

PROSES ETKİLEŞİMİ

- **AYRIK PROSESLER:** ortak veriler üzerinde işlem yapmayan prosesler.
- Örnek: A,B,C,D verilerinin en büyüğünü bulan paralel prog.

```
cobegin
  m1:= max(A,B);
  m2:= max(C,D);
coend;
m:= max(m1,m2);
```

Paralel çalışan deyimler farklı veriler üzerinde işlem yaptıkları için aralarında bir etkileşim söz konusu değildir.

PROSES ETKİLEŞİMİ

- Paralel çalışan deyimler ortak veriler üzerinde işlem yapıp, değerlerini değiştiriyor iseler, istenmeyen durumlar oluşabilir.
- Örnek:

```
j:=10;
cobegin
  (1) print j;
  (2) j:=100;
coend;
```

- Basılan j değeri kaç olacaktır?
- 10 veya 100 olabilir.
- Sonucu paralel çalışmalarını öngörülen deyimlerin [(1) ve (2)] çalışma **sırası** belirler.

YARIŞ DURUMU

- Örnekte görülen durumda sonuç paralel çalışan birimlerin çalışma **sırasına** ve **hızına** bağlıdır; önceden bilinemez.
- Yürütme sırasında, bir prosesin ne zaman aktif olacağını, hangi süre ile aktif olacağını, çalışmasının neresinde kesileceğini önceden kestirmek mümkün değildir.
- Program mantığı prosesin “çalışma hızı” üzerinde kurulu ise öngörülemeyen sonuçlar elde edilebilir.
- Ortak veriler üzerinde **yazma** türünde işlem yapan birden fazla proses bulunuyor ise, verinin son durumu hangi prosesin kesin olarak ne zaman çalıştığına bağlıdır. Bu duruma “**yarış durumu**” denir.

YARIŞ DURUMU

- Örnek: bir gözleyici proses bir olayın kaç kez gerçekleştiğini saymakla görevlidir. Bir raportör proses de belirli aralıklarla gözlenen olay sayısını çıkışa aktarır, olay sayacını sıfırlar.

```
proses gözleyici:
while true do
  begin
    ... olayı gözle
    (1) sayaç:=sayaç+1;
  end;
```

```
proses raportör:
while true do
  begin
    (2) print sayaç;
    (3) sayaç:=0;
    sleep;
  end;
```

Her iki proses “sayaç” **ortak değişkenini** kullanır.

YARIŞ DURUMU

- Hata oluşturabilecek örnek bir senaryo:
- Deyimlerin çalışma sıraları:
(1) »(2) »(1) »(3)

```
proses gözleyici:
while true do
begin
... olayı gözle
(1) sayaç:=sayaç+1;
end;
```


```
proses raportör:
while true do
begin
(2) print sayaç;
(3) sayaç:=0;
sleep;
end;
```

- (1) : Gözleyici sayaç değerini (1) ile 5'e yükseltir ve kesilir.
- (2) : Raportör 5 değerini (2) ile çıkışa aktarır ve kesilir.
- (1) :Gözleyici yeni bir olay gözler, sayacı (1) ile 6'a yükseltir ve kesilir.
- (3) : Kaldığı yerden çalışan raportör (3) ile sayaç değerini sıfırlar.

SONUÇ: aradaki bir olay kaydedilemedi.

