İşletim Sistemleri
Process Senkronizasyonu

Dr. Öğr. Üyesi Ertan Bütün

Bu dersin içeriği hazırlanırken Operating System Concepts (Silberschatz, Galvin and Gagne) kitabı ve
Prof. Dr. M. Ali Akcayol'un (Gazi Üniversitesi Bilgisayar Mühendisliği Bölümü) ders sunumlarından faydalanılmıştır.

https://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/index.html
http://w3.gazi.edu.tr/~akcayol/BMOS.htm

Konular

- Process Synchronization
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Classical Problems of Synchronization

- Bu bölümde örnek olarak birkaç senkronizasyon problemi verilmiştir. Bu problemler, neredeyse önerilen her yeni senkronizasyon yöntemini test etmek için kullanılan problemlerdir.
- Bu bölümde verilen problemlerin senkronizasyon çözümü için semaforlar kullanılmıştır.
  - Dining-Philosophers Problem
  - Bounded-Buffer Problem
  - Readers and Writers Problem

Dining-Philosophers Problem

- Hayatlarını düşünerek ve yemek yiyerek geçiren beş filozof var. Masanın ortasında bir kase pirinç bulunur ve masa beş adet tek çubukla döşenir.
- Zaman zaman bir filozof acıkır ve kendisine en yakın olan iki yemek çubuğunu almaya çalışır. Bir filozof bir seferde yalnızca bir yemek çubuğu alabilir.
  - Filozofun yemek yemek için iki çubuğa ihtiyacı vardır, yedikten sonra ikisini de bırakır.
- Herkese yetecek çubuk olmadığından filozofların hepsi aynı anda yemek yiyemez. Filozoflar açlıktan ölmeden yemek yiyebilirler mi, yerlerse nasıl?
  - Bu problem, çeşitli kaynakların çeşitli processler arasında deadlock ve starvation olmadan bir şekilde tahsis edilmesi ihtiyacının basit bir temsilidir.

Dining-Philosophers Problem Solution

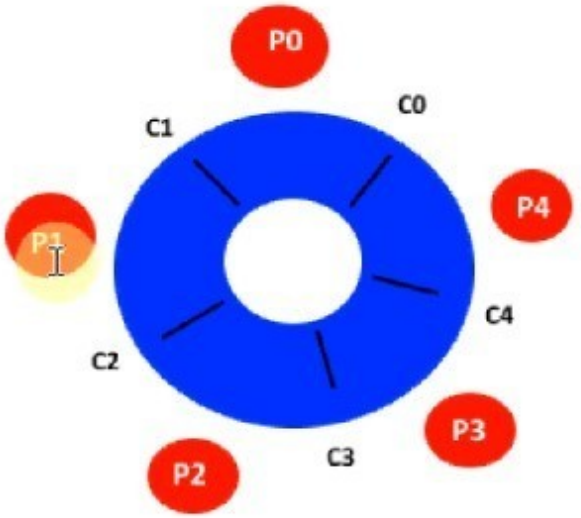
- *i* filozofu için çözümün algoritması:

