4 Prosesler Arası Haberleşme ve Senkronizasyon

Eş Zamanlılık

- Eş zamanlı prosesler olması durumunda bazı tasarım konuları önem kazanır:
  - Prosesler arası haberleşme
  - Kaynak paylaşımı
  - Birden fazla prosesin senkronizasyonu
  - İşlemci zamanı ataması

Sorunlar

Çoklu programlı ve tek işlemcili bir sistemde bir prosesin çalışma hızı öngörülemez:

- Diğer proseslerin yaptıklarına bağlıdır.
- İşletim sisteminin kesmeleri nasıl ele aldığına bağlıdır.
- İşletim sisteminin iş sıralama yaklaşımına bağlıdır.

Sorunlar

- Eş zamanlı çalışan prosesler olması durumunda dikkat edilmesi gereken noktalar:
  - Kaynakların paylaşımı (ortak kullanım)
  - Senkronizasyon

Örnek

- Çoklu programlama, tek işlemci
- **pd** paylaşılan değişken

```
isle()
  begin
    pd = oku();
    pd = pd + 1;
    yazdir(pd);
 end
```


Çözüm

Paylaşılan kaynaklara kontrollü erişim.

Proseslerin Etkileşimi

- Prosesler birbirinden habersizdir.
  - rekabet
- Proseslerin dolaylı olarak birbirlerinden haberleri vardır.
  - Paylaşma yoluyla işbirliği
- Proseslerin doğrudan birbirlerinden haberi vardır.
  - Haberleşme yoluyla işbirliği

Prosesler Arası Rekabet

- Birbirinden habersiz proseslerin aynı kaynağı (örneğin bellek, işlemci zamanı) kullanma istekleri
  - işletim sistemi kullanımı düzenlemeli
- Bir prosesin sonuçları diğerlerinden bağımsız olmalı
- Prosesin çalışma süresi etkilenebilir.

Prosesler Arası Rekabet

- Karşılıklı dışlama
  - Kritik bölge
    - Program kodunun, paylaşılan kaynaklar üzerinde işlem yapılan kısmı.
    - Belirli bir anda sadece tek bir proses kritik bölgesindeki kodu yürütebilir.
- Ölümcül kilitlenme (deadlock)
- Yarış (race)
- Açlık (starvation)

Karşılıklı Dışlama

```
P1()                          P2()
begin                         begin
  <KB olmayan kod>              <KB olmayan kod>
  gir_KB;                       gir_KB;
     <KB işlemleri>                <KB işlemleri>
  cik_KB;                       cik_KB;
  <KB olmayan kod>              <KB olmayan kod>
end                           end
```

- KB: **K**ritik **B**ölge
- İkiden fazla proses de aynı kaynaklar üzerinde çalışıyor olabilir.

Ölümcül Kilitlenme

- Aynı kaynakları kullanan prosesler
- Birinin istediği kaynağı bir diğeri tutuyor ve bırakmıyor
- Proseslerin hiç biri ilerleyemez

  ⇒ ölümcül kilitlenme

PA	PB
al(k1); al(k2); ⇐ k2'yi bekler ….	al(k2); al(k1); ⇐ k1'i bekler ….

Yarış

- Aynı ortak verilere erişen prosesler
- Sonuç, proseslerin çalışma hızına ve sıralarına bağlı
- Farklı çalışmalarda farklı sonuçlar üretilebilir

  ⇒ yarış durumu

Yarış

Örnek:

```
k=k+1      makina dilinde:     yükle Acc,k
                               artir Acc
                               yaz Acc,k
P1                                          P2
…                                           ...
while(TRUE)                                 while(TRUE)
  k=k+1;                                      k=k+1;
…                                           …
```

Not: k'nın başlangıç değeri 0 olsun. Ne tür farklı çalışmalar olabilir? Neden?

