İşletim Sistemlerine Giriş

Süreçler Arası İletişim
(IPC-Inter Process Communication)

Süreçler Arası İletişim

Süreçler, sıklıkla birbirleri ile iletişim kurarlar. Bir sürecin çıktısı başka bir sürecin girdi verisi olabilir. Süreçler arası iletişimde üç konu önemlidir:

1. Bir süreç diğerine nasıl veri gönderir?
2. Bir ya da daha fazla sürecin birbirlerinin yollarını kullanırken dikatli olmaları ve birbirlerinin iletişim yollarına girmemeleri.
3. İletişimdeki uygun sıra nasıl olmalıdır? Bir süreç veri gönderiyor diğeri bu veriyi yazdırıyorsa, ilk süreç veri göndermediğinde ikincinin beklemesi ya da ikinci yazdırırken birincinin beklemesi gereklidir. İletişimde kullanılacak sıra önemlidir.

Süreçler Arası İletişim

Süreçler arası iletişimde kullanılan yöntemler, thread ler arasında da kullanılır çünkü threadlerde süreçlerdeki gibi aynı verileri ortak kullanmaktadırlar.

İletişim Niçin Yapılır ?

1.Kaynak paylaşımı (dosya, I/O aygıtı,...)
2.Karşılıklı haberleşme (iki süreç birbirine haber gönderir)
3.Senkronizayon (Bir sürecin çalışması başka bir sürecin belirli işlemleri tamamlamış olmasına bağlı olabilir)

Süreçler Arası İletişim

İki ya da daha fazla süreç iletişim kurarken ortak bellek bölgelerini(shared memory), ortak dosyaları,tüm süreçlerin erişebileceği kuyruk yapılarını, özel iletişim dosyalarını ve buna benzer sistemler kullanabilir.

Tüm iletişim yöntemlerinde süreçlerin iletişim için kullandıkları mekanizmanın elemanları, süreçlerce ortaklaşa kullanılır.

Yarış Durumları (Race Conditions)

İletişime bir örnek verelim: Süreçler istedikleri dosyaları yazıcıdan çıktı almak istesinler.

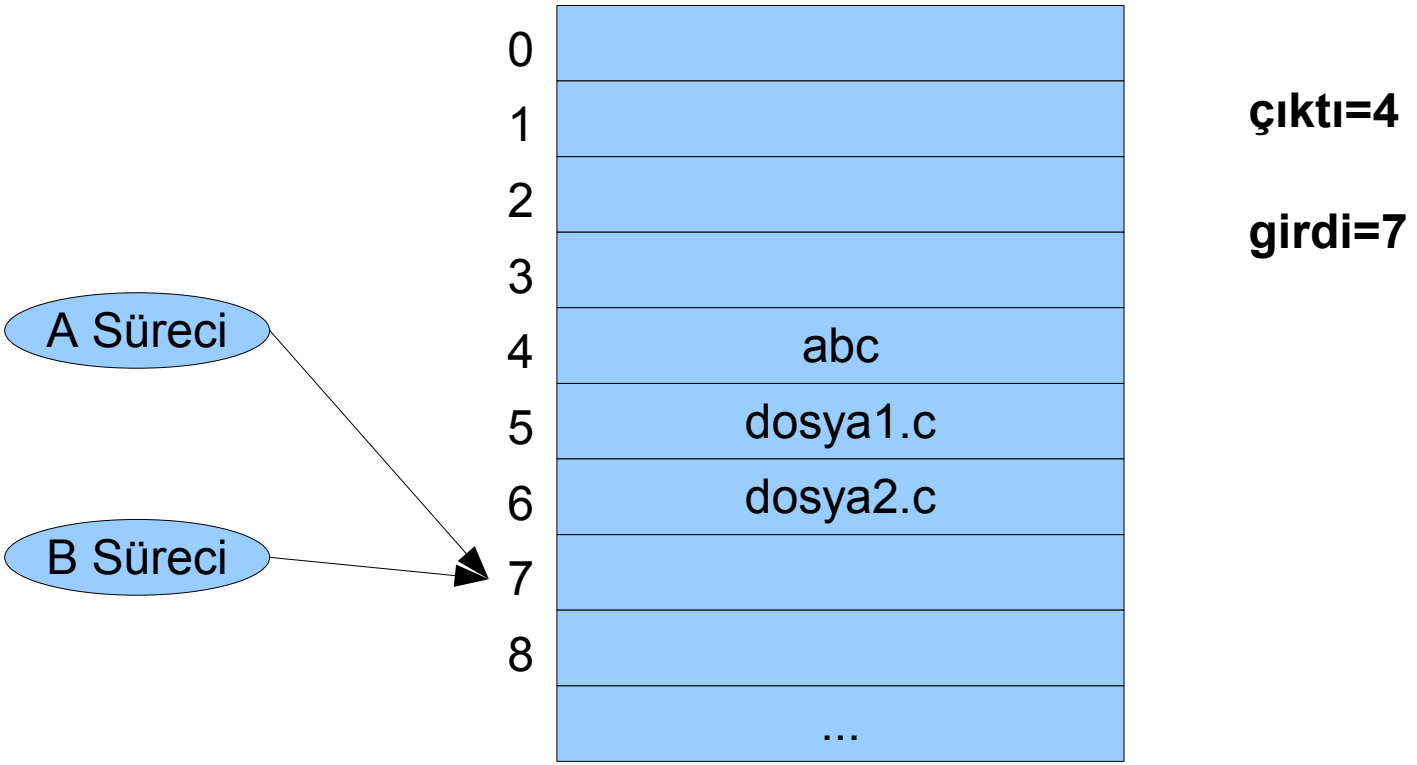
Bir süreç bir dosyayı yazdırmak istediğinde, bekletici(spooler) ismi verilen bir dosyaya yazdırmak istediği dosya bilgisini ekler.