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

                //  eat

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

                 //  think

} while (TRUE);
```

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{
  think...
wait(chopstick[i]);
wait(chopstick[(i+1)mod 5]);
 ....
 eat
 ....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```



Pi	cp[i]	cp[i+1 mod 5]

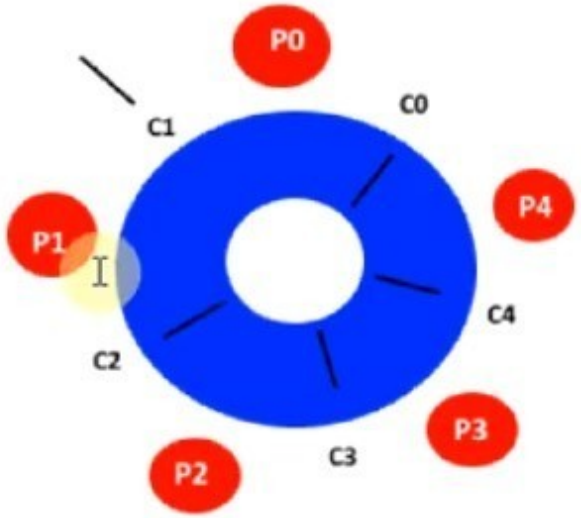
chopstick[i]

0	1	2	3	4
1	1	1	1	1

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{
  think...
wait(chopstick[i]);
wait(chopstick[(i+1)mod 5]);
....
eat
....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```



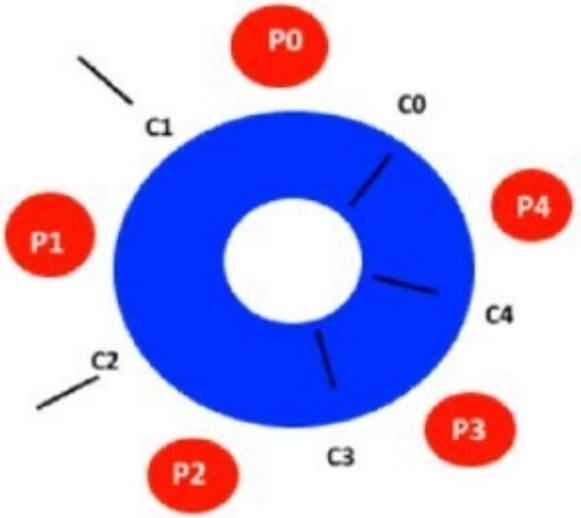
Pi	cp[i]	cp[i+1 mod 5]

chopstick[i]	0	1	2	3	4
	1	0	1	1	1

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{
  think...
wait(chopstick[i]);
wait(chopstick[(i+1)mod 5]);
....
eat
....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```



Pi	cp[i]	cp[i+1 mod 5]

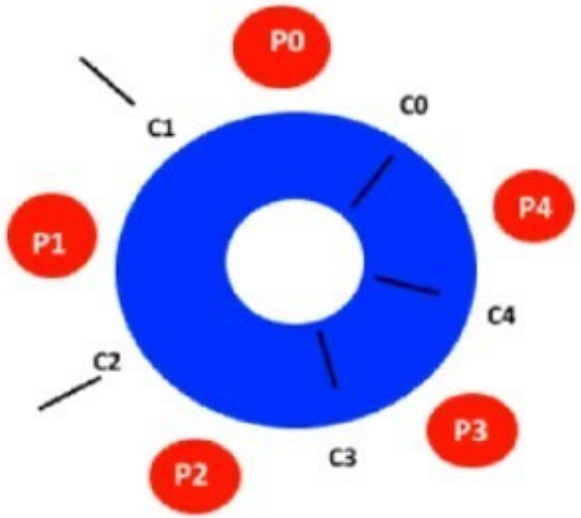
chopstick[i]	0	1	2	3	4
	1	0	0	1	1

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{
  think...
wait(chopstick[i]);
wait(chopstick[(i+1)mod 5]);
....
eat        P1
....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```

Pi	cp[i]	cp[i+1 mod 5]

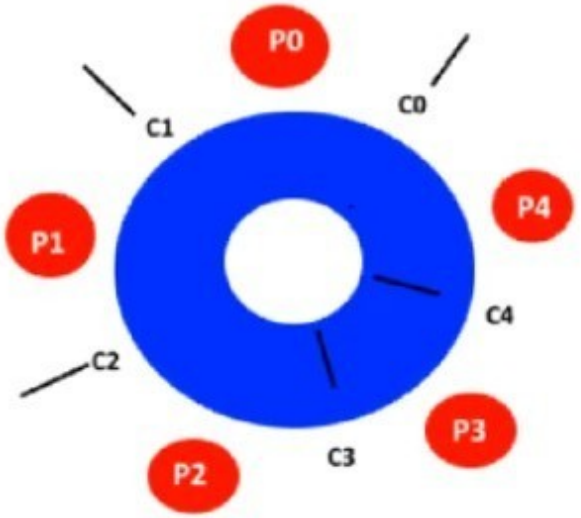


chopstick[i]	0	1	2	3	4
	1	0	0	1	1

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{

