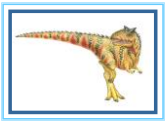
Bölüm 6: Process Senkronizasyonu



Modül 6: Process Senkronizasyonu

- Arka plan
- Kritik Bölge Problemi
- Peterson'un Çözümü
- Donanımsal Senkronizasyon
- Semaforlar
- Klasik Senkronizasyon Problemleri
- Monitörler
- Senkronizasyon Örnekleri
- Atomik İşlemler

Hedefler

- Paylaşılan verinin tutarlılığını sağlamak için kullanılabilecek Kritik Bölge Problemi çözümlerinin tanıtmak
- Kritik Bölge Problemlerine ilişkin yazılım ve donanım çözümleri sunmak
- Atomik işlem kavramını tanıtmak ve atomiklik sağlama mekanizmasını açıklamak

Arka plan

- Paylaşılan verilere eşzamanlı erişim, veri tutarsızlıklarına neden olabilir.
- Veri tutarlılığını korumak için işbirliği içindeki proseslerin düzenli yürütülmesini sağlayan bir mekanizma gerekir.
- Varsayalım ki, bir tampon kullanan üretici-tüketici problemine bir çözüm sağlamak istiyoruz. Bunun için tampon boyutunu tamsayı olarak **bir sayaç** değişkeninde tutabiliriz. Başlangıçta sayaça 0 değeri verilir. Üretici tarafından yeni bir ürün oluşturulduktan sonra sayaç değeri bir arttırılır ve tüketici bir ürünü kullandığında sayaç tüketici tarafından bir azaltılır.

Üretici

```
while (true) {

        /* bir ürün üret ve birSonrakiÜrün değerine ata */
          while (sayaç== TAMPON_BOYUTU)
                  ; // bekle
              tampon [in] = birSonrakiÜrün ;
              in = (in + 1) % TAMPON_BOYUTU;
              sayaç++;
}
```


Tüketici

```
while (true) {
        while (sayaç== 0)
              ; // bekle
              birSonrakiÜrün=  tampon [out];
               out = (out + 1) % TAMPON_BOYUTU;
                sayaç--;

                  /* birSonrakiÜrün tüketilir */
}
```


Yarış Koşulu

- sayaç++ şu şekilde uygulanabilir

```
register1 = sayaç
register1 = register1 + 1
sayaç= register1
```

- sayaç-- şu şekilde uygulanabilir

```
register2 = sayaç
register2 = register2 - 1
sayaç= register2
```

- Başlangıçta "sayaç = 5" iken aşağıdaki çalışma sırasını ele alalım:

S0: üretici register1 = sayaç satırını çalıştırır {register1 = 5}
S1: üretici register1 = register1 + 1 satırını çalıştırır {register1 = 6}
S2: tüketici register2 = sayaç satırını çalıştırır {register2 = 5}
S3: tüketici register2 = register2 - 1 satırını çalıştırır {register2 = 4}
S4: üretici counter = register1 satırını çalıştırır {sayaç = 6 }
S5: tüketici counter = register2 satırını çalıştırır {sayaç = 4}

Kritik Bölge Problemi

- n adet prosesi $\{p_0, p_1, \ldots p_{n-1}\}$ ele alalım.
- Her process içindeki kod segmenti **kritik bölgelere** ayrılmıştır.
  - Proses, ortak değişkenlerine ulaşıyor, tablo güncelliyor, dosyaya yazıyor v.b. olabilir.
  - Bir proses kritik bölgede olduğunda başka bir proses o kritik bölgeye giremez.
- Kritik bölge problemi çözmek için bir protokol gerekmektedir.
- Her process kritik bölgeye girmek için izin istemelidir- **giriş bölgesi**
- Kritik bölgeden çıkana kadar kalır - **çıkış bölgesi**
- Daha sonra geri kalan işlemlerini sürdürebilir - **kalan bölge**
- Özellikle kesintili işlemlerde uygulanır

Kritik Bölge

- p_i prosesinin genel yapısı :

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Figure 6.1 General structure of a typical process P_i.

