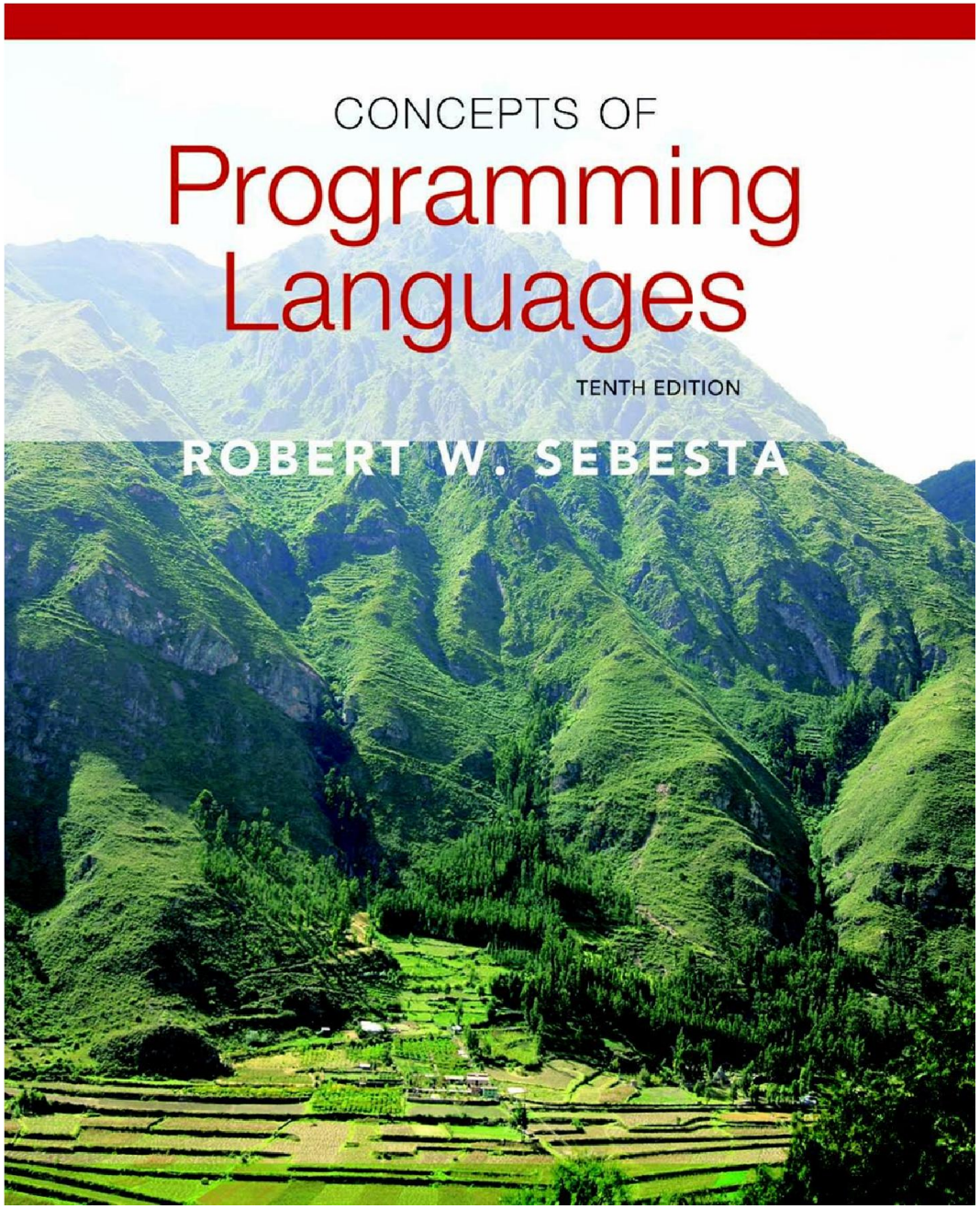
Bölüm 13

Eşzamanlılık



13ç Bölüm konuları

- Giriş
- Altprogram seviyesinde eşzamanlılık
- Semaforlar (semaphores)
- Monitörler (monitors)
- Mesaj geçme (message passing)
- Ada dilinde eşzamanlılık desteği