wait(chopstick[i]);
wait(chopstick[(i+1)mod 5]);
....
eat                P1
....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```

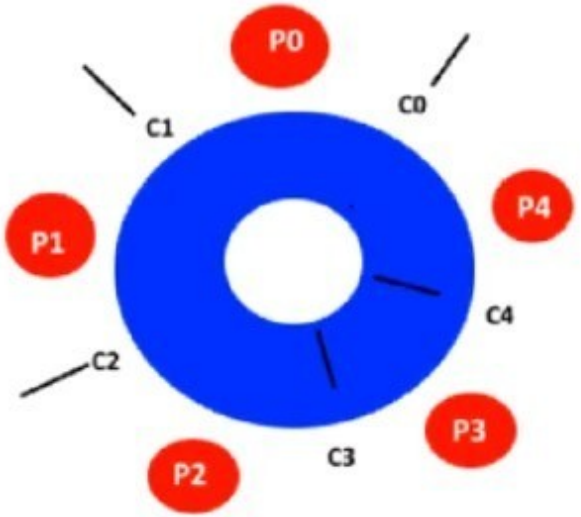


chopstick[i]	0	1	2	3	4
	0	0	0	1	1

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{

wait(chopstick[i]);                P0
wait(chopstick[(i+1)mod 5]);
....
eat                    P1
....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```

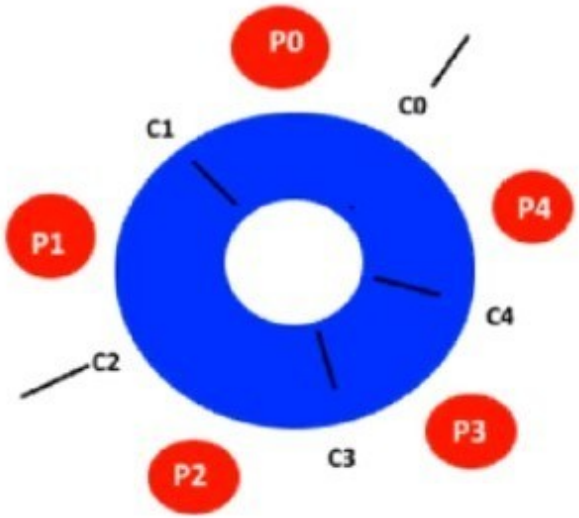


chopstick[i]	0	1	2	3	4
	0	0	0	1	1

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{

wait(chopstick[i]);                P0
wait(chopstick[(i+1)mod 5]);
....
eat
....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```

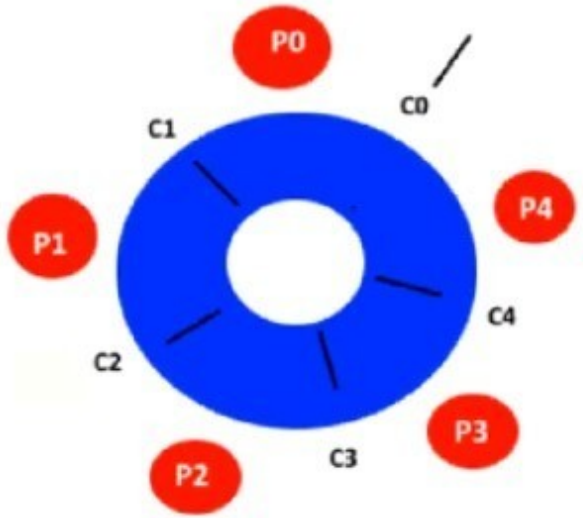


chopstick[i]	0	1	2	3	4
	0	1	0	1	1

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{

wait(chopstick[i]);                P0
wait(chopstick[(i+1)mod 5]);
....
eat
....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```



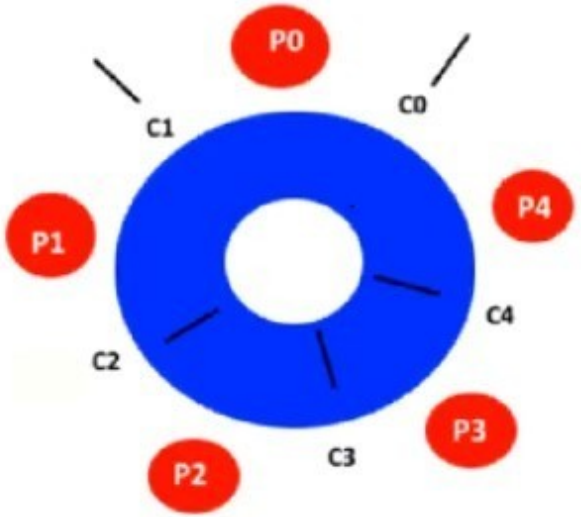
chopstick[i]	0	1	2	3	4
	0	1	1	1	1

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{

wait(chopstick[i]);
wait(chopstick[(i+1)mod 5]);
....
eat
....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```

P0

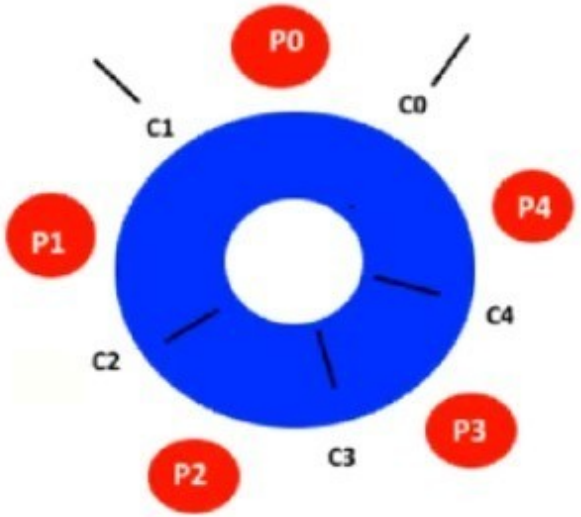


chopstick[i]	0	1	2	3	4
	0	0	1	1	1

Example-Dining-Philosophers Problem Solution

- Örnek çalıştırma