YARIŞ DURUMU

- Hata oluşturabilecek örnek durumlar:
Deyimler birden fazla makina komutu ile gerçekleştirilir.

sayaç:=sayaç+1 

- a) LOAD SAYAÇ
- b) ADD 1
- c) STORE SAYAÇ

YARIŞ DURUMU

a) LOAD SAYAÇ b) ADD 1 c) STORE SAYAÇ	proses gözleyici: while true do begin ... olayı gözle (1) sayaç:=sayaç+1; end;	proses raportör: while true do begin (2) print sayaç; (3) sayaç:=0; sleep; end;
---	---	---

- Deyimlerin çalışma sıraları:

(a) »(2) »(3) »(b) »(c)

- (a): Gözleyici yeni bir olay için sayaç değerini artırmak üzere (a) yürütür ve ACC'e 5 değerini yükler ve kesilir.
- (2): Raportör çalışır, (2) ile sayaç değerini 5 olarak çıkışa yazar,
- (3): Raportör (3) ile sayacı sıfırlar ve kesilir.
- (b): Gözleyici çalışır, (b) ile ACC içeriğini 1 artırır ,
- (c): Gözleyici(c) ile 6 değerini sayaç değişkenine atar.

SONUÇ: zaten rapor edilmiş olan 5 olay tekrar kayıt edildi.

YARIŞ DURUMU

- SONUÇ:

Paralel çalışan prosesler paylaşılan ortak değişkenler üzerinde yazma işlemleri gerçekleştiriyor iseler, yarış durumları nedeniyle tutarsız sonuçlar elde edilmesi mümkündür.

- Yarış durumları fiziksel ve lojik paralellik durumlarının her ikisinde de görülür. Fiziksel paralellik var ise, işlemciler farklı hızlarda çalışırlar. Lojik paralellikte ise, zaman paylaşımlı çalışma nedeniyle proseslerin çalışmalarının hangi noktasında kesilecekleri bilinemez.

KARŞILIKLI DIŞLAMALI ERİŞİM

- Yarış durumlarının yaratacağı belirsizlikleri önlemek için, ortak değişkenlere **yazma** türündeki erişimlerin “**birbirlerini dışlayarak**” gerçekleşmeleri sağlanmalıdır. Bu tür çalışmaya “**karşılıklı dışlamalı erişim**” (mutual exclusion:mx) denir.
- Sorun karşılıklı dışlamayı gerçekleyecek bir yapı ile giderilir.
- Paralel kod içinde ortak değişkenlere erişimin yer aldığı deyimler karşılıklı dışlamayı gerçekleyen yapı ile çevrelenir.
- Dışlamayı gerçekleyen yapı: **mx_begin...mx_end**
- Ortak değişkenlere erişilen deyimler bloğu: **kritik bölüm**
- AMAÇ: **belirli bir anda kritik bölümü içinde aktif olan bir tek prosesin bulunmasını sağlamak.**

KARŞILIKLI DIŞLAMALI ERİŞİM

process P1	process P2
begin	begin
....
mx_begin // kritik bölüme gir	mx_begin //kritik bölüme gir
..... // kritik bölüm // kritik bölüm
mx_end; // kritik bölümden çık	mx_end; // kritik bölümden çık
...	...
end;	end;

- Bir proses kendi kodu içindeki kritik bölümüne ilerlemeden **önce** mx_begin, bölüm içindeki işlemleri tamamladıktan **sonra** da mx_end yürütmelidir. Bu adımlar belirli bir anda sadece bir prosesin kendi kritik bölümü içinde aktif olmasına izin vereceklerdir.

KARŞILIKLI DIŞLAMALI ERİŞİM

- **mx_begin:**
 1. Kritik bölümüne girmiş ve henüz mx_end yürütmemiş başka proses var mı? Eğer var ise, bu deyimi yürüten prosesi beklet (kritik bölümüne ilerlemesini engelle).
 2. Eğer yoksa, prosesi kiritik bölüme ilerlet ve diğer prosesleri bu durumdan haberdar etmek üzere bir işaret oluşturun.
- **mx_end:**
 1. Kritik bölümüne geçmek için bekleyen proses var ise, izin ver.

mx_begin ve mx_end : GERÇEKLENMESİ

- ÇÖZÜM (???)

