BMB214 Programlama Dilleri Prensipleri

Ders 13. Eşzamanlılık (Concurrency)

Erdinç Uzun
Tekirdağ Namık Kemal Üniversitesi
Bilgisayar Mühendisliği Bölümü

Konular

- Concurrency (Eşzamanlılık)
- Alt Program Düzeyinde Eşzamanlılığa Giriş
- Semaforlar (Semaphores)
- Monitors
- İleti geçişi Eşzamanlılık için Ada desteği Java Konuları C # Konu İşlevsel Dillerde Eş Zamanlılık Bildirim Düzeyinde Eş Zamanlılık
- Mesaj Geçişi (Message Passing)
- Ada Eşzamanlılık Desteği
- Java Thread'ler
- C# Thread'ler
- Statement Seviyesinde (Level)Eşzamanlılık
- Diğer Dillerde Eşzamanlılık

Eşzamanlılık

- Eşzamanlılık dört düzeyde gerçekleşebilir:
  - Makine komut seviyesi (Machine instruction level)
  - Yüksek seviyeli dil statement seviyesi
  - Birim seviyesi (Unit level)
  - Program seviyesi (Program level)
- Komut ve program düzeyinde eşzamanlılıkta herhangi bir dil sorunu olmadığından, bu derste bu konular ele alınmayacaktır.

Çok İşlemcili Mimariler

- 1950'lerin sonları – I/O (Input / Output) işlemleri bir genel amaçlı işlemci bir veya daha fazla özel amaçlı işlemciler
- 1960'ların başında - program düzeyinde eşzamanlılık için kullanılan birden fazla işlemci
- 1960'ların ortası - komut düzeyinde eşzamanlılık (instruction-level concurrency) için kullanılan birden çok kısmi işlemci
- Tek Komutlu Çoklu Veri (SIMD) makineleri
- Çok Komutlu Çoklu Veri (MIMD) makineleri
- Bu bölümün birincil odak noktası, paylaşılan bellek MIMD makineleridir (çok işlemciler - multiprocessors)

Eşzamanlılık Kategorileri

- Eşzamanlılık Kategorileri:
  - Fiziksel eşzamanlılık (Physical concurrency) - Birden çok bağımsız işlemci (birden çok (multiple) thread)
  - Mantıksal eşzamanlılık (Logical concurrency) - Fiziksel eşzamanlılığın görünümü, zaman paylaşımlı (time-sharing) bir işlemci tarafından sunulur (yazılım, birden fazla thread varmış gibi tasarlanabilir)
- Coroutines (yarı eşzamanlılık - quasi-concurrency) tek bir thread sahiptir
- Bir programdaki bir thread, kontrol program boyunca akarken ulaşılan program noktaları dizisidir.

Eşzamanlılık Kullanımına Yönelik Motivasyonlar

- Fiziksel eşzamanlılık yapabilen çok işlemcili bilgisayarlar artık yaygın olarak kullanılmaktadır
- Bir makinenin yalnızca bir işlemcisi olsa bile, eşzamanlı yürütmeyi kullanmak için yazılan bir program, eşzamanlı olmayan yürütme için yazılan aynı programdan daha hızlı olabilir.
- Çok faydalı olabilecek farklı bir yazılım tasarlama yolu içerir - birçok gerçek dünya durumu eşzamanlılığı içerir
- Birçok program uygulaması artık yerel olarak veya bir ağ üzerinden birden fazla makineye yayılmıştır

Alt Program Düzeyinde Eşzamanlılık (Subprogram-Level Concurrency) Giriş

- Bir görev (task) veya işlem(process) veya thread (iş parçacığı), diğer program birimleriyle eşzamanlı olarak yürütülebilen bir program birimidir.
- Görevler sıradan alt programlardan şu şekilde farklılık gösterir:
  - Bir görev örtük olarak başlatılabilir
  - Bir program birimi bir görevi yürütmeye başladığında, bu görevin askıya alınması gerekmez
  - Bir görevin yürütülmesi tamamlandığında, kontrol arayana (caller) geri dönmeyebilir
- Görevler genellikle birlikte çalışır

İki Genel Görev Kategorisi

- Ağır (Heavyweight) görevler kendi adres alanlarında yürütülür
- Hafif (Lightweight) görevlerin tümü aynı adres alanında çalışır - daha verimli
- Programdaki başka herhangi bir görevin yürütülmesini herhangi bir şekilde etkilemeyen veya etkilemeyen görev ayrıktır (disjoint).