```
semaphore chopchick[5];

do
{

wait(chopstick[i]);
wait(chopstick[(i+1)mod 5]);
....
eat     P0
....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
...
think
}while(1);
```

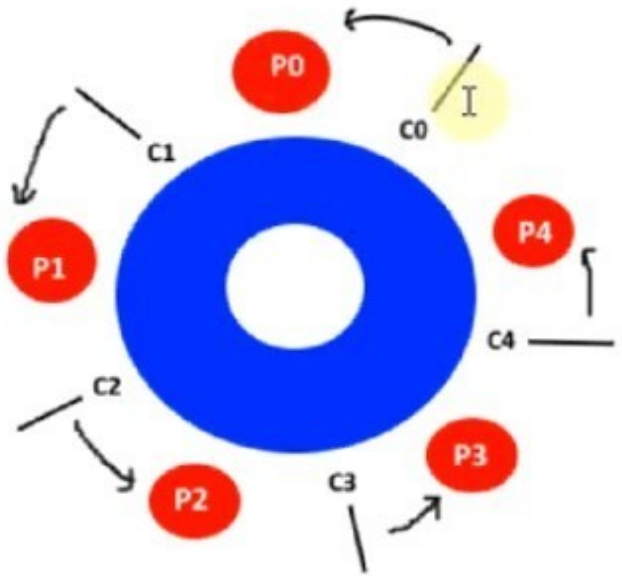


chopstick[i]	0	1	2	3	4
	0	0	1	1	1

Dining-Philosophers Problem Solution

- Bu çözüm, iki komşunun aynı anda yemek yememesini garanti etse de, deadlock problemi yaratabilir. *