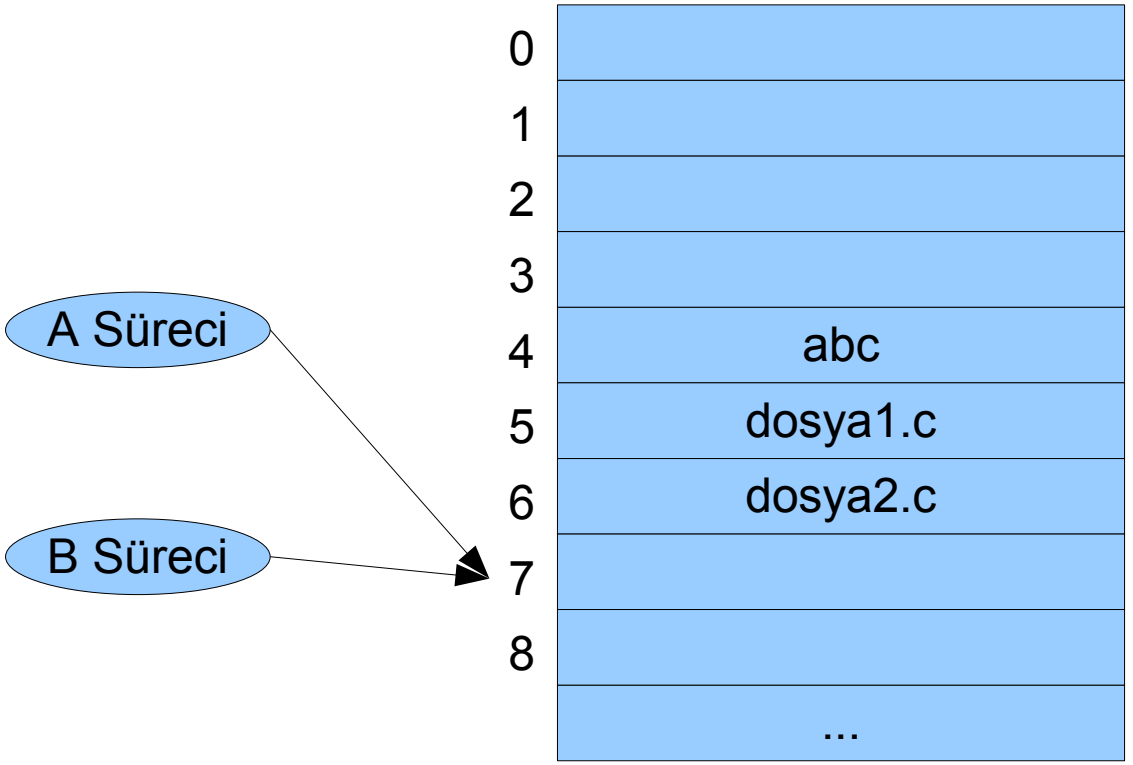
Başka bir süreç belirli sürelerde periyodik olarak klasörü kontrol eder, eğer bir dosya görürse bu dosyayı yazdırır.