Görev Senkronizasyonu (Task Synchronization)

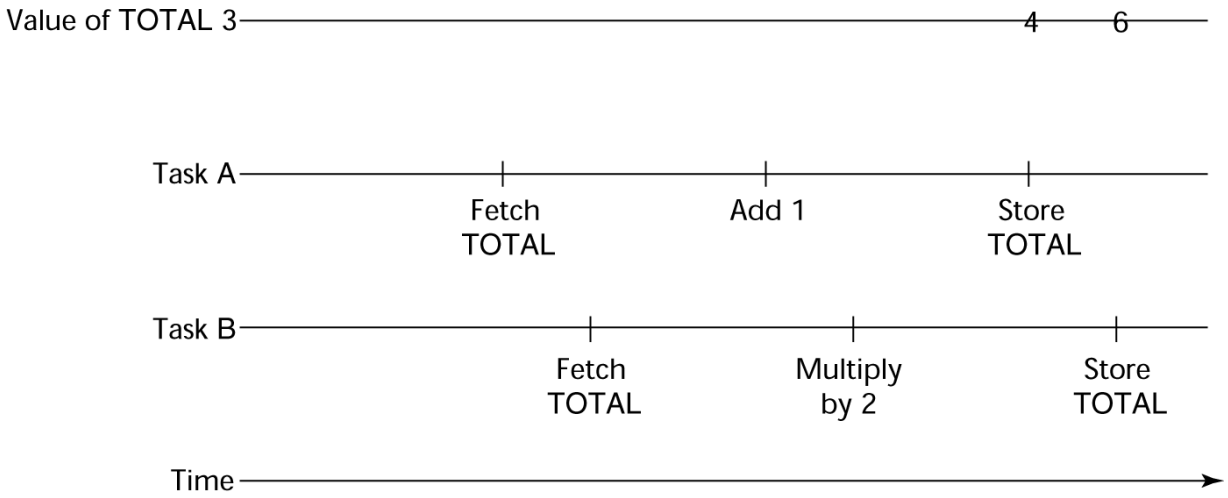
- Görevlerin yürütüldüğü sırayı kontrol eden bir mekanizma
- İki tür senkronizasyon
  - İşbirliği senkronizasyonu (Cooperation synchronization)
  - Rekabet senkronizasyonu (Competition synchronization)
- Görev iletişimi (Task communication), senkronizasyon için gereklidir ve aşağıdakiler tarafından sağlanır:
  - Paylaşılan yerel olmayan değişkenler
  - Parametreler
  - Mesaj Geçişi

Senkronizasyon türleri

- İşbirliği (Cooperation): A görevi, A görevinin yürütülmesine devam edebilmesi için B görevinin belirli bir faaliyeti tamamlamasını beklemelidir, örneğin üretici-tüketici (producer-consumer) problemi.
- Rekabet (Competition): İki veya daha fazla görev, aynı anda kullanılamayan bazı kaynakları kullanmalıdır, örneğin paylaşılan bir sayaç (shared counter)
  - Rekabet genellikle birbirini dışlayan erişimle (mutually exclusive Access) sağlanır (yaklaşımlar daha sonra tartışılacaktır)

Rekabet Senkronizasyonu İhtiyacı

Task A: `TOTAL = TOTAL + 1`
Task B: `TOTAL = 2 * TOTAL`



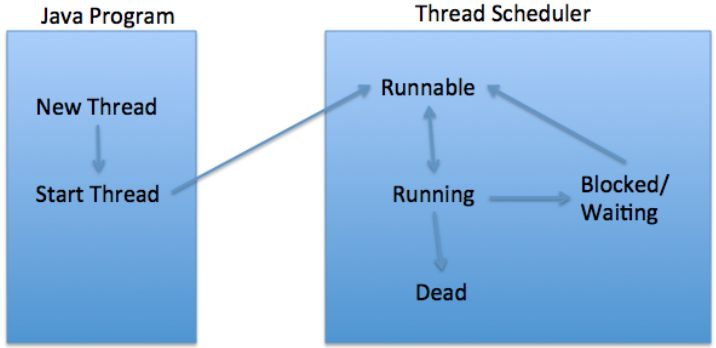
Sıraya bağlı olarak dört farklı sonuç olabilir

Zamanlayıcı (Scheduler)

- Senkronizasyonun sağlanması, görevin yürütülmesini geciktirmek için bir mekanizma gerektirir
- Görev yürütme kontrolü, görev yürütmeyi mevcut işlemcilerle eşleştiren zamanlayıcı (Scheduler) adı verilen bir program tarafından sağlanır.

Görev Yürütme Durumları

- Yeni (New) - oluşturulmuş ancak henüz başlamamış
- Hazır (Ready) - çalıştırmaya hazır ancak şu anda çalışmıyor (kullanılabilir işlemci yok)
- Çalışıyor (Running)
- Bloklanmış (Blocked) - çalışıyor, ancak şimdi devam edemiyor (genellikle bazı olayların gerçekleşmesini bekler)
- Öldü (Dead) - artık hiçbir şekilde aktif değil



Canlılık ve Kilitlenme (Liveness and Deadlock)

- Canlılık (Liveness), bir program biriminin sahip olabileceği veya olmayabileceği bir özelliktir
  - Sıralı kodda (sequential code), birimin sonunda çalışmasını tamamlayacağı anlamına gelir.
- Eşzamanlı bir ortamda, bir görev canlılığını kolayca kaybedebilir
- Eşzamanlı bir ortamdaki tüm görevler canlılığını kaybederse buna kilitlenme (deadlock) denir.