Giriş

- Eşzamanlılık dört seviyede olabilir:
  - Makine komut düzeyi
  - Yüksek seviyeli dil komutu düzeyi
  - Birim düzeyi
  - Program düzeyi
- Dil açısından, makine komut düzeyi ve program düzeyinde yapabileceğimiz birşey olmadığından, burada bu konular işlenmiyor
- Bir programda *kontrol ipliği,* program çalışırken erişilen program noktaları dizisidir

Çok işlemcili mimariler

- Geç 1950ler – bir genel amaçlı işlemci, bir veya daha fazla girdi/çıktı için özel amaçlı işlemci
- Erken 1960lar – program seviyesinde eşszamanlılık için birden çok tam teşekküllü işlemci
- Orta 1960lar – komut düzeyi eşzamanlılığı için birden çok kısmi işlemciler
- Tekli-komut çoklu-veri (Single-Instruction Multiple-Data (SIMD)) makineleri
- Çoklu-komut çoklu-veri makineleri (Multiple-Instruction Multiple-Data (MIMD))
- Burada işlenen konu: ortak hafızalı çoklu-komut çoklu-veri makineleri (shared memory MIMD machines (multiprocessors))

Eşzamanlılık kategorileri (sınıfları)

- Eşzamanlılık sınıfları:
  - *Fiziki eşzamanlılık (physical concurrency)*– Birden çok bağımsız işlemci (birden çok kontrol ipliği)
  - *Mantıki eşzamanlılık (logical concurrency)* – Bir işlemcinin zamanını paylaştırarak eşzamanlılık görüntüsü verilmesi. (Yazılım sanki gerçek çoklu kontrol ipliği varmış gibi tasarlanabilir)
- Korutinler *(yalancı eşzamanlılık):* tek kontrol ipliği vardır

Eşzamanlılık için motivasyon

- Fiziksel eşzamanlılık yeteneğinde olan çok işlemcili bilgisayarlar şu an yaygın kullanımda
- Bir makinede tek işlemci olsa bile, programı eşzamanlı çalışacak şekilde tasarlamak daha hızlı çalışmasını sağlayabilir
- Yazılımın daha değişik bir şekilde tasarlanmasını gerektirir. Gerçek hayatta birçok problemin doğası eşzamalı olduğundan, eşzamanlılığı içeren tasarım, problemle daha uyumlu olur
- Birçok uygulamalar hali hazırda birden çok makine üzerine yayılmıştır (ayni meknda veya network üzerinden)

Altprogram düzeyi eşzamanlılığa giriş

- Diğer program birimleri ile eşzamanlı çalışabilen birime *görev (task), süreç (process)* veya *iplik (thread)* denir
- Görevler, normal altprogramlardan şu yönlerden ayrılır
  - Görev otomatik olarak başlatılabilir
  - Görevi başlatan birimin kendisinin askıya alınma zorunluluğu yoktur
  - Görev işini bitirdiği zaman kontrolün onu başlatana dönme zorunluluğu yoktur
- Görevler genellikle bir amaç için birlikte çalışırlar

Görevlerin iki genel kategorisi

- *Ağırsiklet (heavyweight) görevler* kendi adres uzaylarında (address space) çalışırlar
- *Hafifsiklet (lightweight) görevlerin* hepsi ayni adres uzayında çalışırlar – daha verimli
- Bir görev, eğer başka bir görevle iletişim içinde değilse veya başka bir görevin çalışmasını etkilemiyorsa bu göreve *ayrık (disjoint)* denir