```
semaphore chopchick[5];

do
{
  think...
wait(chopstick[i]);
wait(chopstick[(i+1)mod 5]);
 ....
eat
 ....
signal(chopstick[i]);
signal(chopstick[(i+1)mod 5]);
 ...
think
}while(1);
```



Dining-Philosophers Problem Solution

- Deadlock sorununun birkaç olası çözümü şu şekilde olabilir:
  - En fazla dört filozofun aynı anda masada oturmasına izin verilebilir.
  - Bir filozofun yemek çubuklarını yalnızca her iki yemek çubuğu da varsa almasına izin verilebilir (bunu yapmak için, onları kritik bir bölümden alması gerekir).
  - Asimetrik bir çözüm kullanılabilir - yani, tek sayılı bir filozof önce sol sonra da sağ yemek çubuğunu alırken, çift sayılı bir filozof önce sağ sonra da sol yemek çubuğunu alır.

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
       ...
    /* remove an item from buffer to next_consumed */
       ...
    signal(mutex);
    signal(empty);
       ...
    /* consume the item in next consumed */
       ...
 } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait(rw_mutex);
           ...
        /* writing is performed */
           ...
     signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        ...
        /* reading is performed */
        ...
    wait(mutex);
        read count--;
        if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Problems with Semaphores