Eşzamanlılık İçin Tasarım Sorunları

- Rekabet ve işbirliği senkronizasyonu *
- Görev zamanlamasını kontrol etme
- Bir uygulama görev zamanlamasını nasıl etkileyebilir?
- Görevler yürütmeyi nasıl ve ne zaman başlatır ve bitirir?
- Görevler nasıl ve ne zaman oluşturulur?
  - * En önemli konu

Senkronizasyon Sağlama Yöntemleri

- Semaforlar (Semaphores)
- Monitors
- Mesaj Geçişi (Message Passing)

Semaforlar (Semaphores)

- Dijkstra - 1965
- Bir semafor, bir sayaç ve görev tanımlayıcılarını depolamak için bir kuyruktan (queue) oluşan bir veri yapısıdır
  - Görev tanımlayıcı (task descriptor), görevin yürütme durumuyla ilgili tüm ilgili bilgileri depolayan bir veri yapısıdır
- Semaforlar, paylaşılan veri (shared data) yapılarına erişen kod üzerinde korumalar uygulamak için kullanılabilir
- Semaforların yalnızca iki işlemi vardır, bekle ve bırak (wait and release) (orijinal olarak Dijkstra tarafından P ve V olarak adlandırılır)
- Semaforlar hem rekabet hem de işbirliği senkronizasyonu sağlamak için kullanılabilir

Semaforlarla İşbirliği Senkronizasyonu (Cooperation Synchronization with Semaphores)

- Örnek: Paylaşılan bir arabellek (shared buffer)
- Arabellek, arabelleğe erişmenin tek yolu olarak DEPOSIT ve FETCH işlemleriyle bir ADT (abstract data type) olarak uygulanır.
- İşbirliği için iki semafor kullanın: emptyspots ve fullspots
- Semafor sayaçları, arabellekte empty (boş) spots and full (dolu) spots sayısını saklamak için kullanılır.

Semaforlarla İşbirliği Senkronizasyonu...

- DEPOSIT, arabellekte yer olup olmadığını görmek için önce emptyspot'ları kontrol etmelidir
- Yer varsa emptyspots sayacı azaltılır ve değer girilir.
- Oda yoksa arayan (caller) emptyspots kuyruğunda saklanır
- DEPOSIT bittiğinde, fullspots sayacını artırması gerekir

Semaforlarla İşbirliği Senkronizasyonu...

- FETCH, bir değer olup olmadığını görmek için önce fullspots kontrol etmelidir
  - Fullspot varsa, fullspots sayacı azaltılır ve değer kaldırılır.
  - Arabellekte değer yoksa, arayan (caller) fullspots kuyruğuna yerleştirilmelidir.
  - FETCH bittiğinde, emptyspots sayacını artırır
- Semaforlar üzerindeki FETCH ve DEPOSIT işlemleri, wait and release adlı iki semafor işlemi ile gerçekleştirilir.

Semofarlar
Wait and Release İşlemleri

```
wait (aSemaphore)
if Semaphore sayacı > 0 then
    Semaphore'un sayacını azalt
else
    arayanı (caller) Semaphore'un kuyruğuna koyun
    kontrolü hazır (ready) bir göreve aktarmayı denemek
      - görev hazır kuyruğu boşsa, kilitlenme meydana gelir
end

release (Semaphore)
if Semaphore'un sırası boş then
    aSemaphore'un sayacını artır
else
    çağıran görevi göreve hazır kuyruğuna koy
    denetimi Semaphore'un kuyruğundan bir göreve aktar
end
```

Üretici ve Tüketici Görevleri (Producer and Consumer Tasks)

```
semaphore fullspots, emptyspots;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
      loop
      -- produce VALUE --
      wait (emptyspots); {wait for space}
      DEPOSIT(VALUE);
      release(fullspots); {increase filled}
      end loop;
end producer;
task consumer;
      loop
      wait (fullspots);{wait till not empty}}
      FETCH(VALUE);
      release(emptyspots); {increase empty}
      -- consume VALUE --
      end loop;
end consumer;
```

Semaforlarla Rekabet Senkronizasyonu (Competition Synchronization with Semaphores)

- Erişimi kontrol etmek için üçüncü bir semafor, erişim (access) adı kullanılır (rekabet senkronizasyonu)
  - Erişim sayacı yalnızca 0 ve 1 değerlerine sahip olacaktır.
  - Böyle bir semafora ikili semafor (binary semaphore) denir
- Wait ve release atomik olması gerektiğini unutmayın!

Semaforlar için Üretici Kodu

```
semaphore access, fullspots, emptyspots;
access.count = 0;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
    loop
    -- produce VALUE --
    wait(emptyspots); {wait for space}
    wait(access);     {wait for access)
    DEPOSIT(VALUE);
    release(access); {relinquish access}
    release(fullspots); {increase filled}
    end loop;
end producer;
```

Semaforlar için Tüketici Kodu

```
task consumer;
   loop
   wait(fullspots);{wait till not empty}
   wait(access);   {wait for access}
   FETCH(VALUE);
   release(access); {relinquish access}
   release(emptyspots); {increase empty}
   -- consume VALUE --
   end loop;
end consumer;
```

Semaphore'ların Değerlendirmesi

- Semaforların yanlış kullanımı, işbirliği (cooperation) senkronizasyonunda başarısızlıklara neden olabilir, örneğin, fullspots beklemesi dışarıda bırakılırsa arabellek taşar (overflow).
- Semaforların kötüye kullanılması rekabet (competition) senkronizasyonunda hatalara neden olabilir, örneğin, erişimin serbest bırakılması durumunda program kilitlenecektir.