Görev senkronizasyonu

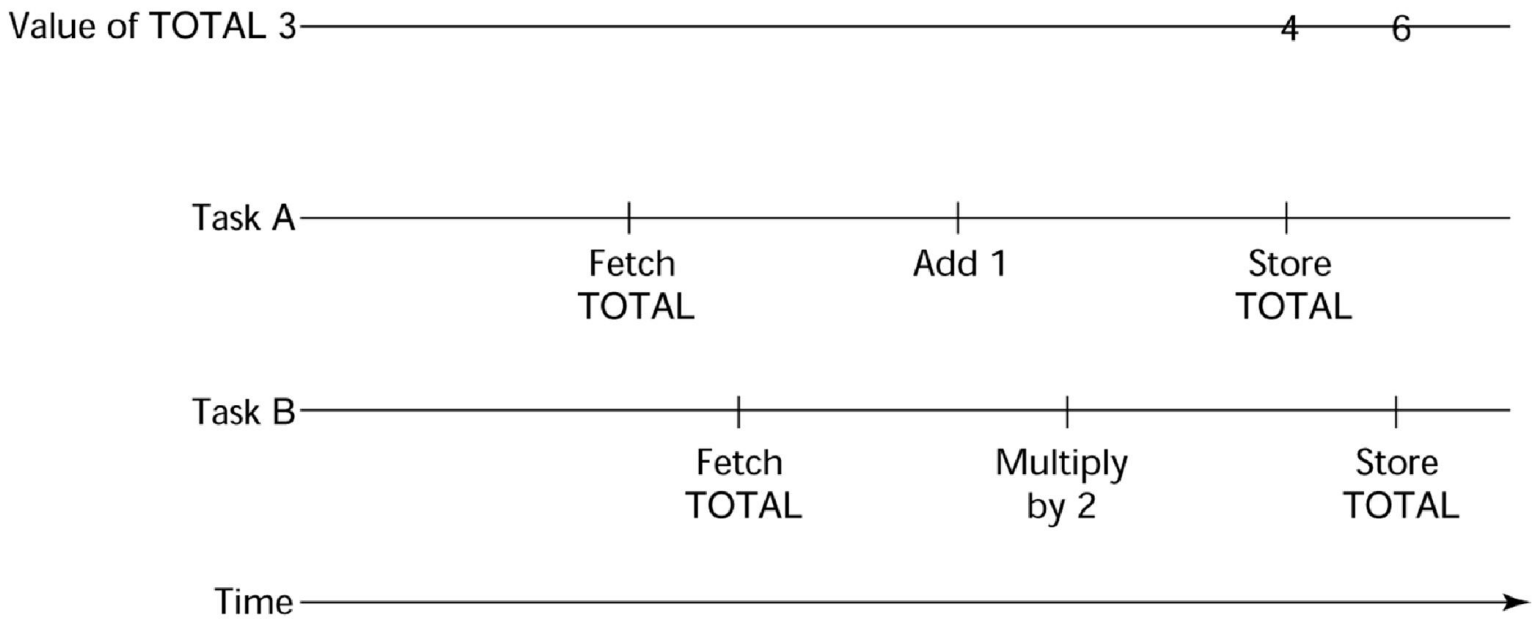
- Görevlerin çalışma sırasını kontrol eden mekanizma
- İki türlü senkronizasyon
  - *İşbirliği (cooperation)* senkronizasyonu
  - *Rekabet (competition)* senkronizasyonu
- Senkronizasyon içim görevlerin haberleşmesi gerekir. Haberleşme şu yollarla sağlanır:
  - Paylaşılan yerel olmayan değişkenler
  - Parametreler
  - Mesaj geçme

Senkronizasyon türleri

- *İşbirliği:* A görevi çalışmaya devam edebilmek için B görevinin belli bir aktiviteyi tamamlamasını beklemeli. Ör: üretici–tüketici problemi
- *Rekabet*: İki veya daha çok görevin eş zamanlı kullanımı mümkün olmayan bir kaynağı kullanmak istemesi ör: ortak sayaç (counter)
  - Rekabet genellikle "karşılıklı dışlayan erişim" (mutually exclusive access) ile mümkün olur

Neden rekabet senkronizasyonuna ihtiyaç var?

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



- Sıraya bağlı olarak, 4 cevap mümkün

Zamanlayıcı (scheduler)

- Senkronizasyon sağlanması için görev çalışmasının geciktirilme mekanizması gerekir
- Görevlerin çalışma kontrolü *zamanlayıcı (scheduler)* denen bir program tarafından yapılır. Zamanlayıcı, görevleri boşta olan işlemcilere eşleştirir.

Görev çalışma durumları

- *Yeni (New)* – yaratıldı ama başlatılmadı
- *Hazır (Ready)* – çalışmaya hazır ama halen çalışmıyor (boşta işlemci yok)
- *Çalışır vaziyette (Running)*
- *Bloke edilmiş (Blocked)* – çalışmakta iken durdu ve devam edemiyor (genellikle bir olayın gerçekleşmesini bekliyor)
- *Ölü (Dead)* – hiçbir şekilde aktif değil

Canlılık (liveness) ve çıkmaz (deadlock)

- Sıralı kodda (sequential code), *canlılık* birimin çalışmasını eninde sonunda bitireceği anlamına gelir
- Parallel bir ortamda, bir görevin canlılığı kolaylıkla kaybolabilir.
- Parallel bir ortamda tüm görevler canlılığını kaybederse, buna *çıkmaz (deadlock)* denir.