Açlık

- Aynı kaynakları kullanan prosesler
- Bazı proseslerin bekledikleri kaynaklara hiç erişememe durumu
- Bekleyen prosesler sonsuz beklemeye girebilir

  $\Rightarrow$ açlık

Prosesler Arasında Paylaşma Yoluyla İşbirliği

- Paylaşılan değişken / dosya / veri tabanı
  - prosesler birbirlerinin ürettiği verileri kullanabilir
- Karşılıklı dışlama gerekli
- Senkronizasyon gerekebilir
- Sorunlar:
  - ölümcül kilitlenme,
  - yarış
  - açlık

Prosesler Arasında Paylaşma Yoluyla İşbirliği

- İki tür erişim:
  - yazma
  - okuma
- Yazmada karşılıklı dışlama olmalı
- Okuma için karşılıklı dışlama gereksiz
- Veri tutarlılığı sağlanması amacıyla,
  - kritik bölgeler var
  - senkronizasyon

Senkronizasyon

- Proseslerin yürütülme sıraları önceden kestirilemez
- Proseslerin üretecekleri sonuçlar çalışma sıralarına bağlı olmamalıdır
- **Örnek:** Bir P1 prosesi bir P2 prosesinin ürettiği bir sonucu kullanıp işlem yapacaksa, P2'nin işini bitirip sonucunu üretmesini beklemeli

Prosesler Arasında Paylaşma Yoluyla İşbirliği

Örnek: a=b korunacak, başta a=1, b=1

P1:	a=a+1; b=b+1;	a=a+1; b=2*b;
P2:	b=2*b; a=2*a;	b=b+1; a=2*a;

- Sıralı çalışırsa sonuçta a=4 ve b=4 √
- Bu sırayla çalışırsa sonuçta a=4 ve b=3 X

Prosesler Arasında Haberleşme Yoluyla İşbirliği

- Mesaj aktarımı yoluyla haberleşme
  - Karşılıklı dışlama gerekli değil
- Ölümcül kilitlenme olabilir
  - Birbirinden mesaj bekleyen prosesler
- Açlık olabilir
  - İki proses arasında mesajlaşır, üçüncü bir proses bu iki prosesten birinden mesaj bekler

Karşılıklı Dışlama İçin Gerekler

- Bir kaynağa ilişkin kritik bölgede sadece bir proses bulunabilir
- Kritik olmayan bölgesinde birdenbire sonlanan bir proses diğer prosesleri etkilememeli
- Ölümcül kilitlenme ve açlık olmamalı

Karşılıklı Dışlama İçin Gerekler

- Kullanan başka bir proses yoksa kritik bölgesine girmek isteyen proses bekletilmemelidir.
- Proses sayısı ve proseslerin bağıl hızları ile ilgili kabuller yapılmamalıdır.
- Bir proses kritik bölgesi içinde sonsuza kadar kalamamalıdır.

Çözümler

- Yazılım çözümleri
- Donanıma dayalı çözümler
- Yazılım ve donanıma dayalı çözümler

Meşgul Bekleme

- Bellek gözü erişiminde bir anda sadece tek bir prosese izin var
- Meşgul bekleme
  - Proses sürekli olarak kritik bölgesine girip giremeyeceğini kontrol eder
  - Proses kritik bölgesine girene kadar başka bir iş yapamaz, bekler.
  - Yazılım çözümleri
  - Donanım çözümleri

Meşgul Bekleme İçin Yazılım Çözümleri - 1

- Meşgul beklemede bayrak/paylaşılan değişken kullanımı
- Karşılıklı dışlamayı garanti etmez
- Her proses bayrakları kontrol edip, boş bulup, kritik bölgesine aynı anda girebilir.

Meşgul Bekleme İçin Yazılım Çözümleri - 2

- Ölümcül kilitlenme (deadlock)
- Ölümcül olmayan kilitlenme
  - proseslerin çalışma hızına göre problem sonsuza kadar devam edebilir
- Açlık (starvation)

Meşgul Bekleme İçin Donanım Çözümleri—1

- Özel makina komutları ile
- Tek bir komut çevriminde gerçekleşen komutlar
- Kesilemezler

`test_and_set` komutu

`exchange` komutu

Donanım Desteği

- Test and Set Instruction

```
boolean testset (int i) {
     if (i == 0) {
          i = 1;
          return true;
     }
     else {
          return false;
     }
}
```


Donanım Desteği

- Exchange Instruction

```
void exchange(int register,
   int memory) {
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```


Donanım Desteği

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true)
  {
    while (!testset (bolt))
      /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . . ,P(n));
}
```

(a) Test and set instruction

```
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
  int keyi;
  while (true)
  {
    keyi = 1;
    while (keyi != 0)
      exchange (keyi, bolt);
    /* critical section */;
    exchange (keyi, bolt);
    /* remainder */
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . ., P(n));
}
```

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Meşgul Bekleme İçin Donanım Çözümleri—2