Monitors

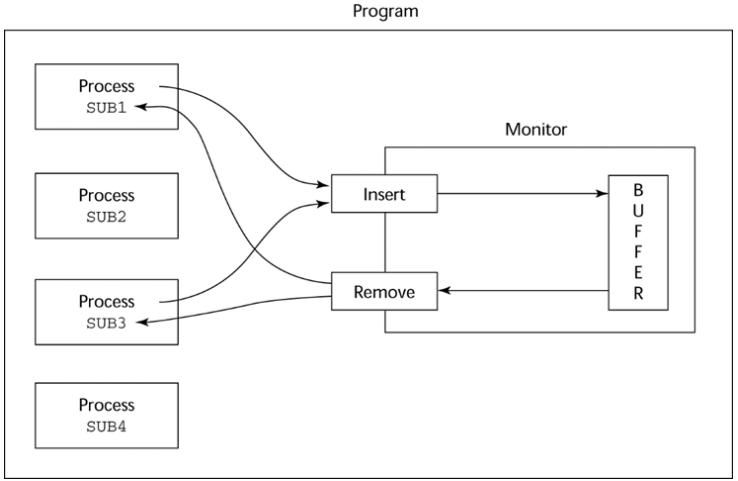
- Ada, Java, C#
- Fikir: erişimi kısıtlamak için paylaşılan verileri ve işlemlerini kapsülleyin
- Bir monitör, paylaşılan veriler için soyut bir veri (abstract data type) türüdür

Rekabet Senkronizasyonu (Cooperation Synchronization)

- Paylaşılan veriler monitörde yerleşiktir (istemci birimleri yerine)
- Tüm erişim yerleşik monitörde
  - Monitor uygulaması, bir seferde yalnızca bir erişime izin vererek senkronize erişimi garanti eder
  - Arama sırasında monitör meşgulse, prosedürleri izlemek için yapılan aramalar dolaylı olarak sıraya alınır

Rekabet Senkronizasyonu...

- Process'ler arası işbirliği hala bir programlama görevidir
  - Programcı, paylaşılan bir arabellekte underflow or overflow olmadığını garanti etmelidir



Monitors
Değerlendirme

- Rekabet senkronizasyonu sağlamanın semaforlardan daha iyi bir yolu
- Semaforlar monitörleri uygulamak için kullanılabilir
- Monitörler semaforları uygulamak için kullanılabilir
- İşbirliği senkronizasyonu desteği semaforlarla çok benzerdir, bu yüzden aynı problemleri vardır

Mesaj Geçişi (Message Passing)

- Mesaj geçişi, eşzamanlılık için genel bir modeldir
  - Hem semaforları hem de monitörleri modelleyebilir
  - Sadece rekabet senkronizasyonu için değil
- Ana fikir: görev iletişimi bir doktoru görmek gibidir - çoğu zaman sizi bekler veya siz onu beklersiniz, ancak ikiniz de hazır olduğunuzda, bir araya gelirsiniz veya buluşursunuz

Mesaj Geçiş Randevusu

- Mesaj geçişiyle eşzamanlı görevleri desteklemek için bir dilin şunlara ihtiyacı vardır:
  - Bir görevin mesajları ne zaman kabul etmeye istekli olduğunu belirtmesine izin veren bir mekanizma
  - Mesajının kabul edilmesini bekleyenleri hatırlamanın bir yolu ve sonraki mesajı seçmenin "adil" bir yolu
- Gönderen (sender) görevin mesajı bir alıcı görevi (receiver task) tarafından kabul edildiğinde, gerçek mesaj iletimi bir buluşma olarak adlandırılır.

Eşzamanlılık
Ada

- Ada 83 Mesaj Geçiş Modeli
  - Ada görevlerinin özellikleri ve paketler gibi gövde bölümleri vardır; spesifikasyon, giriş noktalarının koleksiyonu olan bir arayüze sahiptir:

```
task Task_Example is
   entry ENTRY_1 (Item : in Integer);
end Task_Example;
```

Görev Gövdesi (Task Body)

- Gövde (Body) görevi, bir buluşma gerçekleştiğinde gerçekleşen eylemi tanımlar.
- Mesaj gönderen görev, mesajın kabul edilmesini beklerken ve buluşma sırasında askıya alınır.
- Spesifikasyondaki giriş noktaları (entry points), gövdedeki accept clauses ile açıklanmıştır.

```
accept entry_name (formal parameters) do
   ...
end entry_name;
```

Görev Gövdesi (Task Body)

- Gövde (Body) görevi, bir buluşma gerçekleştiğinde gerçekleşen eylemi tanımlar.
- Mesaj gönderen görev, mesajın kabul edilmesini beklerken ve buluşma sırasında askıya alınır.
- Spesifikasyondaki giriş noktaları (entry points), gövdedeki accept clauses ile açıklanmıştır.

```
accept entry_name (formal parameters) do
   ...
end entry_name;
```