Bir prosesin kritik bölümünü yürütmekte olduğunu belirtmek üzere bir bayrak (meşgul) kullan:

meşgul:true --- kritik bölümünde aktif bir proses var.

meşgul:false --- kritik bölümünde aktif bir proses yok.

```
mx_begin: do while (meşgul)
            end; // kritik bölümünü yürüten proses bitirene
                // kadar while içinde bekle
            meşgul:= true; // kritik bölümüne girdiğini göster
```

```
mx_end: meşgul:=false; // kritik bölümünden çıktığını göster
```

Yazılım ile Çözüm

- Önerilen çözüm **HATALI** - karşılıklı dışlama koşullarını sağlayamaz
 - Çözüm olarak kullanılan **meşgul** değişkeninin kendisi de, erişimi denetim altında tutulmak istenen bir ortak bir değişken!
 - Hatalı çalışma örneği:
 - meşgul=False
 - P1: while (meşgul)—FALSE bulur ve çevrimden çıkar ve hemen çalışması kesilir
 - P2: while (meşgul)—FALSE bulur ve çevrimden çıkar, meşgul←TRUE ve KB'ne ilerler. Bir süre sonra kesilir
 - P1 kaldığı yerden çalışmaya başlar: meşgul←TRUE ve KB'ne ilerler
- SONUÇ:** Her iki proses de KB içinde aktif→hatalı çalışma düzeni

KARŞILIKLI DIŞLAMANIN GERÇEKLENMESİ

- Bu öneri doğru sonuç getirmeyecektir çünkü ortak değişkenlere erişimi denetlemek için, kendisi de bir ortak değişken olan **bayrak** alanına denetimsiz erişim söz konusudur.
- Hatalı çalışma örneği:

meşgul=False

- P1: while (meşgul)—FALSE bulur ve çevrimden çıkar ve hemen çalışması kesilir
 - P2: while (meşgul)—FALSE bulur ve çevrimden çıkar, meşgul←TRUE ve KB'ne ilerler. Bir süre sonra kesilir
 - P1 kaldığı yerden çalışmaya başlar: meşgul←TRUE ve KB'ne ilerler
- SONUÇ:** Her iki proses de KB içinde aktif→hatalı çalışma düzeni

KARŞILIKLI DIŞLAMANIN GERÇEKLENMESİ

- Çözüm şu sınırlamalara cevap vermelidir:
 1. Donanımdan bağımsız bir çözüm bulunacaktır. Donanıma bağlı özel bir komut (test&set, compare&swap) kullanılamaz.
 2. Paralel proseslerin yürütülme hızları hakkında bir ön tahminde bulunulamaz.
 3. Kendi kritik bölümü dışında aktif olan bir proses diğerlerin kritik bölümlerine ilerlemelerine engel olamaz.
 4. Prosesler hiç bir zaman kritik bölümlerine ilerlemek için sonsuz beklemeye giremezler (indefinite postponement).

KARŞILIKLI DIŞLAMANIN GERÇEKLENMESİ

- Çözümler gerçekleşme yöntemine göre iki sınıfa ayrılır:
 1. **Meşgul bekleme**ye dayalı çözümler
 2. **Askıya alınmaya** dayalı çözümler.
 - **Meşgul bekleme**: kritik bölüme geçmesi uygun olmayan proses bir meşgul bekleme çevrimine sokularak ilerlemesi engellenir. Kullanıcı düzeyinde gerçekleştirilebilir.
 - **Askıya alınma**: kritik bölüme geçmesi uygun olmayan prosesin elinden işlemci alınarak bir bekleme kuyruğuna yerleştirilir. İşletim sistemi desteği gerektirir.