Eşzamanlılık tasarım problemleri

- Rekabet ve işbirliği senkronizasyonunun nasıl sağlanacağı (en önemli konu)
- Görev zamanlamasının kontrolü
- Bir uygulama görev zamanlamasını nasıl etkileyebilir?
- Görevler, çalışmaya nasıl ve ne zaman başlarlar/bitirirler?
- Görevler nasıl ve ne zaman yaratılırlar?

Senkronizasyon sağlama yöntemleri

- Semaforlar (Semaphores)
- Monitörler (Monitors)
- Mesaj geçme (Message Passing)

Semaforlar

- Dijkstra – 1965
- *Semafor* bir sayaç ve görev tarifleri kuyruğunu içeren bir veri yapısıdır
  - Görev tarifi, görevin çalışma durumu ile ilgili tüm bilgileri saklayan bir veri yapısıdır
- Semaforların sadece iki işlemi var: *bekle (wait) ve bırak (release)* (original adları *P ile V*)
- Semaforlar hem rekabet, hem de işbirliği senkronizasyonu için kullanılabilirler.

Semaforlar ile işbirliği senkronizasyonu

- Örnek: paylaşılan tampon (buffer)
- Tampon `YUKLE (DEPOSIT)` ve `GETİR (FETCH)` diye operasyonu bulunan bir SVT olarak gerçeklendi.
- İşbirliği için iki tane semafor var: `emptyspots` ve `fullspots`
- Semafor sayaçları tamponda kaç tane boş ve kaç tane dolu yer olduğunu saklamak için kullanıldı

Semaforlar ile işbirliği senkronizasyonu...

- `YUKLE (DEPOSIT)` önce boş yer varmı diye `emptyspots` u kontrol etmesi gerekiyor
- Boş yer varsa, `emptyspots` un sayacı bir azaltılır ve bir değer yüklenir
- Yer yoksa, çağıran `emptyspots` un kuyruğuna eklenir
- `YÜKLE (DEPOSIT)` işini bitirdeğinde, `fullspots` un sayacını artırması gerekir

Semaforlar ile işbirliği senkronizasyonu...

- `GETİR (FETCH)` öncelikle değer olduğunu doğrulamak için `fullspots` u kontrol etmeli
  - Dolu bir yer varsa, değer alınır ve `fullspots` un sayacı bir azaltılır
  - Tamponda hiç değer yoksa, çağıran `fullspots` un kuyruguna konur
  - `GETİR (FETCH)` işini bitirdiğinde, `emptyspots` un sayacını bir artırır
- Semaforlar üzerindeki `GETİR (FETCH)` ve `YÜKLE (DEPOSIT)` işlemleri semaforların *bekle (wait) ve serbest bırak (release)* işlemleri sayesinde başarılır.

Semaforlar: Bekle (Wait) ve serbest bırak (release) işlemleri

```
wait(aSemaphore)
if aSemaphore'un sayacı > 0
then
    aSemaphore'un sayacını 1 azalt
else
   çağıranı aSemaphore'un kuyruğuna koy
   kontrolü, hazırda bir göreve vermeye çalış
     -- hazır görev kuyruğu boş ise, çıkmaz (deadlock)
end

release(aSemaphore)
if aSemaphore'un kuyruğu boştur
then
   aSemaphore'un sayacını 1 artır
else
   çağıranı "hazır görev kuyruğuna" koy
   kontrolü aSemaphore un kuyruğundaki bir göreve ver
end
```

Üretici ve tüketici görevleri

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

Semaforlar ile rekabet senkronizasyonu

- `access`, adında 3. bir semafor erişimi kontrol etmek için kullanılır (rekabet senkronizasyonu)
  - **`access`** in sayacı sadece 0 vey 1 değerini alacak
  - Böyle semaforlara ikili semafor denir
- wait ve release işlemleri atomik olmalıdır (bölünemeyen)

Semaforlar için üretici kodu

```
semaphore access, fullspots, emptyspots;
access.count = 0;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
   loop
   -- produce VALUE --
   wait(emptyspots); {yer için bekle}
   wait(access);     {erişim için bekle}
   DEPOSIT(VALUE);
   release(access); {erişim hakkını geri
ver}
   release(fullspots);
   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;
```