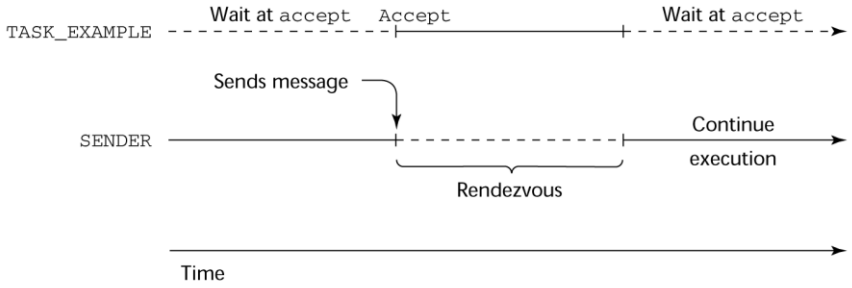
Görev Gövdesi
Örnek

```
task body Task_Example is
   begin
   loop
      accept Entry_1 (Item: in Float) do
      ...
      end Entry_1;
   end loop;
end Task_Example;
```

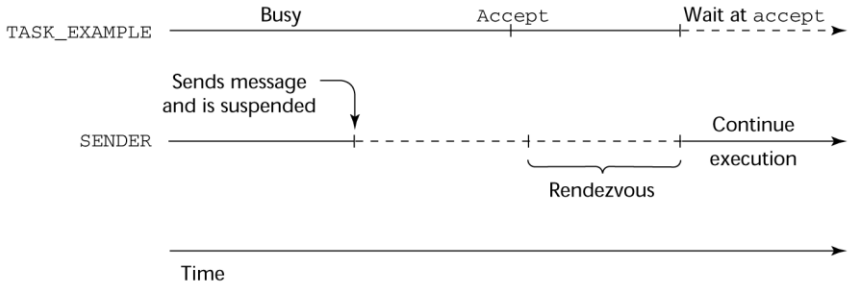
Mesaj Geçişi Anlambilimi (Message Passing Semantics) Ada

- Görev (task), accept clause'nin en üstüne kadar yürütülür ve bir mesaj bekler
- accept clause'nin yürütülmesi sırasında gönderen (sender) askıya alınır
- accept parameter'leri, bilgileri herhangi bir yönde veya her iki yönde iletebilir
- Her `accept` clause, bekleyen mesajları saklamak için ilişkili bir kuyruğa sahiptir