Yazdırma işleminden sonra yazdırdığı dosyayı bekletici(spooler) klasöründen siler.

Yarış Durumları (Race Conditions)

Varsayalım ki bekletici klasöründe birden fazla boş hücre bulunsun, yazıdırılacak olan dosyalar bu boş hücrelere konulsun. ***çıktı*** bir sonraki yazıdırılacak olan dosyayı, ***girdi*** de bir sonraki boş hücre bilgisini tutsun.





A süreci girdi değerini ***7*** olarak okusun. Okuduktan hemen sonra işlemci süreçler arasında geçiş yapsın ve B süreci seçilsin. B süreci de girdi değerini ***7*** olarak okuyacaktır. B süreci ***girdi=7*** alanına kendi yazdırmak istediği dosyanın bilgisini eklesin ve girdi değerini ***8*** yapsın. B sürecinin çalışması kesilsin ve tekrar A sürecine geri dönülsün.
A süreci bir sonraki boş hücre değerini yani girdi değişkeninin değerini ***7 olarak*** görmektedir. A süreci de yazdırmak istediği dosya bilgisini ***7.*** hücreye ekler ve ***girdi=8*** yapar.
B sürecinin yazdırmak istediği dosya bilgisi silinir.
Bu tip durumlara ***YARIŞ DURUMLARI (RACE CONDITIONS)*** denilir.

Kritik Bölümler (Critical Sections)

Bir program içerisinde ortak olarak paylaşılan alanlara erişim yapılan kod bölümlerine ***kritik bölge(critical section)*** denilir.

Yarış durumlarından sakınmak için karşılıklı dışlama (mutual exclusion) yöntemi kullanılır. Bir süreç ortak değişken üzerinde işlem yapıyorsa başka bir süreç bu değişken üzerinde işlem yapamaz ve dışlanır.

Örnekteki problem, A sürecinin girdi değişkeni ile işi bitmeden B sürecinin bu değişkeni kullanmasından kaynaklanmaktadır.

Karşılıklı Dışlamada Dikkat Edilmesi Gerekenler

1. İki süreç aynı anda kritik bölge içerisine girmemelidir.
2. Kritik bölüme girmek isteyen süreçler eğer başka bir süreç kritik bölgede değilse engellenmemelidir.
3. Bir süreç kritik bölge içinde sonsuza kadar beklememelidir.
4. Kritik bölge dışında çalışan bir süreç, başka süreçleri bloklayamaz.
5. Sistemdeki işlemci sayısı ve hızı ile ilgili kabuller yapılmamalıdır, bu değerlerden bağımsız olmalıdır.