- Sakıncaları
  - Meşgul bekleme olduğundan bekleyen proses de işlemci zamanı harcar
  - Bir proses kritik bölgesinden çıktığında bekleyen birden fazla proses varsa açlık durumu oluşabilir.
  - Ölümcül kilitlenme riski var

Donanım Desteği ile Karşılıklı Dışlama

- Kesmeleri kapatmak
  - Normal durumda proses kesilene veya sistem çağrısı yapana kadar çalışmaya devam eder.
  - Kesmeleri kapatmak karşılıklı dışlama sağlamış olur
  - Ancak bu yöntem işlemcinin birden fazla prosesi zaman paylaşımlı olarak çalıştırma özelliğine karışmış olur

Makina Komutları ile Karşılıklı Dışlama

- Yararları
  - İkiden fazla sayıda proses için de kolaylıkla kullanılabilir.
  - Basit.
  - Birden fazla kritik bölge olması durumunda da kullanılabilir.

Donanım + Yazılım Desteği ile Karşılıklı Dışlama: Semaforlar

- Prosesler arasında işaretleşme için semafor adı verilen özel bir değişken kullanılır.
- Semafor değişkeninin artmasını bekleyen prosesler askıya alınır.
  - signal(sem)
  - wait(sem)
- *wait* ve *signal* işlemleri kesilemez
- Bir semafor üzerinde bekleyen prosesler bir kuyrukta tutulur

Semaforlar

- Semafor tamsayı değer alabilen bir değişkendir
  - Başlangıç değeri ≥0 olabilir
  - *wait* işlemi semaforun değerini bir eksiltir.
  - *signal* işlemi semaforun değerini bir arttırır.
- Bu iki yol dışında semaforun değerini değiştirmek mümkün değildir.

Semaforlar

- Sadece 0 veya 1 değerini alabilen semaforlara **ikili semafor** adı verilir.
- Herhangi bir tamsayı değeri alabilen semaforlara **sayma semaforu** adı verilir.

Semafor

```
struct semaphore {
     int count;
     queueType queue;
}

void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0)
     {
          place this process in s.queue;
          block this process
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0)
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.3 A Definition of Semaphore Primitives

Binary Semaphore Primitives

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};

void semWaitB(binary_semaphore s)
{
     if (s.value == 1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
void semSignalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Semaforlar ile Karşılıklı Dışlama

```
semafor s = 1;
P1()
begin
  <KB olmayan kod>
  wait(s);
    <KB işlemleri>
  signal(s);
  <KB olmayan kod>
end
```

```
semafor s = 1;
P2()
begin
  <KB olmayan kod>
  wait(s);
    <KB işlemleri>
  signal(s);
  <KB olmayan kod>
end
```

- KB: **K**ritik **B**ölge
- İkiden fazla proses de aynı kaynaklar üzerinde çalışıyor olabilir.

Semaforlar ile Senkronizasyon

```
semafor s = 0;
P1()
begin
  <senkronizasyon noktası
  öncesi işlemleri>
  signal(s);
  <senkronizasyon noktası sonrası
  işlemleri>
end
```

```
semafor s = 0;
P2()
begin
  <senkronizasyon noktası öncesi
    işlemleri>
  wait(s);
    <senkronizasyon noktası sonrası
    işlemleri>
end
```

- İkiden fazla prosesin senkronizasyonu da olabilir.

Örnek: Üretici/Tüketici Problemi

- Bir veya daha fazla sayıda *üretici* ürettikleri veriyi bir tampon alana koyar
- Tek bir *tüketici* verileri birer birer bu tampon alandan alır ve kullanır
- Tampon boyu sınırsız

Örnek: Üretici/Tüketici Problemi

- Belirli bir anda sadece bir üretici veya tüketici tampon alana erişebilir
  ⇒ karşılıklı dışlama
- Hazır veri yoksa, tüketici bekler
  ⇒ senkronizasyon

Üretici / Tüketici Problemi

```
producer:
while (true) {
   /* produce item v */
   b[in] = v;
   in++;
}
```

```
consumer:
while (true) {
   while (in <= out)
        /*do  nothing */;
   w = b[out];
   out++;
   /* consume item w */
}
```

Producer/Consumer Problem

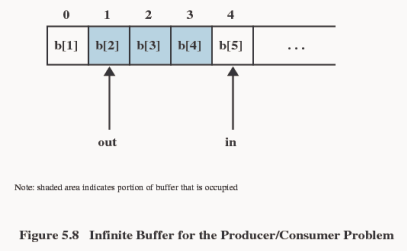

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Producer with Circular Buffer