Kritik Bölge Probleminin Çözümü

1. **Karşılıklı Dışlama (Mutual Exclusion)** – Eğer P_i prosesi kendi kritik bölgesinde çalışıyorsa,başka hiçbir proses onun kritik bölgesine giremez.
2. **İlerleme** - Eğer kritik bölgede hiçbir proses çalışmıyorsa ve kritik bölgeye girmek isteyen proses var ise mutlaka birisi seçilip ilerlemelidir.
3. **Sınırlandırılmış Bekleme (Bounded Waiting)** - Bir proses beklerken diğer proseslerin kritik bölgeye girme sayısının bir sınırı vardır.
   - Her bir proses sıfırdan farklı bir hızla çalışsın.
   - n adet prosesin **göreceli hızına** ilişkin bir varsayım yoktur.

Peterson Çözümü

- İki proses çözümü
- LOAD ve STORE talimatlarının atomik ve durdurulamaz olduğunu varsayalım.
- İki proses iki değişken paylaşsın:
  - int **turn**;
  - boolean **flag[2]**
- **turn** değişkeni kritik bölgeye girme sırasının kimde olduğunu gösterir.
- **flag** dizisi, bir prosesin kritik bölgeye girmek için hazır olup olmadığını belirtmek için kullanılır.
- **flag [i]** = true ise **P_i** prosesi hazır demektir

P_i Prosesi Algoritması

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
            kritik bölge
    flag[i] = FALSE;
            kalan bölge
} while (TRUE);
```

- Aşağıdaki şartlar sağlanmıştır:
  1. Karşılıklı dışlama korunur.
  2. İlerleme gereksinimi sağlanır.
  3. Sınırlı bekleme gereksinimi karşılanabilmektedir.

Donanım Senkronizasyonu

- Bir çok sistem kritik bölge kodu için donanım desteği sağlar.
- Tek işlemcili sistemler – kesmeler devre dışı bırakılabilir.
  - Yürütülmekte olan kodu kesinti olmadan çalıştırır.
  - Çok işlemcili sistemlerde genelde çok verimsizdir.
    - Bunu kullanan işletim sistemleri ölçeklenebilir değildir.
- Modern makineler özel atomik donanım talimatlarını destekler.
  - Atomic = kesintisiz(non-interruptable)
  - Bellek kelimesini test et veya değer ata
  - Ya da iki bellek kelimesi arasında içerik değiştir

Kilitlenme ile Kritik Bölge Probleminin Çözümü

```
do {
    kilitle
            kritik bölge
    kilidi aç
            geri kalan kısım
} while (TRUE);
```

TestAndSet Komutu

- Tanım:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

TestAndSet Kullanılarak Çözüm

- Paylaşılan boolean değişken kilitli, FALSE olarak başlatılır.
- Çözüm:

```
do {
    while ( TestAndSet (&kilit ))
            ;  // bekle

            //   kritik bölge

    kilit = FALSE;

            //     geri kalan kısım

} while (TRUE);
```

Takas(Swap) Komutu

- Tanım:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Takas Kullanılarak Çözüm

- Paylaşılan Boolean değişken kilit FALSE değerine sahiptir; Her proses , yerel Boolean türünde bir anahtar değişkenine sahiptir
- Çözüm:

```
do {
        anahtar= TRUE;
        while ( anahtar== TRUE)
                Swap (&kilit, &anahtar);

                //    kritik bölge

        kilit= FALSE;

                //      geri kalan kısım

} while (TRUE);
```

TestandSet() ile Sınırlı Beklemeli Karşılıklı Dışlama

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
            key = TestAndSet(&lock);
    waiting[i] = FALSE;
            // kritik bölge
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
            j = (j + 1) % n;
    if (j == i)
            lock = FALSE;
    else
            waiting[j] = FALSE;
            // geri kalan kısım
} while (TRUE);
```

Semafor

- Beklemeyi gerektirmeyen bir senkronizasyon aracıdır.
- *S* Semaforu – tamsayı değişken
- İki standart işlem değişiklik yapar S: wait() ve signal()
  - P() ve V(). biçiminde çağırılırlar.
- Daha az karmaşık
- Sadece iki bölünmez (atomik) işlem üzerinden erişilebilir.
  - wait (S) {
    while S <= 0
    ; // bekleme
    S--;
    }
  - signal (S) {
    S++;
    }