- Semafor işlemlerinin yanlış kullanımı hatalara neden olabilir:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - wait (mutex) ya da signal (mutex) (ya da her ikisini) unutmak
- Deadlock ya da starvation olasıdır.

Konular

- Process Synchronization
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Monitors

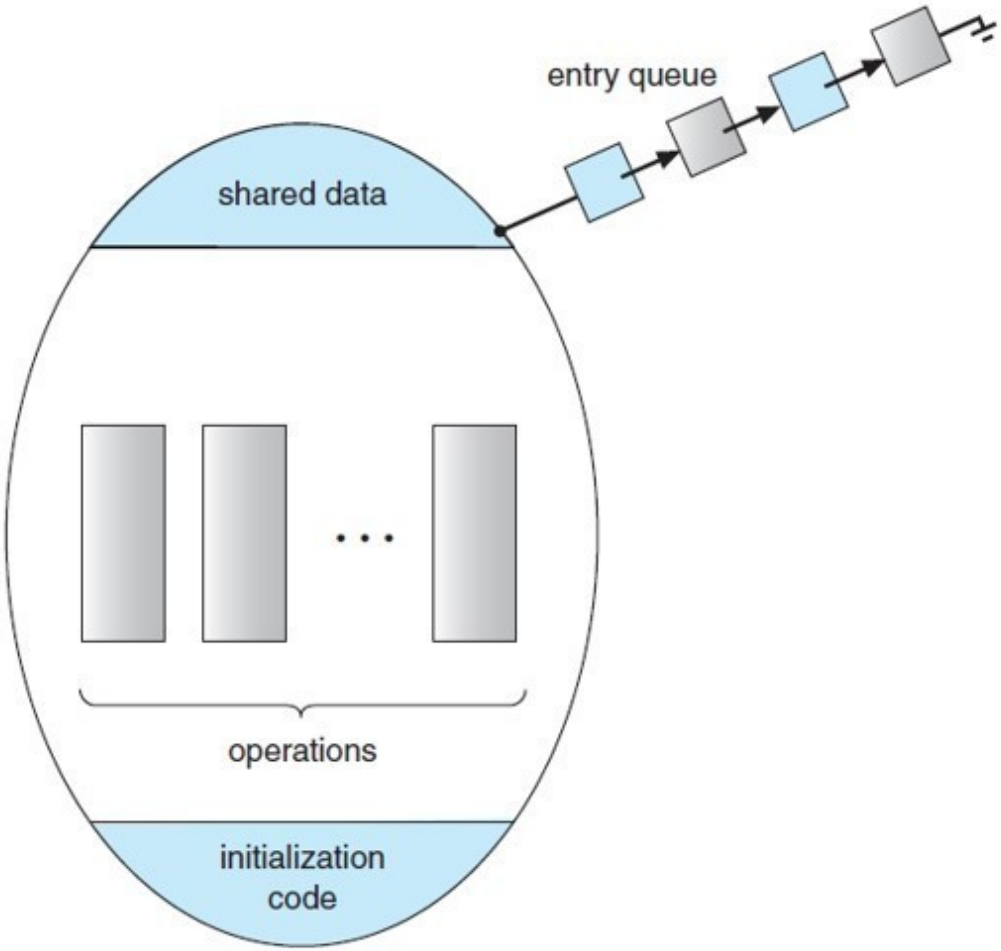
- Semaforların kullanımıyla ilgili oluşabilecek hatalarla başa çıkmak için yüksek-seviye dil yapıları geliştirilmiştir, bunlardan biri de monitor'lerdir.
- Montior process senkronizasyonu için uygun ve etkili bir mekanizma sağlayan yüksek seviye bir soyutlamadır.
- **Monitör yapısı, monitörde aynı anda yalnızca bir işlemin etkin olmasını garanti eder. Dolayısıyla, programcının senkronizasyon kısıtlamasını açıkça kodlaması gerekmez.**

Monitors

- Monitor türünün genel yapısı şekilde verilmiştir.
- Soyut bir veri türü (abstract data type) olan **monitor** türü, mutual exclusion'ın sağlandığı programcıların tanımladığı bir dizi işlemlerden oluşur.
- Sadece monitor içinde tanımlanan bir fonksiyon, monitor içinde tanımlanan değişkenlere ve kendi parametrelerine erişebilir.

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Bir Monitor'ün şematik görünümü



Monitors

- Şu ana kadar tanımlanan monitör yapısı, bazı senkronizasyon şemalarını modellemek için yeterince güçlü değildir.