Buluşma Zaman Çizgileri



(a) TASK_EXAMPLE waits for SENDER

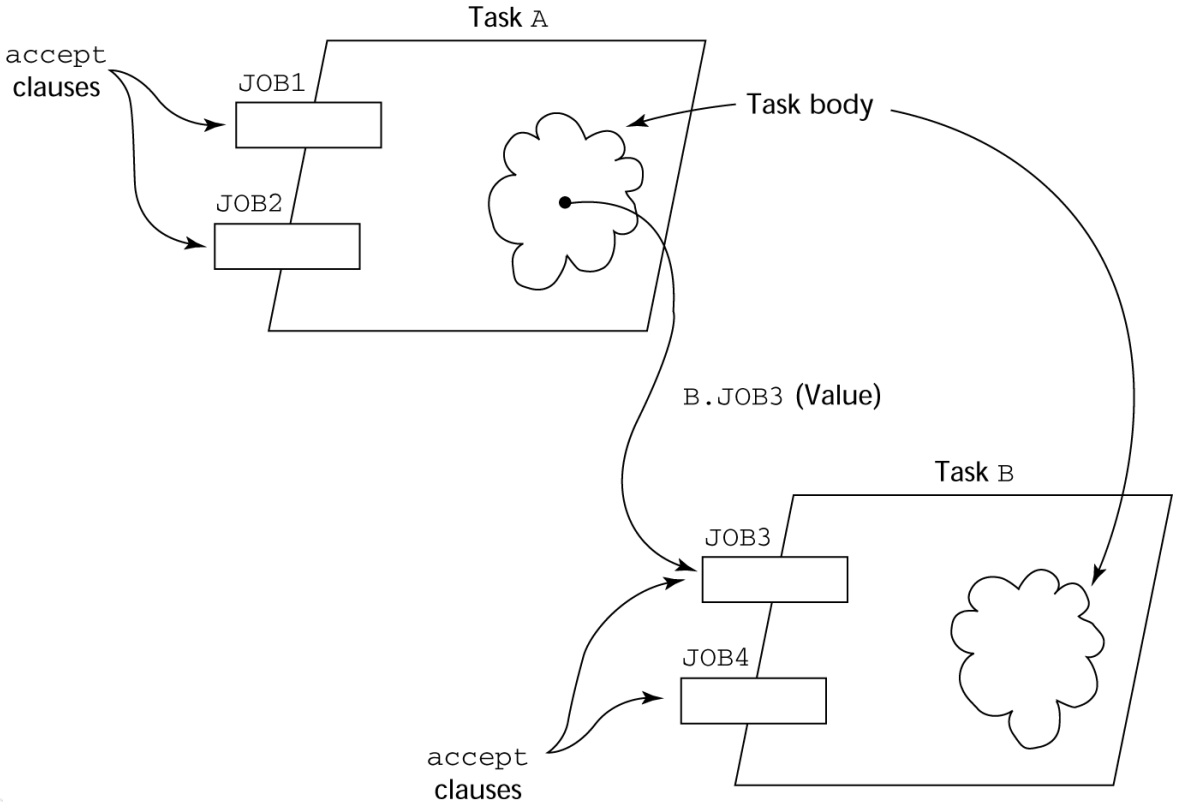


(b) SENDER waits for TASK_EXAMPLE

Mesaj Geçişi: Server/Actor Görevleri

- Bir görev, accept clauses sahiptir ancak başka hiçbir koda server task (sunucu görevi) (bir önceki örnek bir server task'tir.)
- accept clauses olmayan bir göreve actor task denir
  - Bir actor task diğer görevlere mesaj gönderebilir
  - Not: Gönderen (sender), alıcının (receiver) entry adını bilmelidir, ancak bunun tersi olamaz (asimetrik - asymmetric)

Bir Randevunun Grafiksel Gösterimi



Çoklu Giriş Noktaları (Multiple Entry Points)

- Görevlerin birden fazla entry noktası olabilir
  - Spesifikasyon görevinin her biri için bir entry clause vardır.
  - Görev gövdesi, bir döngü içinde olan bir `select` clause yerleştirilmiş her bir `entry` clause için bir `accept` clause sahiptir.

Birden Çok Girişli Görev (Task with Multiple Entries)

```
task body Teller is
  loop
    select
      accept Drive_Up(formal params) do
      ...
      end Drive_Up;
      ...
    or
      accept Walk_Up(formal params) do
      ...
      end Walk_Up;
      ...
    end select;
  end loop;
end Teller;
```

Çoklu accept Clauses Görevlerin Anlambilimi (Semantics of Tasks with Multiple `accept` Clauses)

- Tam olarak bir **`entry`** kuyruğu boş değilse, ondan bir mesaj seçin
- Birden fazla **`entry`** kuyruğu boş değilse, belirsiz olmayan bir şekilde bir mesajın kabul edileceği birini seçin
- Hepsi boşsa, bekle (wait)
- Yapı (consruct) genellikle seçici bekleme (selective wait) olarak adlandırılır
- Genişletilmiş (Extended) **`accept`** clause - clause ardından, ancak sonraki clause'dan önce kod
  - Arayanla (caller) eşzamanlı olarak yürütülür

Mesaj Geçişi ile İşbirliği Senkronizasyonu

- Guarded accept clauses sağlandı

```
when not Full(Buffer) =>
  accept Deposit (New_Value) do
    ...
  end
```

- Bir when clause açık veya kapalı olduğu bir accept clause
  - Koruması (guard) doğru olan bir clause açık (open) denir
  - Koruması (guard) yanlış olan bir clause kapalı (close) denir
  - Korumasız (without guard) bir clause her zaman açıktır

Guarded `accept` Clauses ile select Anlambilimi

- select ilk önce tüm clause'ların korumaları (guard) kontrol eder
- Tam olarak bir tane açıksa, kuyruğu mesajlar için kontrol edilir
- Birden fazla açıksa, mesajları kontrol etmek için aralarından belirleyici olmayan bir şekilde bir sıra seçin
- Hepsi kapalıysa, bu bir çalışma zamanı hatasıdır (runtime error)
- Bir select clause, hatayı önlemek için bir else clause içerebilir
- else clause tamamlandığında döngü tekrar eder

Mesaj Geçişi ile Rekabet Senkronizasyonu

- Paylaşılan verilere (shared data) birbirini dışlayan erişimi (mutually exclusive access) modelleme
- Örnek - paylaşılan bir arabellek (shared buffer)
- Bir görevde arabelleği (buffer) ve işlemlerini kapsülleyin
- Rekabet senkronizasyonu, accept clauses anlambiliminde örtüktür
  - Herhangi bir zamanda bir görevdeki yalnızca bir accept clause etkin olabilir

Kısmi Paylaşımlı Arabellek Kodu (Partial Shared Buffer Code)

```
task body Buf_Task is
  Bufsize : constant Integer := 100;
  Buf : array (1..Bufsize) of Integer;
  Filled : Integer range 0..Bufsize := 0;
  Next_In, Next_Out : Integer range 1..Bufsize := 1;
  begin
    loop
      select
        when Filled < Bufsize =>
          accept Deposit(Item : in Integer) do
            Buf(Next_In) := Item;
          end Deposit;
          Next_In := (Next_In mod Bufsize) + 1;
          Filled := Filled + 1;
        or
        ...
    end loop;
  end Buf_Task;
```

Bir Tüketici Görevi (Consumer Task)

```
task Consumer;
task body Consumer is
   Stored_Value : Integer;
  begin
    loop
      Buf_Task.Fetch(Stored_Value);
      -- consume Stored_Value -
    end loop;
  end Consumer;
```

Eşzamanlılık
Ada 95

- Ada 95, eşzamanlılık için Ada 83 özelliklerinin yanı sıra iki yeni özellik içerir
  - Korumalı nesneler (Protected objects): Paylaşılan bir veri yapısına erişimin buluşma olmadan yapılmasına izin vermek için paylaşılan verileri (shared data) uygulamanın daha verimli bir yoludur
  - Eşzamansız iletişim (Asynchronous communication)