Karşılıklı Dışlama Örneği

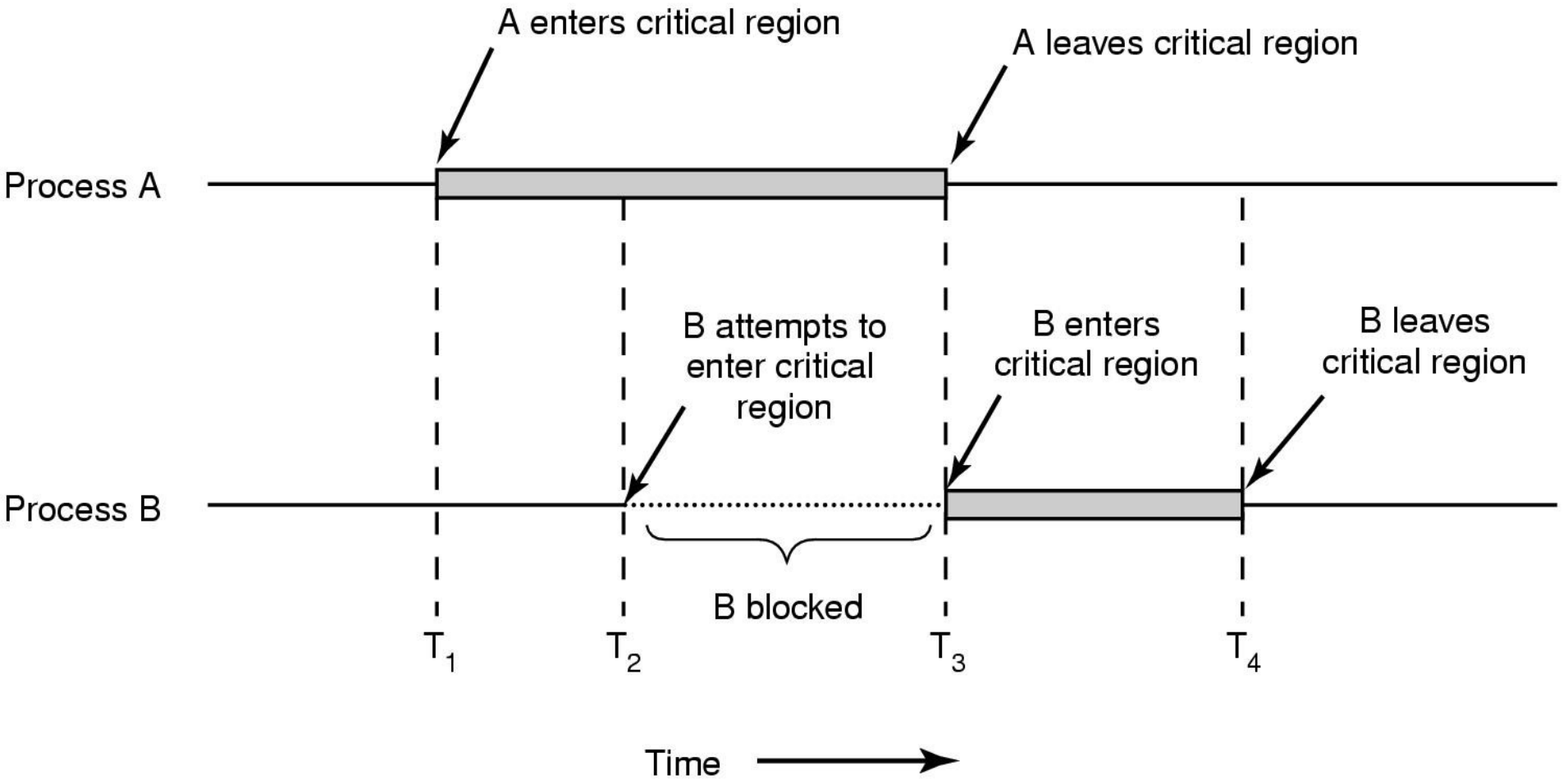
Süreç 1:

```
while (TRUE){
    //kritik bölge dışındaki  kodlar
    karsilikli_disla_basla
        //kritik bölge kodları
    karsilikli_disla_bitir
    //kritik bölge dışındaki kodlar

}
```

Süreç 2:

```
while (TRUE){
    //kritik bölge dışındaki  kodlar
    karsilikli_disla_basla
        //kritik bölge kodları
    karsilikli_disla_bitir
    //kritik bölge dışındaki kodlar

}
```

Karşılıklı Dışlama Örneği



Yoğun Beklemeli Karşılıklı Dışlama (mutual exclusion with busy waiting)

1.Kesmeleri Devre Dışı Yapmak (Disable Interrupts)

Süreçler kritik bölgeye girdiklerinde tüm kesmeleri devre dışı bırakabilirler. Kritik bölge dışına çıktıklarında eski durumuna getirebilirler.
Bu durumda eğer bir süreç bu işlemi uygulasa ve çıktığında eski durumuna getirmez ise sistem çalışmaz duruma gelir.
Sistemde birden fazla işlemci varsa bu işlem sadece tek bir işlemciyi etkiler. Kullanıcı süreçlerine bu hakkı vermek oldukça tehlikelidir. Çekirdek bazen bu yöntemi kullanır.