Genel Senkronizasyon Aracı Olarak Semafor

- **Sayma (Counting)** semaforu – tamsayı değeri kısıtlanmamış bir etki alanı içinde değişebilir.
- **İkili (Binary)** semafor – Tamsayı değeri yalnızca 0 ve 1 olarak değişebilir; uygulaması daha basit olabilir.
  - Ayrıca **mutex locks** olarak da bilinir.
- S sayma semaforu ikili semafor olarak uygulanabilir
- Karşılıklı dışlama şartını sağlar.

```
Semaphore mutex;    //  1 e kurulur
do {
    wait (mutex);
        // Krtik Bölge
    signal (mutex);
            // geri kalan bölge
} while (TRUE);
```

Semafor Uygulanması

- Herhangi iki prosesin aynı anda, aynı semafor üzerinde wait() ve signal() fonksiyonlarını çalıştıramayacağı garanti altına alınmalıdır.
- Sonra, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
- Bu nedenle, uygulamalar kritik bölge problemleri
  - Şimdi kritik bölge uygulamasında **busy waiting (meşgul bekleme)** olabilir.
    - Fakat uygulama kodu kısadır.
    - Kritik bölge nadiren meşgul olursa, kısa süreli yoğun beklemeler olur.
- Uygulamalar kritik bölgelerde çok fazla zaman harcayabilir bu nedenle bunun iyi bir çözüm olmadığını unutmayın.

Yoğun Beklemesiz Semafor Uygulanması

- Her semaforun bir bekleme kuyruğu vardır.
- Bekleme kuyruğundaki her girdi iki veri öğesine sahiptir:
  - Değer (integer tipinde)
  - listedeki sonraki kaydı gösteren işaretçi
- İki işlem :
  - **block** – Çalışmakta olan process i bekleme kuyruğuna alınmasıdır.
  - **wakeup** – bekleme kuyruğundaki bir process in hazır kuyruğuna yerleştirilmesidir.

Yoğun Beklemesiz Semafor Uygulanması (Devam)

- Bekleme uygulaması :

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Sinyal uygulaması :

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Kilitlenme ve Açlık

- **Deadlock (Kilitlenme)**– Kilitlenme iki yada daha çok görevin karşılıklı olarak hiç bir zaman gerçekleşmeyecek koşulları beklemeleriyle oluşan bir durumdur.
- Let S ve Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Starvation (Açlık)** – Belirsiz engelleme
  - Askıya alınmış bir process asla semafor kuyruğundan silinmez.
- **Priority Inversion (Öncelik Değişimi)** –Yüksek öncelikli bir processin ihtiyaç duyduğu bir kaynağı daha düşük öncelikli bir process kilitli tutuyorsa planlama problemi meydana gelir.
  - **priority-inheritance protocol (Önceliğin Kalıtımı Protokolü)** aracılığıyla çözülmüştür.

Senkronizasyonun Klasik Problemleri

- Yeni önerilen senkronizasyon planlarını sınamak için kullanılan klasik problemler.
  - Bounded-Buffer Problem (Sınırlı Tampon Problemi)
  - Readers and Writers Problem (Okuyucu ve Yazıcı Problemi)
  - Dining-Philosophers Problem (Yemek Yiyen Filozoflar Problemi)

Sınırlı Buffer (Tampon) Problemi

- *N* adet buffer'dan her biri, tek bir öge tutabilir.
- Semafor mutex, 1 değeri ile başlatılır.
- Semafor full, 0 değeri ile başlatılır.
- Semafor empty, N değeri ile başlatılır.