Semaforların değerlendirmesi

- Semaforların yanlış kullanımı, işbirliği senkronizayonunda aksaklığa yol açabilir. Ör: `emptyspots` üzerine wait yapılmazsa tampon taşar.
- Semaforların yanlış kullanımı, rekabet senkronizasyonunda aksaklığa yol açabilir. Ör: `access` release yapılmazsa program çıkmaza girer (deadlock)

Monitörler

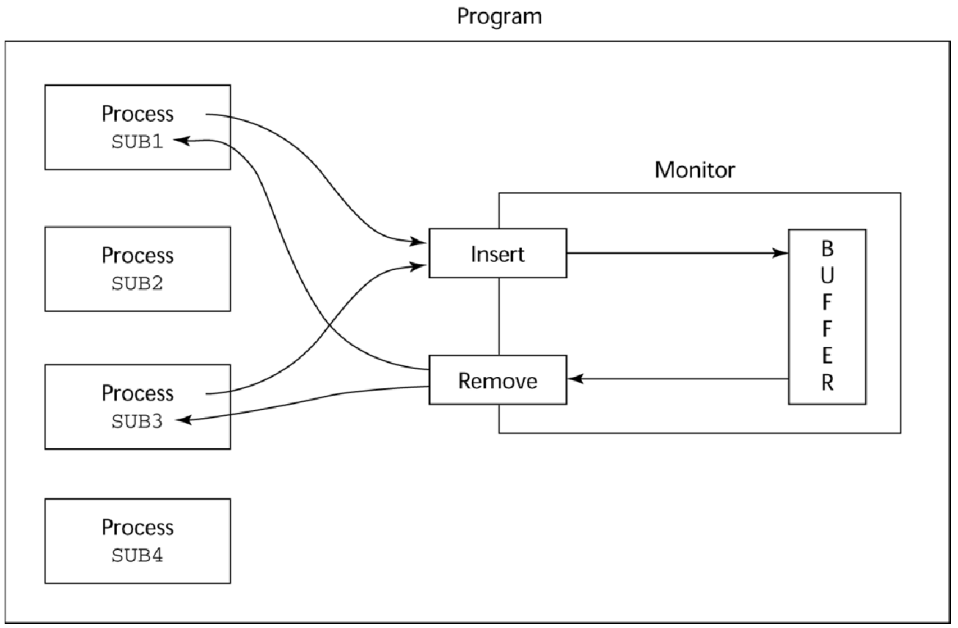
- Ada, Java, C# dillerinde var
- Ana fikir: paylaşılan veriyi ve işlemleri, erişimi sınırlandırmak için kapsülle
- Monitör, paylaşılan veri için bir soyut veri tipidir.

Rekabet senkronizasyonu

- Paylaşılan veri, monitörün içindedir
- Tüm erişim de monitörün içindedir
  - Monitörün implementasyonu, ayni anda sadece bir erişime izin vererek senkronizasyonu garanti eder.
  - Çağrı yapıldığı anda monitör meşgul ise, çağrı yapan kuyruğa konur

İşbirliği senkronizasyonu

- Süreçler (processes) arasında işbirliği hada daha bir programlama işidir
  - Programcı, paylaşlan bir tamponun sınırlarını aşmayacağını garanti etmelidir



Monitörlerin değerlendirmesi

- Rekabet senkronizasyonu için semaforlardan daha iyi
- Semafor kullanılarak monitör gerçeklenebilir
- Monitör kullanarak semafor gerçeklenebilir
- İşbirliği senkronizasyonu semaforlara benzer, problemleri de ayni

Mesaj geçme (message passing)

- Mesaj geçme, genel bir eşzamanlılık modelidir
  - Hem semaforları, hem monitörleri modelleyebilir
  - Sadece işbirliği senkronizasyonu için değil
- Ana fikir: görev iletişimi (task communication) doktoru görme gibidir – vaktin çoğunda ya siz onu beklersiniz, ya o sizi bekler. Her ikiniz de hazır olduğunuzda, buluşursunuz.

Mesaj geçme buluşması

- Mesaj geçme yöntemi ile eşzamanlı görevleri destekleyebilmek için bir dilin ihtiyaçları şöyledir:
  - Bir görevin mesaj almaya istekli olduğunu belirtebileceği bir mekanizma
  - Mesajının kabulü için bekleyenlerin kimler olduğunun hatırlanması, ve bir sonraki mesajın seçiminin adil olması
- Bir yollayıcı (sender) görevin mesajı bir alıcı (receiver) görev tarafından kabul edildiğinde, mesaj iletimine *randevu (rendezvous*) denir.

