Thread Uygulamaları

- Swing Multi-Thread uygulama geliştirmek için ek özellikler sunar.
- GUI üzerindeki tüm işlemler **event dispatch thread** olarak adlandırılan tek bir thread ile gerçekleştirilir.
- Arayüzde yaptığımız tüm işlemler bir **event queue** üzerinden sıra ile yapılır.
- Swing GUI bileşenleri **non thread-safe** (thread korumasız)dır.
- Thread koruması önceki örneklerdeki gibi **synchronized** ile sağlanamaz.
- Bileşenlere güvenli erişimin garanti edilmesi için Swing tek thread'in bileşenlere erişmesine izin verir(**thread confinement**).

Class SwingWorker

- javax.swing
- Arayüz işlemlerinden ayrı çalışacak bir thread oluşturmamızı sağlar.
- İçerdiği methodlar:

Method	Description
doInBackground	Defines a long computation and is called in a worker thread.
done	Executes on the event dispatch thread when <code>doInBackground</code> returns.
execute	Schedules the <code>SwingWorker</code> object to be executed in a worker thread.
get	Waits for the computation to complete, then returns the result of the computation (i.e., the return value of <code>doInBackground</code>).
publish	Sends intermediate results from the <code>doInBackground</code> method to the <code>process</code> method for processing on the event dispatch thread.
process	Receives intermediate results from the <code>publish</code> method and processes these results on the event dispatch thread.
setProgress	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.

Class SwingWorker

- Fig 26.24 : Fibonacci Serisini hesaplayan örnek. (`dolnBackground` , `done` , `get`) kullanımı örneği.
- SwingWorker Generic bir Class'dır.
- `SwingWorker< Long, Object >`
 - Birinci Parametre: `dolnBackground`'un dönüş değeri
 - İkinci Parametre: Ara sonuçların gösterimi için `publish – process` methodları arasında gönderilen bilgiyi temsil eder. (bu örnekte kullanılmadı)

ClassSwingWorker

- Fig 26.26 Asal sayı hesaplama:
- Ara sonuç bilgisinin gösterimi.
(setProgress , publish , process) kullanımı
örneği.