Sınırlı Buffer Problemi (Devam)

- Üretici process'in yapısı

```
do {
        // produce an item in nextp
      wait (empty);
      wait (mutex);
        // add the item to the buffer
      signal (mutex);
      signal (full);
} while (TRUE);
```

Sınırlı Buffer Problemi (Devam)

- Tüketici process'in yapısı

```
do {
        wait (full);
        wait (mutex);

              // remove an item from  buffer to nextc

        signal (mutex);
        signal (empty);

              // consume the item in nextc

} while (TRUE);
```

Okuyucular-Yazıcılar Problemi

- Bir veri kümesi eşzamanlı processler dizisi arasında paylaşılmıştır.
  - Okuyucular – veri kümesini yalnızca okuyabilirler; herhangi bir güncelleme yapmazlar.
  - Yazıcılar – hem okuyabilir hem yazabilir.
- Problem – aynı anda birden fazla okuyucuya izin verilir.
  - Tek bir anda yalnızca tek bir yazıcı paylaşılmış veriye ulaşabilir.
- Birkaç varyasyon ile okuyucular ve yazıcılar işlem görür – tüm öncelikler dahil.
- Paylaşılmış veri
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Okuyucular-Yazıcılar Problemi (Devam)

- Yazıcı process yapısı

```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

Okuyucular-Yazıcılar Problemi (Devam)

- Okuyucu process'in yapısı

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
              wait (wrt) ;
        signal (mutex)

              // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
              signal (wrt) ;
        signal (mutex) ;
} while (TRUE);
```

Okuyucular-Yazıcılar Probleminin Çeşitleri

- *Birinci* varyasyon – yazıcı paylaşılmış nesne üzerinde izne sahip olmadığı sürece okuyucuyu bekletme
- *İkinci* varyasyon – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations
- Problem bazı sistemlerde kernelin sağladığı okuyucu-yazıcı kilitleri ile çözülmüştür.

Yemek Yiyen Filozoflar Problemi

- Filozoflar yaşamlarını yemek yiyerek ve düşünerek geçirirler.
- Komşularıyla etkileşime geçmeden, arasıra kasesindeki yemeği yemek için 2 yemek çubuğunu almaya çalışıyor (her seferinde bir tane)
  - Yemek için ikisine de ihtiyaç duyar, işini bitirdiğinde ikisini de serbest bırakır.
- 5 filozof durumunda
  - Paylaşılmış veri
    - Bir kase pirinç (veri seti)
    - Semafor chopstick [5] initialized to 1

Yemek Yiyen Filozoflar Problem Algoritması

- Filozof *i* nin yapısı:

```
do {
        wait ( chopstick[i] );
          wait ( chopStick[ (i + 1) % 5] );

                // eat

          signal ( chopstick[i] );
          signal (chopstick[ (i + 1) % 5] );

             // think

} while (TRUE);
```

- Bu algoritmadaki problem nedir?

Semaforların Sorunları

- Semafor işlemlerinin yanlış kullanımı:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - wait (mutex) ya da signal (mutex) (ya da her ikisi de) ihmal etme
- Deadlock (Kilitlenme) ve starvation(açlık)

Monitörler

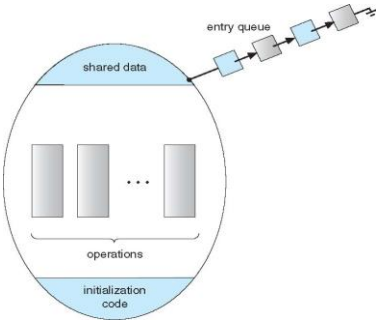
- Process senkronizasyonu için kullanışlı ve etkili bir mekanizma sağlayan yüksek seviyeli bir soyutlamadır (abstraction) .
- *Soyut (abstract) veri türleri*, dahili değişkenler sadece prosedür içindeki kod tarafından erişilebilirdirler.
- Sadece bir süreç aynı anda monitörün içinde etkin olabilir.
- Fakat bazı senkronizasyon şemalarını modellemek için yeterince güçlü değildir.