Ada dilinde eşzamanlılık desteği

- Ada 83 mesaj geçme modeli
  - Ada görevlerinin (tasks) spesifikasyon ve göve kısımları var (paketlerdeki gibi); spesifikasyon, giriş noktalarından (entry points) oluşan arayüzü barındırır

```
task Task_Example is

   entry ENTRY_1 (Item : in Integer);

end Task_Example;
```

Görev gövdesi (task body)

- Görev gövdesi, randevu gerçekleştiği zaman eylemi anlatır
- Mesaj göderen görev, mesajının kabulünü beklerken ve randevu esnasında askıya alınır
- Spesifikasyondaki giriş noktaları, gövdede **`accept`** komutu ile tanımlanır

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

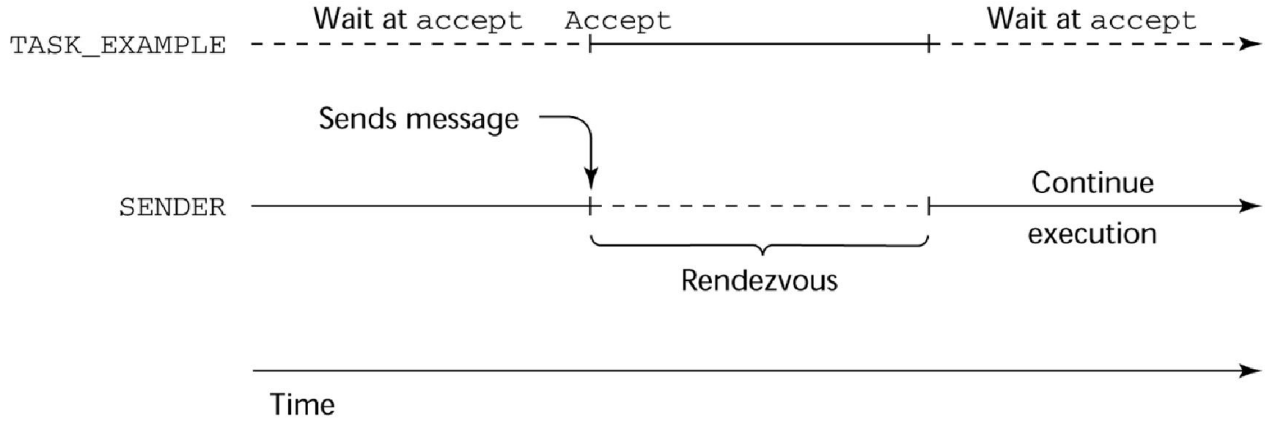
Örnek görev gövdesi

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

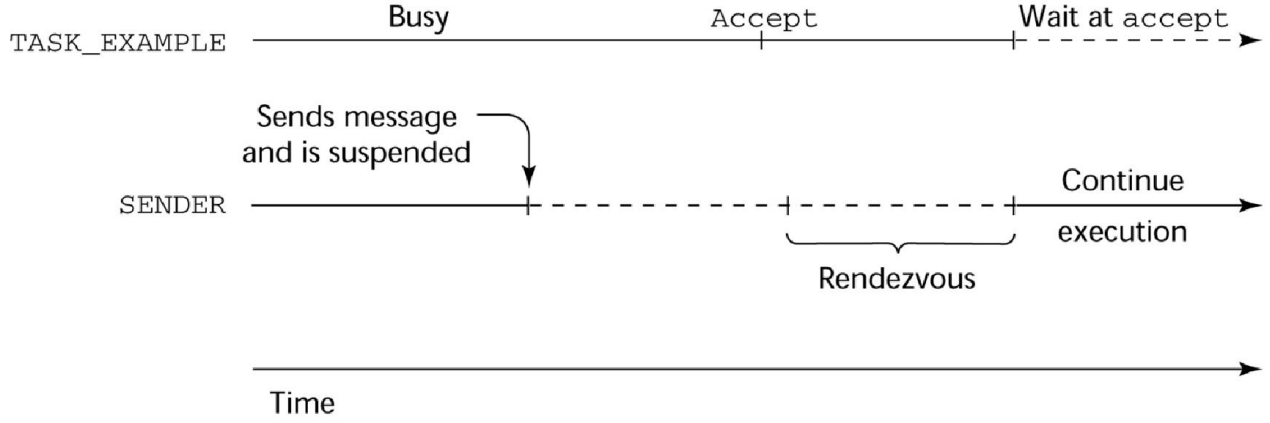
Ada mesaj geçme semantiği

- Görev, **`accept`** komutunun başına kadar çalışır ve bir mesaj bekler
- `accept` içeriği çalışırken, mesajı göderen askıya alınır
- `accept` parametreleri bilgiyi çift yönlü gönderebilir
- Her `accept` komutunun kendine ait, beklemekte olan mesajları sakladığı kuyruğu (queue) vardır