```
producer:
while (true) {
  /* produce item v */
  while ((in + 1) % n == out)  /* do
  nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```


Consumer with Circular Buffer

```
consumer:
while (true) {
  while (in == out)
     /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  /* consume item w */
}
```


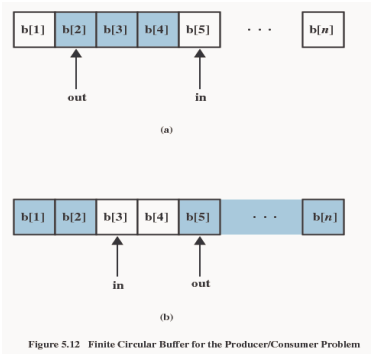

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

Üretici / Tüketici Problemi (Sonsuz Büyüklükte Tampon)

```
semafor s=1;
semafor n=0;
uretici()
begin
  while(true)
  begin
    uret();
    wait(s);
    tampona_ekle();
    signal(s);
    signal(n);
  end
end
```

```
semafor s=1;
semafor n=0;
tuketici()
begin
  while(true)
  begin
    wait(n);
    wait(s);
    tampondan_al();
    signal(s);
    tuket();
  end
end
```

Not: Birden fazla üretici prosesi çalışabilir. Tüketici prosesi tek.

Örnek: Okuyucu / Yazıcı Problemi

- Birden fazla okuyucu dosyadan okuma yapabilir.
- Bir anda sadece bir yazıcı dosyaya yazma yapabilir ⇒ karşılıklı dışlama
- Bir yazıcı dosyaya yazıyorsa okuyucu aynı anda okuyamaz ⇒ karşılıklı dışlama

LINUX'da Semafor İşlemleri

- Semafor ile ilgili tutulan bilgiler:
  - semaforun değeri
  - semaforun =0 olmasını bekleyen proses sayısı
  - semaforun değerinin artmasını bekleyen proses sayısı
  - semafor üzerinde işlem yapan son prosesin kimliği (pid)

LINUX'da Semafor İşlemleri

- Başlık dosyaları:
  - sys/ipc.h
  - sys/sem.h
  - sys/types.h
- Yaratma

```
int semget(key_t key, int nsems,int semflg);
semflag : IPC_CREAT|0700
```


LINUX'da Semafor İşlemleri

- İşlemler

```
int semop(int semid, struct sembuf *sops,
        unsigned nsops);
struct sembuf{
   ...
   unsigned short sem_num; /*numaralama 0 ile baslar*/
   short sem_op;
   short sem_flg;
};
sem_flg:       SEM_UNDO (proses sonlanınca işlemi geri al)
               IPC_NOWAIT (eksiltemeyince hata ver ve dön)
sem_op : =0    sıfır olmasını bekle (okuma hakkı olmalı)
         ≠0    değer semafor değerine eklenir (değiştirme
        hakkı olmalı)
```


LINUX'da Semafor İşlemleri

- Değer kontrolü

int semctl(int semid, int semnum,int cmd, arg);

cmd: IPC_RMID
GETVAL
SETVAL

LINUX'da Semafor İşlemleri

- Eksiltme işlemi gerçeklemesi

```
void sem_wait(int semid, int val)
{
  struct sembuf semafor;

  semafor.sem_num=0;
  semafor.sem_op=(-1*val);
  semafor.sem_flg=0;
  semop(semid, &semafor,1);
}
```


LINUX'da Semafor İşlemleri

- Artırma işlemi gerçeklemesi

```
void sem_signal(int semid, int val)
{
  struct sembuf semafor;

  semafor.sem_num=0;
  semafor.sem_op=val;
  semafor.sem_flg=0;
  semop(semid, &semafor,1);
}
```

Linux'da Semafor İşlemleri Örneği-1

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 1234

int sonsem;

void signal12(void)
{
}
```


Linux'da Semafor İşlemleri Örneği - 2

```
void sem_signal(int semid, int val){
  struct sembuf semafor;
  semafor.sem_num=0;
  semafor.sem_op=val;
  semafor.sem_flg=0;
  semop(semid, &semafor,1);
}

void sem_wait(int semid, int val)
{
  struct sembuf semafor;
  semafor.sem_num=0;
  semafor.sem_op=(-1*val);
  semafor.sem_flg=0;
  semop(semid, &semafor,1);
}
```


Linux'da Semafor İşlemleri Örneği - 3