- Bu amaçla, ek senkronizasyon mekanizmaları tanımlanması gerekir. Bu mekanizmalar, durum yapıları (**condition construct)** tarafından sağlanır.
- Özel olarak hazırlanmış bir senkronizasyon şeması yazması gereken bir programcı, bir veya daha fazla **durum(condition) değişkenini** tanımlayabilir:
  - `condition x, y;`

Monitors

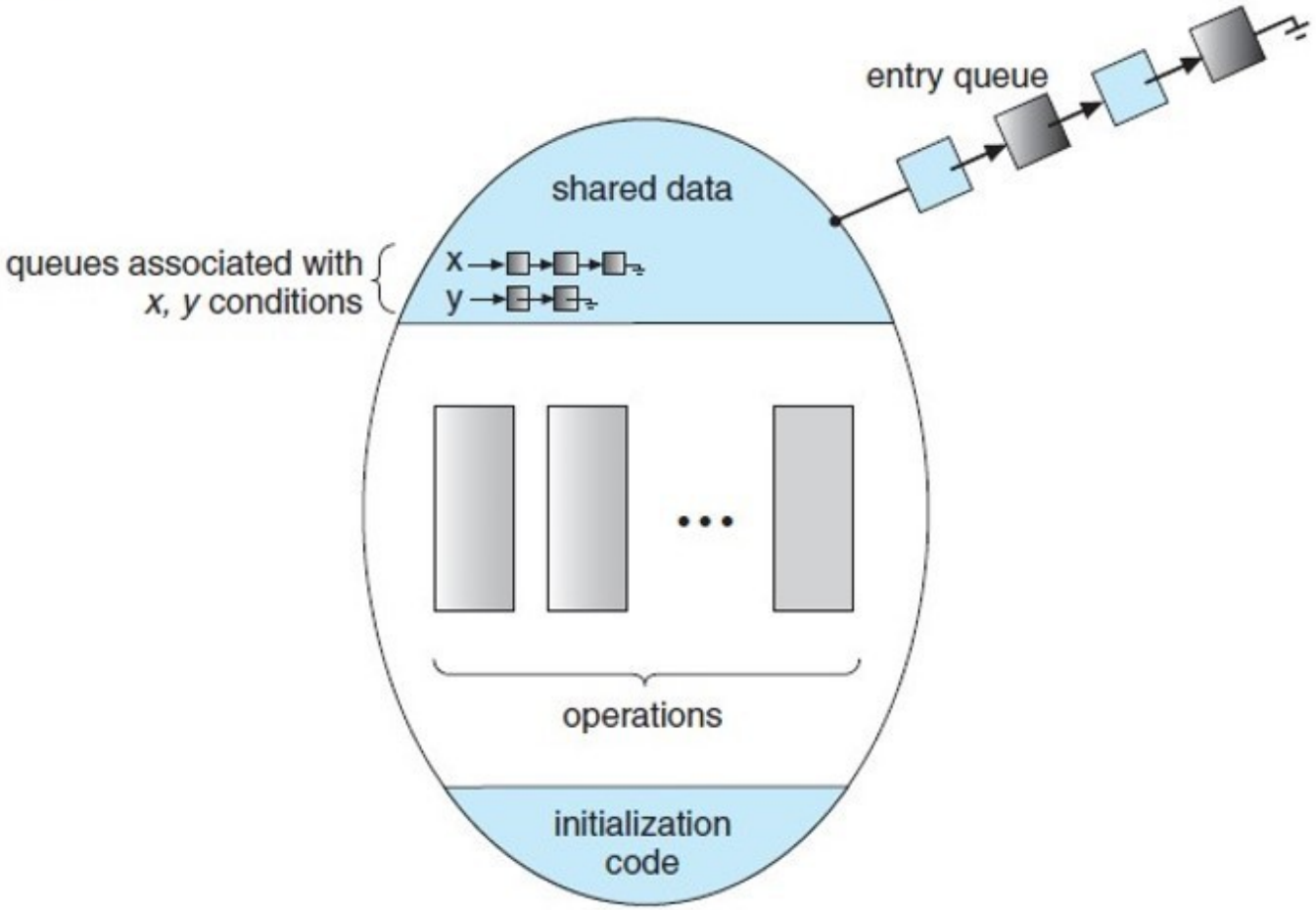
- Sadece durum değişkeni üzerinden çağrılabilen **`wait()`** ve **`signal()`** işlemleri tanımlanabilir:
  - **`x.wait();`**

  `wait()'i` çağıran process, başka bir process **`x.signal()'i`** çalıştırıncaya kadar beklemeye alınır.
  - **`x.signal();`**

  ile **`x condition`** değişkenine bağlı beklemekte olan bir process çalışmaya devam eder.

Monitors

- **x** ve **y** durum değişkenlerine bağlı process'lerin monitör içinde çalışması.



Monitors

- **`x.signal()`** işlemini bir **`P`** process'i başlatmış olsun. Aynı anda, `x` durumuna bağlı beklemekte olan bir **`Q`** process'i olsun.
- Hem Q hem de P paralel olarak çalışamaz. Q devam ettirilirse, P beklemelidir.
- Bu durumda iki olasılık vardır:
  - Signal and wait: P process'i, Q process'inin monitör'den ayrılmasını veya başka bir duruma geçmesini bekler.
  - Signal and continue: Q process'i, P process'inin monitör'den ayrılmasını veya başka bir duruma geçmesini bekler.*

Monitor Solution to Dining Philosophers