Rendevu zaman çizelgesi



(a) TASK_EXAMPLE waits for SENDER

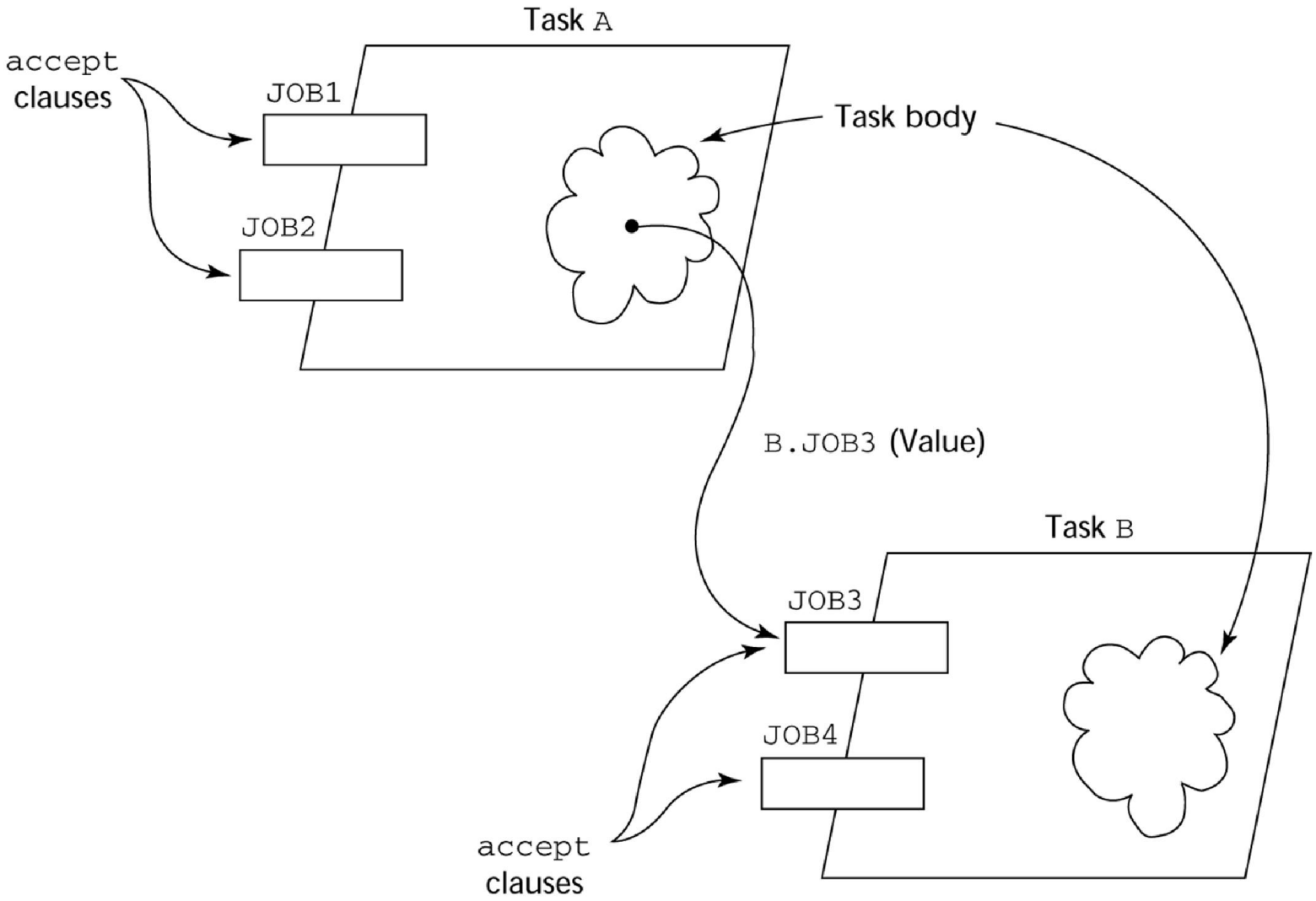


(b) SENDER waits for TASK_EXAMPLE

Mesaj geçme: sunucu (server)/oyuncu (actor) görevleri

- Sadece `accept` komutu olup da başka kodu olmayan görevlere *sunucu görevler* denir (yukarıdaki örnekte verilen *sunucu* bir görevdir)
- `accept` komutu olmayan görevlere *oyuncu görevler* denir
  - Oyuncu görev, diğer görevlere mesaj göderebilir
  - Not: Göndericinin, alıcıdaki giriş (**`entry`**) isimlerini bilmesi gerekir. Tersi olmayabilir. (asimetrik)

Randevunun grafiksel gösterimi



Çoklu giriş noktaları

- Görevlerin birden çok giriş (`entry`) noktaları olabilir
  - Görev spesifikasyonunda her giriş için bir `entry` komutu olur
  - Görev gövdesinde, `select` komutu içinde her `entry` komutu için bir `accept` komutu bulunur. Select de bir döngü içinde olur.

Çoklu girişli bir görev

```
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` komutu olan görevlerin semantiği

- Boş olmayan sadece bir tane `entry` kuyruğu varsa, ondan bir mesaj seç
- Birden çok `entry` kuyruğu boş değilse, bir tanesini tesadüfen seç ve o seçilenden mesajı kabul et
- Hepsi boş ise, bekle
- Bu yapıya *seçmeli bekleme (selective wait)* denir
- Gelişmiş (extended) `accept` komutunun kodu bittikten, ama bir sonraki accept komutu başlamadan önceki kod.
  - Çağıran ile paralel olarak çalışır

Mesaj geçme ile işbirliği senkronizasyonu

- Şartlı (guarded) **`accept`** komutları sayesinde

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

- **`when`** cümleciği olan bir **`accept`** komutu *açık (open)* veya *kapalı (closed)* olabilir.
  - Şartı doğru olan **`accept`** komutu *açık* tır.
  - Şartı yanlış olan **`accept`** komutu *kapalı* dır.