Yoğun Beklemeli Karşılıklı Dışlama (mutual exclusion with busy waiting)

2.Değişkenleri Kilitleme (Locking Variables)

Süreçler arasında paylaşılan bir değişkeniniz olsun. Bu değişkenin ilk değeri 0 olsun. Bir süreç kritik bölgeye girdiğinde ilk önce bu değişkeni kontrol etsin. Eğer kilit değişkenimizin değeri 0 ise süreç bunu 1 yapsın ve kritik bölgede yapmak istediklerini yapsın. Eğer 1 ise, 0 oluncaya kadar beklesin.

Bu yöntemde de bahsedilen problem vardır.
Bir süreç kilit değerini 0 olarak okusun, 1 yapamadan işlemci başka bir sürece geçerse, her iki süreçte kritik bölgeye girmiş olur.

Yoğun Beklemeli Karşılıklı Dışlama (mutual exclusion with busy waiting)

3.Dikkatli Değiştirme (Strict Alternation)

Bu yöntem aşağıdaki kodda gösterilen maktık temelinde çalışmaktadır. Bu tip beklemeye ***yoğun bekleme*** denilir. Yoğun bekleme kullanan kilitlere ***döngü kilit(spin lock)*** denilir.

Süreç 1:

```
while (TRUE){
    while (kilit!=0); //bekle
    kritik_bolum();
    kilit=1;
    kritik_olmayan_bolum();
}
```

Süreç 2:

```
while (TRUE){
    while (kilit!=1); //bekle
    kritik_bolum();
    kilit=0;
    kritik_olmayan_bolum();
}
```

Yoğun Beklemeli Karşılıklı Dışlama (mutual exclusion with busy waiting)

3.Dikkatli Değiştirme (Strict Alternation)

Problemler:
* İşlemci zamanı boşuna harcanır.(Bir sürecin işlemci zamanı while döngüsünde geçer.)

* İkiden fazla süreç için geçerli değildir.

* Bir sürecin kritik olmayan bölgesi diğer sürecin kritik bölgeye girmesine engel olur.

Örneğin, 1. süreç kritik bölgeden çıksın ve kiliti=1 yapsın. 2. süreç de kritik bölgedeki işini çok hızlı bitirsin ve kiliti=0 yapsın. Şu anda iki süreçde kritik olmayan bölümde olsunlar. 1. süreç kritik bölgeye girsin ve hemen çıkarak kilit=1 yapsın. 1. süreç, 2. sürecin kritik olmayan bölgesi bitmeden tekrar kritik bölgeye girmeye çalışırsa giremez.

Yoğun Beklemeli Karşılıklı Dışlama (mutual exclusion with busy waiting)

4.Peterson'ın Çözümü (Peterson's Solution)

peterson_s.c

```
#define FALSE 0
#define TRUE 1
#define N 2

int sira; //kimin sirasi
int ilgili[N];  //ilk degerler FALSE dir.

void kritik_bolgeye_gir(int surec){
     int diger_surec;
     diger_surec=1-surec;
     ilgili[surec]=TRUE;
     sira=surec;
     while (sira==surec && ilgili[diger_surec]==TRUE);
}

void kritik_bolgeden_cik(int surec){
     ilgili[surec]=FALSE;
}
```

Her süreç kritik bölgeye girmeden *önce kritik_bolgeye_gir()* metodunu,

çıktıktan sonra da *kritik_bolgeden_cik()* metodunu kendi süreç numarası ile çalıştırır.

peterson_s.c

```
#define FALSE 0
#define TRUE 1
#define N 2

int sira; //kimin sirasi
int ilgili[N];  //ilk degerler FALSE dir.

void kritik_bolgeye_gir(int surec){
     int diger_surec;
     diger_surec=1-surec;
     ilgili[surec]=TRUE;
     sira=surec;
     while (sira==surec && ilgili[diger_surec]==TRUE);
}

void kritik_bolgeden_cik(int surec){
     ilgili[surec]=FALSE;
}
```

0. ve 1. süreç aynı anda kritik_bolgeye_gir() metodunu çalıştırsınlar. 0. süreç kaynak ile ilgili olduğunu belirtsin ve sira=0 yapsın.
1. süreçte ilgili olduğunu belirtsin ve sira=1 yapsin.
0. süreç while döngüsüne girdiğinde hemen çıkar ve bu şekilde kritik bölge kodunu yapmaya başlar.
1. süreç ise while döngüsünde bekler. 0. süreç çalışmasını bitirdiğinde ve ilgisini FALSE yaptığında, 1. süreçde kritik bölgeye girer ve yapması gerekli olan işleri yapar.