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }

   procedure Pn (…) {……}

   Initialization code (…) { … }
   }
}
```

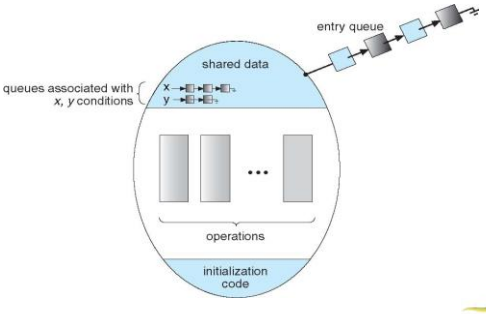

Monitor'ün Şematik Görünümü



Durum Değişkenleri

- condition x, y; //x ve y durumları
- Durum değişkenlerindeki iki işlem:
  - x.wait () –bir process x.signal() gelene kadar askıya alınır.
  - x.signal () – (varsa) x.wait () ile beklemeye alınmış process lerden biri devam ettirilir.
    - x.wait () ile beklemeye alınmış değişken yoksa, değişken üzerinde hiçbir etkisi yoktur.

Durum Değişkenleri İle Monitör



Durum Değişkenleri Seçimleri

- Eğer P process'i Q proces'i ile birlikte x.signal () çağırır ve Q x.wait () durumunda ise bir sonraki adımda ne olur?
  - Q devam durumunda ise, P beklemelidir.
- Seçenekler şunlardır :
  - **Signal and wait** – P, Q monitörü bırakmasını bekler ya da başka durumları bekler
  - **Signal and continue** – Q, P monitörü bırakmasını bekler ya da başka durumları bekler
  - Hem artıları hem eksileri vardır – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Yemek Yiyen Filozofların Çözümü

```
monitor DiningPhilosophers
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
          state[i] = THINKING;
              // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
     }
```

Yemek Yiyen Filozofların Çözümü (Devam)

```
        void test (int i) {
              if ( (state[(i + 4) % 5] != EATING) &&
              (state[i] == HUNGRY) &&
              (state[(i + 1) % 5] != EATING) ) {
                  state[i] = EATING ;
               self[i].signal () ;
               }
         }

         initialization_code() {
              for (int i = 0; i < 5; i++)
              state[i] = THINKING;
        }
    }
```

Yemek Yiyen Filozofların Çözümü (Devam)

- Her filozof aşağıdaki sırayla pickup() ve putdown() operasyonlarını çağırır :

```
DiningPhilosophers.pickup (i);

    EAT

DiningPhilosophers.putdown (i);
```

- Deadlock (kilitlenme) olmaz fakat starvation (açlık) mümkün.

Semafor Kullanarak Monitör Uygulama

- Değişkenler

```
semaphore mutex;  // (initially = 1)
semaphore next;   // (initially = 0)
int next_count = 0;
```

- Her prosedür ***F*** ile değiştirilecektir.

```
wait(mutex);
    …
        body of F;
    …
if (next_count > 0)
   signal(next)
else
   signal(mutex);
```

- Monitörün içinde mutual exclusion (karşılıklı dışlama) sağlanır.

Monitor Uygulama – Durum Değişkenleri

- Her koşul değişkeni ***x*** için şunlara sahibiz :

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- x.wait işlemi şu şekilde uygulanabilir :

```
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```

Monitor Uygulama (Devam)

- x.signal işlemi şu şekilde uygulanabilir :

```
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Bir Monitör İçindeki Processleri Sürdürme

- X durum kuyruğunda birden fazla process mevcutsa ve x.signal() çalıştırıldıysa, hangisi sürdürülmelidir?
- FCFS (first come first serve -ilk gelen ilk işlem görür) genellikle yeterli değildir.
- **conditional-wait (koşullu bekleme)** x.wait(c) biçiminde form oluşturur.
  - C, **priority number** (öncelik sayısı)dır.
  - Process with lowest number (yüksek öncelik) is scheduled next

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
            if (busy)
                x.wait(time);
            busy = TRUE;
    }
    void release() {
            busy = FALSE;
            x.signal();
    }
initialization code() {
     busy = FALSE;
    }
}
```

Senkronizasyon Örnekleri

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Senkronizasyon

- Multitasking, multithreading (gerçek zamanlı threadler de dahil olmak üzere) ve multiprocessing desteklemek için çeşitli kilitler uygular.
- Kısa kod bölümlerinde verimlilik sağlamak için **adaptive mutexes (uyarlanabilir muteksler)** kullanır.
  - standart semafor spin-lock gibi başlar.
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- **condition variables (durum değişkenleri)** kullanır
- Uzun bölümlerde kod veriye erişim ihtiyacı duyduğunda okuyucu-yazıcı kilitleri kullanır.
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows XP Senkronizasyon