```
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((i + 4) % 5);
      test((i + 1) % 5);
   }

   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;
    }
}
```

```
DiningPhilosophers.pickup(i);
                ...
                eat
                ...
DiningPhilosophers.putdown(i);
```

Monitor Implementation Using Semaphores

- Değişkenler

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

- her bir F fonksiyonu şöyle değiştirilir:

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

- Böylece monitor içinde mutual exclusion sağlanır.

Monitor Implementation Using Semaphores

- Her condition **x** değişkeni için için **x_sem** semaforu ve **x_count** tamsayı değişkeni tanımlanır: (başlangıç değerleri 0)

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

- **x.wait()** işlemi şöyle gerçekleştirilebilir:

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

Monitor Implementation Using Semaphores

- **`x.signal()`** işlemi şöyle gerçekleştirilebilir:

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

Resuming Processes within a Monitor

- **x** condition için birkaç process askıya alınırsa ve bir process tarafından **x.signal ()** işlemi yürütülürse, askıya alınan işlemlerden hangisi daha sonra devam ettirilecek?
- Basit bir çözüm, ilk gelene ilk hizmet (FCFS) sıralaması kullanmaktır, böylece en uzun süredir bekleyen process önce devam ettirilir.
- Ancak birçok durumda, bu kadar basit bir programlama şeması yeterli değildir. Bu amaçla, conditional-wait yapısı kullanılabilir:
  - `x.wait(c);`
  - `c`, priority number olarak adlandırılan öncelik numarasıdır.
  - En küçük öncelik numarasına (yüksek öncelik) sahip process, bir sonraki adımda devam ettirilir.

Monitor Example Using Priority Number - Single Resource allocation

- conditional-wait yapısı kullanılarak rekabet içindeki processler arasında tek bir kaynağın tahsisini kontrolü eden bir resource allocator monitörü örneği
- Söz konusu kaynağa erişmesi gereken bir process aşağıdaki sırayı izlemelidir:

```
R.acquire(t);
    ...
    kaynağa erişim
    ...
R.release();
//R, ResourceAllacator türünün bir örneğidir
```

```
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;
   }
}
```

Konular

- Process Synchronization
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - 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
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- 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 Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Konular

- Process Synchronization
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Alternative Approaches

Transactional memory

- **Multicore sistemlerde**, **mutex lock**, **semafor** gibi mekanizmalarda **deadlock** gibi problemlerin **oluşma riski bulunmaktadır.**
- Bunun yanı sıra, **thread sayısı arttıkça deadlock problemlerinin ortaya çıkma olasılığı artmaktadır.**
- **Klasik mutex lock (veya semafor)** kullanılarak paylaşılmış veride güncelleme yapan **update()** fonksiyonu aşağıdaki gibi yazılabilir.

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

Alternative Approaches

Transactional memory

- Klasik kilitleme yöntemlerine alternatif olarak **programlama dillerine yeni özellikler eklenmiştir.**
- Örneğin, **atomic(S)** kullanılarak **S işlemlerinin tümünün transaction olarak gerçekleştirilmesi sağlanır.**

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```

- **Lock işlemine gerek kalmadan** ve **kilitlenme olmadan işlem tamamlanır.**

Alternative Approaches

OpenMP (Open Multi-Processing)

- OpenMP, C, C++ ve Fortran için compiler direktiflerinden oluşan API'dir.
- OpenMP, paylaşılmış hafızada eşzamanlı çalışmayı destekler.
- OpenMP, **`#pragma omp critical`** komutu ile critical sectionı belirler ve aynı anda sadece bir thread çalışmasına izin verir.

```
void update(int value)
{
    #pragma omp critical
    {
        counter += value;
    }
}
```