Meşgul Bekleme (busy waiting)

Meşgul Bekleme :

- İşlemci zamanı boşa harcanır - proses, sorgulama sonucu değişmeyeceği halde (*sorgulama sonucu ancak bir başka prosesin çalışmasıyla değişebilir*), **kendine ayrılan zaman dilimini** bir while çevrimi içinde sorgulama işlemini sürdürerek geçirir:

KARŞILIKLI DIŞLAMANIN GERÇEKLENMESİ

- Çözüm: iki paralel proses (P1,P2) için geçerli.

ortak değişken turn:=1;

yerel değişkenler my_turn ve his_turn

Process P1: **Process P2:**
my_turn:=1; my_turn:=2;
his_turn:=2; his_turn:=1;

mx_begin:
do while
 turn < my_turn;

end;
mx_end:
 turn:= his_turn;

- Geçerli bir çözüm, ancak kısıtlamaları var:

- 1) prosesler sadece birbirlerini izleyen sırayla KB'e girebilir - kendi KB'ü dışındaki bir prosesin bir başka prosesin KB'e girmesini engellememe ilkesine aykırı
- örnek: proseslerden birinin daha yavaş çalışması durumunda, diğeri onu beklemek durumunda kalacaktır
- 2) proses sayısına bağlı çözüm - sadece iki proses için geçerli

DEKKER ALGORİTMASI

- Dekker Çözümü: iki paralel proses (P1,P2) için geçerli.
 ortak değişkenler: turn:=1; need[1]:=false, need[2]:=false,
 yerel değişkenler: me ve other; P1(me:=1, other:=2)/ P2(me:=2, other:=1)
- ```

mx_begin:
1. need[me]:= true; // KB'e girme isteğini bildir
2. while need[other] do // diğerinin isteği kalkana kadar bekle
 begin
3. if turn < me then
 begin 4. need[me]:= false; //geçici bir süre isteğinden vazgeç
5. while (turn < me) do; // sıranı bekle-meşgul bekleme
6. need[me]:= true; //isteğini tekrar bildir
 end;
 end (while);

mx_end:
begin need[me]:= false; turn:=other; end;

```

## DEKKER ALGORİTMASI- Örnek Senaryo 1

```

mx_begin:
1. need[me]:= true; // KB'e girme isteğini bildir
2. while need[other] do // diğerinin isteği kalkana kadar bekle
 begin
3. if turn < me then
 begin 4. need[me]:= false; //geçici bir süre isteğinden vazgeç
5. while (turn < me) do; // sıranı bekle-meşgul bekleme
6. need[me]:= true; //isteğini tekrar bildir
 end;
 end (while);

mx_end:
begin need[me]:= false; turn:=other; end;

```

turn:= 1;

**Senaryo 1:** P1:aktif, P2:çalışmıyor (need[2]:= false)

1. need[1]:= true: 2. while'dan çıkar ve KB'e ilerler.

## DEKKER ALGORİTMASI- Örnek Senaryo 2

```

mx_begin:
1. need[me]:= true; // KB'e girme isteğini bildir
2. while need[other] do // diğerinin isteği kalkana kadar bekle
 begin
 3. if turn < me then
 begin
 4. need[me]:= false; // geçici bir süre isteğinden vazgeç
 5. while (turn < me) do; // sıranı bekle-meşgul bekleme
 6. need[me]:= true; // isteğini tekrar bildir
 end;
 end (while);
 end;
mx_end:
begin need[me]:= false; turn:=other; end;

```

turn:= 1;

**Senaryo 2:** P1: KB içinde, P2 çalışır.

1. need[2]:= true;
  2. while'a girer, need[1]:= true olduğu için,
  3. if turn < 2 doğru olduğu için 4. need[2]:= false ile isteğini geri çeker ve
  5. while (turn < 2) do ile meşgul beklemeye başlar.
- \*\*** Ne zamana kadar? P1 mx\_end yürütünce, turn=2 olacağından 5. while'dan çıkar. 6. need[2] ile geri çekmiş olduğu isteğini günceller ve 2. while'a geri döner, ancak artık need[1]:= false olduğundan, tekrar çevrime girmez ve KB'e ilerler.