Ada 95
Korumalı Nesneler (Protected Objects)

- Korumalı bir nesne (protected object), soyut bir veri türüne benzer
- Korunan bir nesneye erişim, bir görevde olduğu gibi girişlere aktarılan mesajlar veya korumalı alt programlar yoluyla yapılır.
- Korumalı bir prosedür (protected procedure), korumalı nesnelere karşılıklı olarak özel okuma-yazma erişimi sağlar
- Korumalı bir fonksiyon (protected function), korumalı nesnelere eşzamanlı salt okunur erişim sağlar

Ada Değerlendirmesi

- Eşzamanlılığın mesaj geçişi modeli güçlü ve geneldir
- Korumalı nesneler, senkronize edilmiş paylaşılan veriler (synchronized shared data) sağlamanın daha iyi bir yoludur
- Dağıtılmış işlemcilerin (distributed processors) yokluğunda, monitörler ve mesaj geçişi görevler arasında seçim yapmak tercih meselesidir.
- Dağıtılmış sistemler (distributed systems) için, mesaj geçişi eşzamanlılık için daha iyi bir modeldir

Java Threads

- Java'daki eşzamanlı birimler, run adlı metotlardır.
  - Bir run metodu kodu, bu tür diğer metotlarla eşzamanlı yürütme halinde olabilir
  - run metotlarının yürütüldüğü sürece thread adı verilir

```
class myThread extends Thread
   public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

Java Thread Çalışmasını Kontrol Etme

- Thread sınıfı, thread'lerin yürütülmesini kontrol etmek için çeşitli metotlara sahiptir.
  - yield, çalışan thread'ten işlemciyi gönüllü olarak teslim etme talebidir
  - sleep metodu, thread engellemek için metodun çağırıcısı (caller) tarafından kullanılabilir.
  - join metodu, başka bir thread'in run metodunu yürütmesini tamamlayana kadar bir metodu yürütmesini geciktirmeye zorlamak için kullanılır.

Java Thread Öncelikleri (Priorities)

- Bir thread'in varsayılan önceliği, onu oluşturan thread dizisiyle aynıdır
- main bir thread'i oluşturursa, varsayılan önceliği NORM_PRIORITY'dir.
- Thread'leri, diğer iki öncelik sabitini, MAX_PRIORITY ve MIN_PRIORITY tanımlanır
- Bir thread'in önceliği setPriority metodu ile değiştirilebilir.

Semaphores (Semaforlar)
Java

```java
// java program to demonstrate
// use of semaphores Locks
import java.util.concurrent.*;

//A shared resource/class.
class Shared
{
    static int count = 0;
}

class MyThread extends Thread
{
    Semaphore sem;
    String threadName;
    public MyThread(Semaphore sem, String threadName)
    {
        super(threadName);
        this.sem = sem;
        this.threadName = threadName;
    }
```

Semaphore in Java - GeeksforGeeks

Semaphores (Semaforlar) Java

```
@Override
public void run() {

    // run by thread A
    if(this.getName().equals("A"))
    {
        System.out.println("Starting " + threadName);
        try
        {
            // First, get a permit.
            System.out.println(threadName + " is waiting for a permit.");

            // acquiring the lock
            sem.acquire();

            System.out.println(threadName + " gets a permit.");

            // Now, accessing the shared resource.
            // other waiting threads will wait, until this
            // thread release the lock
            for(int i=0; i < 5; i++)
            {
                Shared.count++;
                System.out.println(threadName + ": " + Shared.count);

                // Now, allowing a context switch -- if possible.
                // for thread B to execute
                Thread.sleep(10);
            }
        } catch (InterruptedException exc) {
                System.out.println(exc);
            }

            // Release the permit.
            System.out.println(threadName + " releases the permit.");
            sem.release();
    }

    // run by thread B
    else
    {
        System.out.println("Starting " + threadName);
        try
        {
            // First, get a permit.
            System.out.println(threadName + " is waiting for a permit.");

            // acquiring the lock
            sem.acquire();

            System.out.println(threadName + " gets a permit.");

            // Now, accessing the shared resource.
            // other waiting threads will wait, until this
            // thread release the lock
            for(int i=0; i < 5; i++)
            {
                Shared.count--;
                System.out.println(threadName + ": " + Shared.count);

                // Now, allowing a context switch -- if possible.
                // for thread A to execute
                Thread.sleep(10);
            }
        } catch (InterruptedException exc) {
                System.out.println(exc);
            }
            // Release the permit.
            System.out.println(threadName + " releases the permit.");
            sem.release();
    }
  }
}
```

Semaphores (Semaforlar)
Java

```java
// Driver class
public class SemaphoreDemo
{
    public static void main(String args[]) throws InterruptedException
    {
        // creating a Semaphore object
        // with number of permits 1
        Semaphore sem = new Semaphore(1);

        // creating two threads with name A and B
        // Note that thread A will increment the count
        // and thread B will decrement the count
        MyThread mt1 = new MyThread(sem, "A");
        MyThread mt2 = new MyThread(sem, "B");

        // stating threads A and B
        mt1.start();
        mt2.start();

        // waiting for threads A and B
        mt1.join();
        mt2.join();

        // count will always remain 0 after
        // both threads will complete their execution
        System.out.println("count: " + Shared.count);
    }
}
```

Java Threads ile Rekabet Senkronizasyonu (Competition Synchronization)

- synchronized modifier içeren bir metod, yürütme sırasında nesne üzerinde başka herhangi bir metodun çalışmasına izin vermez.