- Tek işlemcili sistemlerde global kaynaklara güvenli erişim için interrupt masks (kesme maskeleri) kullanır.
- Birden fazla işlemcili sistemlerde **spinlocks** kullanır.
  - Spinlocking-thread asla preempted (işlem önceliğine sahip) olmamalıdır.
- Ayrıca kullanıcı tarafında muteksler, semaforlar, olaylar ve zamanlayıcılar gibi davranan **dispatcher objects (dağıtıcı nesneler)** sağlar.
  - **Events (olaylar)**
    - Bir olay daha çok bir durum değişkeni gibi davranır.
  - Timers (zamanlayıcılar) süre aşıldığında (time expired) bir ya da daha fazla thread a bildirir.
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Senkronizasyon

- Linux:
  - Kernel 2.6 versiyonundan önceki versiyonlarda, kritik bölgeleri tamamlamak için kesmeler devre-dışı bırakır.
  - Versiyon 2.6 ve sonrası, tamamen önleyicidir.
- Linux şunları sağlar :
  - semaforlar
  - spinlocks
  - Hem okuyucu-yazıcı sürümleri
- Tek CPU'lu sistemlerde, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Senkronizasyon

- Pthreads API, OS-independent (işletim sisteminden bağımsız)'dır.
- Şunları destekler:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Atomik İşlemler

- System Model (Sistem Modeli)
- Log-based Recovery (Günlük Tabanlı Kurtarma)
- Checkpoints (Denetim Noktaları)
- Concurrent Atomic Transactions (Eşzamanlı Atomik İşlemler)

Sistem Modeli

- İşlemlerin tek bir mantıksal birim olarak gerçekleşmesini (tamamı ya da hiçbiri)garanti eder.
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - Tek bir mantıksal fonksiyonu gerçekleştiren talimatlar ya da işlemler topluluğudur.
  - Burada sabit disk değişiklikleri ile ilgiliyiz - disk
  - Transaction, okuma ve yazma işlemleri dizisidir.
  - commit (transaction başarılı) ya da abort (transaction başarısız) tarafından sonlandırılır.
  - Aborted transaction (İptal edilen transaction) rolled back (geri alınma)işlemi ile yapılan değişiklikleri geri almalıdır.

Depolama Ortamı Türleri

- Volatile (Geçici) depolama – Burada saklanan bilgiler sistemin çökmesi durumunda kaybolur.
  - Örnek: Ana bellek (Ram), cache bellek
- Nonvolatile (Kalıcı) depolama – Bilgiler genelde korunur.
  - Örnek : disk ve tape
- Stable (Sabit) depolama – Bilgiler asla kaybolmaz.
  - Aslında mümkün değildir, bu nedenle hatalara karşı kopyalama ve RAID gibi yöntemler uygulanır.

Amaç, uçucu bellekte yaşanabilecek hatalara karşı "transaction atomicity" sağlamaktır.

Günlük Tabanlı Kurtarma

- Bir transaction tarafından yapılan sabit diske ilişkin tüm değişiklikleri kaydedin.
- En yaygını write-ahead logging'dir.
  - Sabit diskteki günlükteki her bir günlük kaydı, tek bir transaction (işlem)'in yaptığı işi kaydeder, bu kayıtlar şunları içerir :
    - İşlem Adı
    - Veri Ögesinin Adı
    - Eski Değer
    - Yeni Değer
  - $<T_i$ starts$>$ T_i işlemi başladığında günlüğe yazılır.
  - $<T_i$ commits$>$ T_i işlemi gerçekleştiğinde günlüğe yazılır.
- Veriler üzerinde işlem gerçekleşmeden önce günlük kaydının sabit diske ulaşması gerekir.