## DEKKER ALGORİTMASI- Örnek Senaryo 3

```

mx_begin:
1. need[me]:= true; // KB'e girme isteğini bildir
2. while need[other] do // diğerinin isteği kalkana kadar bekle
 begin
 3. if turn < me then
 begin
 4. need[me]:= false; // geçici bir süre isteğinden vazgeç
 5. while (turn < me) do; // sıranı bekle-meşgul bekleme
 6. need[me]:= true; // isteğini tekrar bildir
 end;
 end (while);
 end;
mx_end:
begin need[me]:= false; turn:=other; end;

```

**Senaryo 3:** P1: KB içinde, P2 çalışır.

**\*\*** noktasına kadar Senaryo 2 ile aynı adımlar: P2 beklemede

Ne zamana kadar? P1 mx\_end yürütünce, turn=2 olacağından, P2 5. while'dan çıkar, 2. while'a geri döner ve kesilir.

P1 çalışır, tekrar KB'e girmek üzere 1. need[1]:= true atamasını yapar ve 2. while çevrimine girer. 3. if turn < 1 doğru olduğunu görür ve kesilir.

P2 çalışınca 2. while'a geri döner ve need[1]:= true olduğu için çevrime girer, 3. if turn < 2 doğru olmadığından meşgul bekleme çevrimine girmez ve 2. while geri döner. 2.-3. adımları P2 kesilene kadar tekrarlanır.

P1 çalışıp, 4. need[1]:= false; ile isteğini geri çektikten sonra ancak 2. while çevrimine tekrar girmez ve KB'e ilerler.

## PETERSEN ALGORİTMASI

- Petersen Çözümü: iki paralel proses (P1,P2) için geçerli.  
ortak değişkenler: enter1, enter2 (false, false)  
turn:P1 (veya P2)

process P1:

**mx\_begin:**

enter1:=true; *//girme isteğini bildir*

turn:= P2; *// öncelik sırasını diğer prosese ver*

while (enter2) and (turn==P2) do; *//sıra diğerinde ise bekle*

**mx\_end:**

enter1:= false; *//isteğini geri çek*

## Petersen Algoritması-Örnek Senaryolar

- enter1, enter2 (false, false)  
turn:P1 (veya P2)  
**process P1:**  
**mx\_begin:**  
enter1:=true; *//girme isteğini bildir*  
turn:= P2; *// öncelik sırasını diğer prosese ver*  
while (enter2) and (turn==P2) do; *//sıra diğerinde ise bekle*  
**mx\_end:**  
enter1:= false; *//isteğini geri çek*

### Senaryo 1:

- P1 çalışıyor, P2 pasif  
istek\_1=TRUE ve sıra=P2  
istek\_2=FALSE olduğu için P1 while döngüsünü geçer → P1 KB içinde
- P1 KB'de, P2 KB'ye girmek ister  
istek\_2=TRUE ve sıra=P1;  
istek\_1=TRUE olduğu için P2 while döngüsünde bekler
- P1 mx\_end yürütünce P2 döngüden çıkar



## DIJKSTRA ALGORİTMASI

```

process Pi;
int j;
mx_begin:
 L0: b[i]:=false;
 L1: if k < i then // k bu prosese işaret etmiyor ise
 begin
 c[i]:= true;
 if b[k] then k:=i; // Pk KB'sinden çıkmış ise aday ol
 go to L1;
 end;
 L4: else begin
 c[i]:= false;
 for j:=1 to n do
 if j < i and not c[j] then go to L1;
 end;
mx_end:
 c[i]:= true; b[i]:= true;

```

## DIJKSTRA ALGORİTMASI

- Çevrimden çıkmanın tek yolu L4'ü L1'e geri dönmeden bitirmektir. Proses kendisi dışında tüm c[j] leri true bulmalı. Meşgul çevrimi içindeki tüm proseslere ait b dizisi elemanları false değerini alacaklardır. Beklemede olmayan k. prosese ait b[k] true olacak ve tüm diğer prosesler k < i bulacaklardır.