```
…
public synchronized void deposit( int i) {…}
public synchronized int fetch() {…}
…
```

- Yukarıdaki iki metot, birbirleriyle etkileşime girmelerini önleyecek şekilde senkronize edilmiştir.
- Bir metodun yalnızca bir kısmının müdahale olmadan çalıştırılması gerekiyorsa, bu metot aracılığıyla senkronize edilebilir.

```
synchronized statement
    synchronized (expression)
        statement
```

Java Threads ile İşbirliği Senkronizasyonu (Cooperation Synchronization)

- Java'da işbirliği senkronizasyonu wait, notify ve notifyAll metotlarıyla sağlanır.
- Tüm metotlar, Java'daki kök sınıf olan Object'te tanımlanır, bu nedenle tüm nesneler onları miras alır.
- wait metodu bir döngü içinde çağrılmalıdır
- notify metodu, bekleyen thread'e beklediği olayın gerçekleştiğini söylemek için çağrılır
- notifyAll metodu, nesnenin bekleme listesindeki tüm iş parçacıklarını uyandırır

C# Threads

- Genel olarak Java tabanlı ancak önemli farklılıklar var
- Temel Thread işlemleri
  - Herhangi bir metod kendi thread'i ile çalışabilir
  - Bir Thread nesnesi oluşturularak thread oluşturulur
  - Bir Thread oluşturmak, eşzamanlı yürütmeyi başlatmaz; Start metodu ile talep edilmelidir
  - Join ile başka bir thread'in bitmesini beklemek için bir thread yapılabilir
  - Sleep ile bir thread askıya alınabilir
  - Abort ile bir thread sonlandırılabilir

Thread'lerin Senkronizasyonu (Synchronizing Threads)

- C# thread senkronize etmenin üç yolu
  - Interlocked sınıfı
    - Senkronize edilmesi gereken tek işlem bir tamsayının artırılması veya azaltılması olduğunda kullanılır
  - Lock statement
    - Bir thread'teki kritik bir kod bölümünü işaretlemek için kullanılır

    lock (expression) {… }
  - Monitor sınıfı
    - Daha karmaşık senkronizasyon sağlamak için kullanılabilecek dört metot sağlar

C# Eşzamanlılık Değerlendirmesi

- Java thread üzerinde bir ilerleme, örneğin herhangi bir metot kendi thread’ini çalıştırabilir
- Thread sonlandırma Java'dakinden daha temizdir
- Senkronizasyon daha karmaşıktır

Statement-Level Concurrency (Statement Seviyesinde Eşzamanlılık)

- Amaç: Programcının, programı çok işlemcili mimariyle eşleştirebileceği yollar hakkında derleyiciye bilgi vermek için kullanabileceği bir mekanizma sağlamak
- İşlemciler ve diğer işlemcilerin hafızaları arasındaki iletişimi en aza indirme

High-Performance Fortran

- Programcının, çok işlemcili bilgisayarlar için kodu optimize etmesine yardımcı olmak üzere derleyiciye bilgi sağlamasına olanak tanıyan bir uzantılar koleksiyonu
- İşlemci sayısını, verilerin bu işlemcilerin belleklerine dağıtımını ve verilerin hizalanmasını belirtme
  - İşlemci Sayısı
    `!HPF$ PROCESSORS procs (n)`
  - Veri dağıtımı
    `!HPF$ DISTRIBUTE (`*kind*`) ONTO procs ::`
    identifier_list
    - kind burada BLOCK (verileri işlemcilere bloklar halinde dağıtma) veya CYCLIC (verileri işlemcilere bir seferde bir öğe olarak dağıtma) olabilir
  - Bir dizinin dağılımını diğeriyle ilişkilendirme
    `ALIGN` *array1_element* `WITH` *array2_element*

Statement-Level Concurrency Örneği

```
 REAL list_1(1000), list_2(1000)
     INTEGER list_3(500), list_4(501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs ::
                          list_1, list_2
!HPF$ ALIGN list_1(index) WITH
              list_4 (index+1)
      ...
      list_1 (index) = list_2(index)
      list_3(index) = list_4(index+1)
```

Statement-Level Concurrency

- FORALL ifadesi, eşzamanlı olarak yürütülebilecek ifadelerin bir listesini belirtmek için kullanılır

```
FORALL (index = 1:1000)
    list_1(index) = list_2(index)
```

- Herhangi bir atama gerçekleşmeden önce atamaların 1.000 RHS'sinin tamamının değerlendirilebileceğini belirtir

Diğer Dillerde Eşzamanlılık

- Python Thread'ler
- Javascript Asenkron – Senkron, Async, concurrency, parallelism
- Fonksiyonel Dillerde Eşzamanlılık