Günlük Tabanlı Kurtarma Algoritması

- Sistem, günlüğü kullanarak geçici bellekteki herhangi bir hatayı işleyebilir.
  - Undo(T_i) T_i tarafından güncellenen tüm verilerin değerlerini geri yükler.
  - Redo(T_i) Tüm verilerin değerlerini T_i işleminden gelen yeni değerler ile değiştirir.
- Undo(T_i) ve redo(T_i) idempotent (eş kuvvetli) olmalıdır.
  - bir execution (çalıştırma)ile birden fazla execution ın sonucun aynı olması.
- Eğer sistem hatası olursa tüm veriler günlük aracılığıyla geri yüklenir.
  - Eğer günlük $<T_i$ commits$>$ olmadan $<T_i$ starts$>$ içeriyorsa, undo(T_i)
  - Eğer günlük $<T_i$ starts$>$ ve $<T_i$ commits$>$ içeriyorsa, redo(T_i)

Checkpoints (Denetim Noktaları)

- Günlük dosyaları ve recovery (kurtarma) süreleri uzayabilir.
- Denetim noktaları günlük ve kurtarma süresini kısaltır.
- Denetim noktası şeması:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record <checkpoint> to the log on stable storage
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti All other transactions already on stable storage

Eşzamanlı İşlemler

- Serial execution (dizisel yürütme)'a eşdeğer olmalıdır. – serializability (serileştirilebilirlik)
- Kritik bölgedeki tüm işlemlerinizi gerçekleştirebilir.
  - Çok kısıtlayıcı, verimsiz.
- Concurrency-control algorithms (eşzamanlı kontrol algoritmaları) serializability sağlar.

Serializability (Serileştirilebilirlik)

- A ve B isimli iki veri öğesi düşünün.
- T_0 ve T_1 isimli iki transaction düşünün.
- T_0, T_1'i atomically (atomik) çalıştırılsın.
- Uygulamalar dizisi (Execution sequence) schedule olarak adlandırılır.
- Atomik olarak çalıştırılan işlemler dizisine ise serial schedule (dizisel planlama) denir.
- N adet işlem için, N! tane geçerli serial schedule vardır.

Plan 1: Önce T_0 , Sonra T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule (Dizisel Olmayan Planlama)

- Nonserial schedule (dizisel olmayan planlama) örtüşmeli çalıştırmaya (overlapped execute) olanak sağlar.
  - Resulting execution not necessarily incorrect
- S bir planlama ve O_i, O_j birer işlem olsun.
  - Conflict (çatışma) if access same data item, with at least one write
- If O_i, O_j consecutive and operations of different transactions & O_i and O_j don't conflict
  - Then S' with swapped order O_j O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
  - S is conflict serializable

Plan 2: Concurrent Serializable Schedule (Eşzamanlı Serileştirilebilir Planlama)

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Kilitleme Protokolü (Locking Protokol)

- Ensure serializability by associating lock with each data item
  - Erişim kontrolü için kilitleme protokolü izlenir.
- Locks (Kilitler)
  - Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
  - Exclusive – Ti has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- Eğer lock already held, yeni istek beklemek zorunda kalabilir.
  - Okuyucular-yazıcılar algoritmasıyla benzerdirler.

Two-phase Locking Protocol İki aşamalı Kilitleme Protokolü

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - Growing – obtaining locks
  - Shrinking – releasing locks
- Deadlock (kilitlenme)'yi önlemez.

Timestamp-Tabanlı Protokoller

- Select order among transactions in advance – timestamp-ordering
- Transaction T_i associated with timestamp TS(T_i) before T_i starts
  - TS(T_i) < TS(T_j) if Ti entered system before T_j
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If TS(T_i) < TS(T_j), system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j

Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order
- Suppose Ti executes read(Q)
  - If $TS(T_i) < W\text{-timestamp}(Q)$, Ti needs to read value of Q that was already overwritten
    - read operation rejected and T_i rolled back
  - If $TS(T_i) \geq W\text{-timestamp}(Q)$
    - read executed, R-timestamp(Q) set to max(R-timestamp(Q), $TS(T_i)$)

Timestamp-ordering Protokolü

- Suppose Ti executes write(Q)
  - If $TS(T_i) < R\text{-timestamp}(Q)$, value Q produced by T_i was needed previously and T_i assumed it would never be produced
    - Write operation rejected, T_i rolled back
  - If $TS(T_i) < W\text{-timestamp}(Q)$, T_i attempting to write obsolete value of Q
    - Write operation rejected and T_i rolled back
  - Otherwise, write executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

Timestamp Protokolü Altında Olası Planlama

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Bölüm 6 Sonu