Yoğun Beklemeli Karşılıklı Dışlama (mutual exclusion with busy waiting)

4.TSL Komutu (TSL instruction-*Test and Set*)

Bir çok bilgisayar; özellikle birden fazla işlemcili olanlar, TSL (Test and Set Lock) komutuna sahiptir. Bu komut kesilemez ve tek bir komut çevriminde(instruction cycle) gerçekleşir. Bu komut bir bellek gözünün değerini yazmaça okur ve belleğe sıfır olmayan bir değer yazar. Bu komut bitmeden başka CPU bu belleğe erişemez.

```
kritik_bolgeye_gir:
        tsl yazmac,bayrak
        cmp yazmac,#0
        jnz kritik_bolgeye_gir
        ret
```

```
kritik_bolgeden_cik:
        mov bayrak,#0
        ret
```

Üretici – Tüketici Problemi

Peterson algoritması ve TSL komutu da doğru bir şekilde kullanılabilir fakat bu metotlar yoğun beklemeyi yani işlemcinin boşuna kullanılmasını gerektirir.

Ayrıca yoğun olarak bekleyen süreç, kritik bölgede çalışan bir süreçten daha öncelikli olabilir.

Bu durumda zamanlayıcı(scheduler) her zaman yoğun bekleyen sürece çalışma hakkı verebilir, bu şekilde sonsuza kadar iki süreçde bekler.

Yoğun bekleme yerine, beklenen sürecin bloklandığı çeşitli yöntemler vardır. ***sleep()*** ve ***wakeup()*** gibi sistem çağrıları, işletim sistemlerinde bulunur. sleep() süreci bloklar, wakeup() da verilen süreci uyandırır ve hazır durumuna getirir.

Üretici – Tüketici Problemi

Örneğin, iki süreç ortak bir tampon alanı kullansınlar. Bu alana konulan değeri üretici üretsin, tüketici de bu değeri alarak kullansın.

Üretici eğer tampon boş ise yeni bir değer üretip koymalı, tüketici de sadece tamponda değer var ise bu alanı okumalıdır. Diğer durumlarda süreçler uyumalıdır.

Üretici yeni bir ürünü tampon alanına koyduğunda tüketiciyi uyandırır ve kendisi uyur. Tüketici de tampon alanındaki değeri aldıktan sonra üreticiyi uyandırır ve kendisi uyur.

Üretici – Tüketici Problemi

uretici_tuketici.c

```
#define TRUE 1
#define FALSE 0
#define N 100   //tampondaki bos hucre sayisi

int sayac=0;

void uretici(void){
     int urun;
     while (TRUE){
          urun_uret(&urun);
          if (sayac==N) uyu();
          urunu_koy(urun);
          sayac= sayac+1;
          if (sayac==1) uyan(tuketici);
     }
}
void tuketici(void){
     int urun;
     while(TRUE){
        if (sayac==0) uyu();
        urun_al(&urun);
        sayac = sayac -1;
        if (sayac == N-1) uyan(uretici);
        urunu_tuket(urun);
     }
}
```

Burada da bir yarış durumu olabilir. sayac alanına ortak erişim yapıldığı ve kontrolsüz yapıldığı için hata oluşabilir.

Örneğin, tampon boş olsun. Tüketici sayac alanını 0 okur. Bu noktada zamanlayıcı üreticiyi çalıştırsın, üretici yeni bir ürünü tampona eklesin. sayac ı arttırsın.

Üretici tüketiciyi uyandırmak için uyan() metodunu çağırsın.

Tüketici uyu() metodunu tam olarak çalıştırmadan uyan() metodu çalıştırılır. tuketici sayac değerini 0 okuduğu için uyur.