  - Şartı yoksa, **`accept`** komutu her zaman açıktır.

Şarltlı `accept` komutu olan `select` in sematiği

- `select` önce tüm accept komutlarındaki şartları kontrol eder
- Tam olarak bir tanesi açık ise, mesajına bakılır
- Birden çok açık ise, tesadüfen bir tanesi seçilir ve mesaj kuyruğu kontrol edilir
- Hepsi kapalı ise, çalışma zamanı hatasıdır
- `select` içine `else` koyarak bu hatadan kaçınabiliriz
  - `else` kısmının çalışması bittiğinde, döngü tekrar eder.

Mesaj geçme ile rekabet senkronizasyonu

- Paylaşılan veriye karşılıklı dışlayıcı erişimin modellenmesi
- Örnek—paylaşılan tampon (shared buffer)
- Tamponu ve işlemlerini bir görev içine kapsülle
- Rekabet senkronizasyonu `accept` komutlarının semantiğinde vardır
  - Sadece bir tane accept komutu herhangi bir anda çalışır vaziyettedir

Paylaşılan tampon kodu (kısmi)

```
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

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

Görev sonlanması (task termination)

- Bir görevin çalışması, eğer kontrol gövdenin sonuna gelmişse, *tamamlanmıştır (completed)*
- Eğer bir görev, kendine bağlı görevler yaratmamışsa ve tamamlanmışsa, *sonlandırılır (terminated)*
- Bir görev kendine bağlı görevler yaratmışsa, kendisi tamamlanmış olsa bile, ancak kendine bağlı olan görevler de sonlandırıldığı zaman sonlandırılır.

Özet

- Eşzamanlı çalışma komut, ifade veya altprogram düzeyinde olabilir
- Fiziksel eşzamanlılık: eşzamanlı birimleri çalıştıracak birden çok işlemci olduğu zaman
- Mantıki eşzamanlılık: eşzamanlı birimlerin tek işlemci üzerinde çalıştırılması
- Altprogram eşzamanlılığını destekleyen iki kolaylık: rekabet senkronizasyonu ve işbirliği senkronizayonu
- Mechanizmalar: semaforlar, monitörler, randevu (mesaj geçme)