```
int main(void)
{
  int f;

  sem=semget(SEMKEY, 1, 0700|IPC_CREAT);
  semctl(sem, 0, SETVAL,0);
  f=fork();

  if (f==-1)
  {
    printf("fork error\n");
    exit(1);
  }
```


Linux'da Semafor İşlemleri Örneği - 4

```
if (f>0) /*anne */
{
  printf("Anne calışmaya başladı…\n");
  sem_wait(sem,10);

  printf("cocuk semaforu artırdı\n");
  semctl(sem,0,IPC_RMID,0);
}
```


Linux'da Semafor İşlemleri Örneği - 5

```
else /*cocuk */
  {
    printf("  Cocuk çalışmaya başladı…\n");
    sem=semget(SEMKEY, 1,0);

    sem_signal(sem,1);
    printf("  Cocuk: semafor değeri = %d\n",
      semctl(sonsem,0,GETVAL,0));
  }
  return(0);
}
```


Linux'da Sinyal Mekanizması

- sinyaller asenkron işaretlerdir
  - işletim sistemi prosese yollayabilir
  - bir proses bir başka prosese yollayabilir
- sinyal alınınca yapılacak işler tanımlanır
- öntanımlı işleri olan sinyaller var
  - öntanımlı işler değiştirilebilir
  - SIGKILL (9 no.lu) sinyali yakalanamaz

Linux'da Sinyal Mekanizması

- Başlık dosyası:
  - sys/types.h
  - signal.h
- bir sinyal alınınca yapılacak işin bildirilmesi:

```
#typedef void (sighandler_t  *)(int);
sighandler_type signal(int signum,
                       sighandler_t sighandler);
sighandler:  SIG_IGN
             SIG_DFL
             kullanıcı tarafından tanımlanır
```


Linux'da Sinyal Mekanizması

- bir prosesten diğerine sinyal yollama:

```
int kill(pid_t pid,  int sig);
```

- bir sinyal bekleme:
  - başlık dosyası: `unistd.h`

```
int pause(void);
```


Linux'da Sinyal Mekanizması Örnek-1


```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void signal12(void)
{
    printf("12 numarali sinyali aldim. \n");
    exit(0);
}
```


Linux'da Sinyal Mekanizması Örnek - 2


```
int main (void)
{
  int f;

  signal(12, (void *)signal12);

  f=fork();
  if (f==-1)
  {
    printf("fork hatasi\n");
    exit(1);
  }
```


Linux'da Sinyal Mekanizması Örnek - 3


```
 else if (f==0)  /*cocuk*/
  {
    printf("  COCUK: basladi\n");
    pause();
  }
  else  /*anne*/
  {
    printf("ANNE: basliyorum...\n");
    sleep(3);
    printf("ANNE: cocuga sinyal yolluyorum\n");
    kill(f,12);
    exit(0);
  }
  return 0;
}
```


Linux'da Paylaşılan Bellek Mekanizması

- Birden fazla proses tarafından ortak kullanılan bellek bölgeleri
- Prosesin adres uzayına eklenir
- Başlık dosyaları:
  - sys/ipc.h
  - sys/shm.h
  - sys/types.h

Linux'da Paylaşılan Bellek Mekanizması

- Paylaşılan bellek bölgesi yaratma

```
int shmget(key_t key, int size, int shmflag);
shmflag: IPC_CREAT|0700
```

- Adres uzayına ekleme

```
void *shmat(int shmid,const *shmaddr,int shmflg);
```

- Adres uzayından çıkarma

```
int shmdt(const  void *shmaddr);
```

- Sisteme geri verme

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
cmd: IPC_RMID
```


Linux'da Paylaşılan Bellek Mekanizması

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMKEY 5678
```


Linux'da Paylaşılan Bellek Mekanizması

```
int main (void)
{
  int *pb, pbid, i;
  pbid=shmget(SHMKEY, sizeof(int),0700|IPC_CREAT);
  pb=(int *)shmat(pbid,0,0);
  *pb=0;
  for (i=0;i<10;i++)
  {
   (*pb)++;
    printf("yeni deger %d\n", (*pb));
  }
  shmdt((void *)pb);
  return 0;
}
```