Her seferinde üretici yeni bir değeri tampona ekler, en sonunda her iki süreçde sonsuza kadar uyurlar.

Semaforlar (Semaphores)

Yoğun bekleme gerektirmeyen, süreç sayısından bağımsız bir yöntemdir.

E.W. Dijkstra tarafından 1965 de geliştirilmiştir. Yazılım ve işletim sistemi desteği ile çalışır.

DOWN =(P) ve UP =(V) isimli iki metodu bulunmaktadır. Bu metotlar kesilemez metotlardır ve işletim sistemi tarafından gerçekleştirilirler. Genelde kesmeler iptal edilerek semafora erişilir. Birden fazla işlemci varsa TSL komutu ile semafor korunmalıdır.

Semafor adı verilen özel bir değişkene erişim için kullanılırlar.

Bu değişkene sadece bu metotlar ile erişilebilir.

Semaforlar (Semaphores)

s değişkeni semafor olsun.

DOWN = P metodu =kritik_bolgeye_gir

```
DOWN(s){
    if (s>0)
        s=s-1;
    else
        s_de_bekle(uyu);
}
```

UP = V metodu =kritik_bolgeden_cik

```
UP(s){
    if (s_de_bekleyen)
        sıradakini_aktif_yap;
    else
        s=s+1;
}
```

semaforlar tamsayı değerleridir (>=0). Kritik bölge erişimleri ikili semafor (binary semaphore = mutex) kullanılarak yapılır. mutex ve semaphore veri yapıları ve metotları işletim sistemlerinde bulunur.

Semafor Örnek

uretici_tuketici_semafor.c

```
#define TRUE 1
#define FALSE 0

typedef int semafor;

semafor mutex = 1; //bu degisken ile kritik bolge erisimini
                   //kontrol edelim
int urun = 0;      //uretici her seferinde1 arttirir,tuketici
                   //bu degeri yazar

void uretici(void){
     while (TRUE){
           down(&mutex);
              urun = urun +1;
           up(&mutex);
     }
}
void tuketici(void){
     while (TRUE){
           down(&mutex);
              printf("%d\n",urun);
           up(&mutex);
     }
}
```

Mesaj Geçirme (Mesaj Passing)

Süreçler arasında mesaj gönderip, almak için ***send ve receive*** şeklinde iki sistem çağrısı bulunur. Bu çağrılar aşağıdaki gibi kütüphane fonksiyonu olarak tanımlansınlar.

send (varış,&mesaj); //varış ile verilmiş olan sürece mesajı gönderir

receive(kaynak,&mesaj);//kaynakdan gelen mesajı alır. Eğer mesaj yoksa gelene kadar bloklanır ya da bir hata kodu geriye çevirir.

Mesaj Geçirme (Mesaj Passing)

Bu işlemlerde çeşitli problemler çıkabilir.

Örneğin, mesaj gidecek olan bilgisayar ağ üzerinde başka bir bilgisayarda ise mesaj ağda kaybolabilir. Bu yüzden gönderici mesaj göndermeli, alıcı mesajı aldığını belirten kabul (ackowledge) mesajı geriye göndermelidir. Eğer gönderici kabul mesajını belirli bir süre alamazsa mesajı tekrar gönderir.

Mesaj ulaşmış fakat kabul mesajı ağda kaybolmuş olabilir. Gönderici mesajı iki kez göndermiş, alıcıda mesajı iki kez almış olur. Bu problem mesaja sıra numarası verilerek çözülür, aynı numaralı iki mesaj gelirse sadece biri kabul edilir.

Bu işlemler aynı makine üzerinde olsa dahi semaforlara göre oldukça yavaştırlar. Mesajlaşma yazmaç üzerinden yapılarak hızlandırılabilinir.

Mesaj Geçirme (Mesaj Passing)

Mesajlar alıcı tarafından alınmadığında postakutusu (mailbox) adı verilen yapılarda depolanır. Depolanan mesajlar, alıcı tarafından alınır.

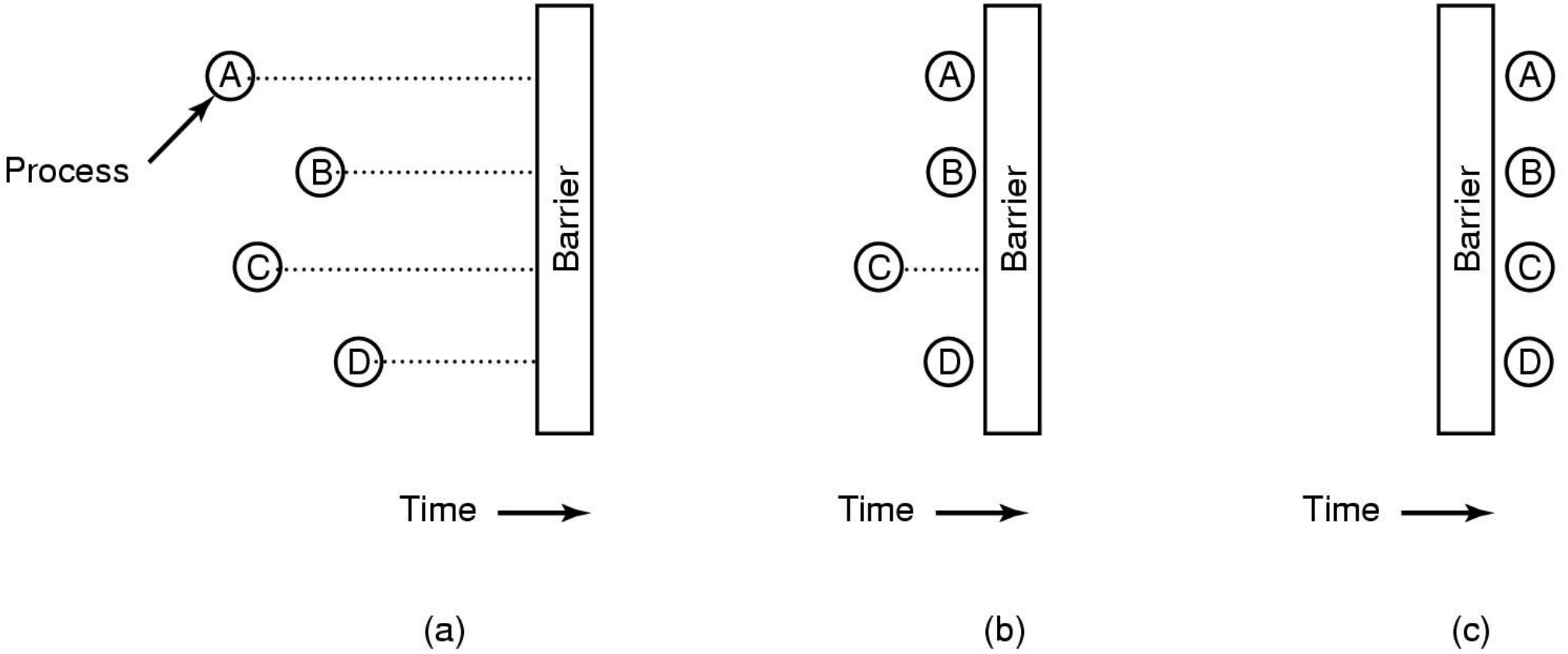
Mesajlaşmada, ilk önce gönderici mesajı göndermek için send() metodunu çalıştırır ve süreç bloklanır. receive() metodu bir alıcı tarafından çalıştırıldığı anda, mesaj tamponlama yapılmadan alıcıya geçirilir. Aynı şekilde, receive() metoduda mesaj yoksa gelene kadar bloklanır. Bu yönteme ***randevu(rendezvous)*** denilir.

Bariyerler (Barriers)

Bir grup işin yaptıkları işlerde aşamadan aşamaya geçerken birbirlerini beklemelerini sağlar. Bir süreç bariyeri geçince bloklanır. Bariyeri geçen her süreç bloklanır. Son süreç de geçtiğinde hepsi tekrar çalışırlar.

Örneğin; bir milyon x bir milyon boyutundaki matrisi çok işlemcili bir sistemde belirli bir işe tabi tutulsun. n+1. adım n. adım bitmeden gerçekleşmemelidir.

Bariyerler (Barriers)



İşletim Sistemlerine Giriş

Süreçler Arası İletişim
(IPC-Inter Process Communication)