KONKURENTNO PROGRAMIRANJE

Konkurentni programi

- Sadrže više izvršnih delova koji mogu istovremeno biti aktivni

- Konkurentnost može biti:
  - Fizička – izvršni delovi se izvršavaju paralelno na različitim procesorima
  - Logička – svi aktivni izvršni delovi se izvršavaju na istom procesoru u *time sharing*-u

Nivoi konkurentnosti

- **Nivo instrukcija** – različite mašinske instrukcije se izvršavaju istovremeno
- **Nivo naredbi** – naredbe višeg programskog jezika se izvršavaju istovremeno
- **Nivo potptograma** – različiti potprogrami se izvšavaju istovremeno (korutine, procesi)
- **Nivo programa** – više programa se izvršava istovremeno

Razvoj višeprocesorskih sistema

- 1950-te godine – Jedan glavni procesor i jedan ili više specijalizovanih procesora za ulazmo-izlazne operacije
- Rane 1960-te – više kompletnih procesora - konkurentnost na nivou programa
- Srednje 1960-te – više parcijalnih procesora (sa ograničenim skupom instrukcija) – konkurentnost na nivou instrukcia
- SIMD arhitekture – jedan procesor ima dekoder instrukcija, ostali samo ALU, svi u jednom trenutku izvršavaju istu instrukciju (SIMD – Single Instruction Multiple Data)
- MIMD arhitekture – više procesora sa zajedničkom (deljivom) memorijom pri čemu svaki procesor izvršava svoj skup instrukcija (MIMD – Multiple Instructions Multiple Data)

Paralelizam na nivou naredbi

Konkurentno izvršavanje proizvoljnog skupa naredbi

```
cobegin/parbegin
    stmt_1;
    stmt_2;
    ...
    stmt_n;
coend/parend
```

Paralelne petlje

- Matlab:

```
parfor loopVar=initVal:endVal;
   statements;
end
```

- Fortran 90:

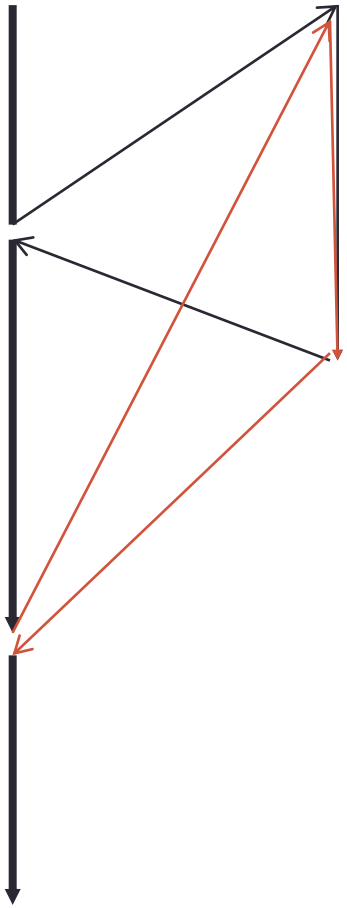
```
forall ( parameters )
    statements

FORALL (i=1:n, j=1:n)
    A(i,j)=i+j
```

Konkurentnost na nivou potprograma

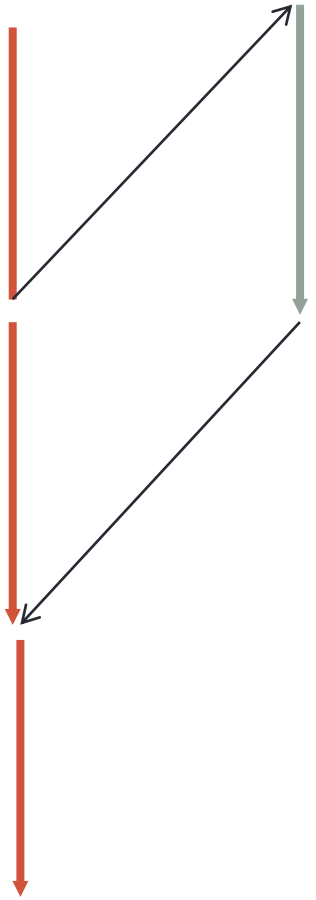
- Proces, nit (thread) ili task je deo programa koji se može izvršavati konkurentno sa drugim procesima.
- Razlika između potprograma i procesa:
  - Proces se mora eksplicitno startovati
  - Kada neki programski modul pokrene novi task, njegovo izvršenje se ne prekida – nastavljaju da se izvršavaju „zajedno“
  - Kada se izvršenje procesa završi, upravljanje ne mora da se vrati modulu koji ga je kreirao i pokrenuo

Razlika između potprograma i procesa

POTPROGRAM



PROCES



Vrste sinhronizacije

- Sinhronizacija pristupa zajedničkim resursima - međusobno isključivanje procesa kod pristupa zajedničkim resursima.

- Sinhronizacija procesa, tj. sinhronizacija komunikacije.

Metode za sinhronizaciju

- Za sinhronizaciju pristupa deljivim resursima:
  - Semafori
  - Monitori
- Za komunikaciju među procesima:
  - Razmena poruka (Message Passing)

Semafori

- Definisao ih Dijkstra 1965.
- Služe za sinhronizaciju pristupa zajedničkim resursima.
- Semafor je struktura podataka koja sadrži:
  - Brojač – sadrži broj procesa koji mogu da pristupe zajedničkom resursu
  - Red deskritora procesa koji čekaju pristup
    - Deskriptor sadrži sve podatke potrebne da se suspendovani proces nastavi
- Semafori podržani u programskim jezicima: Ada, Java, C#...

Operacije semafora

- Cekaj (wait)

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
   decrement aSemaphore's counter
else
   put the caller in aSemaphore's queue
   attempt to transfer control to ready task
end
```

- Propusti (release)

```
release(aSemaphore)
if aSemaphore's queue is empty then
   increment aSemaphore's counter
else
   put the calling task in the task ready queue
   transfer control to a task from aSemaphore's queue
end
```

Binarni semafori

- Regulišu samo pristup zajedničkom resursu (podacima ili nekom uređaju).

```
wait(aSemaphore)
  if aSemaphore's counter == 1 then
      aSemaphore's counter = 0
  else
      put the caller in aSemaphore's queue
      attempt to transfer control to ready task
  end

release(aSemaphore)
  if aSemaphore's counter == 0 then
      aSemaphore's counter = 1
  else
      put the calling task in the task ready queue
      transfer control to a task from aSemaphore's queue
  end
```

Producer – Consumer problem

- Više procesa kreira podatke (izvršava akciju produce) i smešta ih u zajednički bafer
- Više procesa uzima podatke iz bafera (izvršava akciju consume)

- 1 rešenje:
  - 2 semafora – jedan reguliše dodavanje elementa u bafer, drugi čitanje
- 2 rešenje:
  - 3 semafora – pored prethodnih dodaje se i semafor koji reguliše pristup baferu – u jednom trenutku samo jedan proces može da pristupa baferu

Producer – Consumer sa 2 semafora

```
semaphore fullspots, emptyspots;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
    loop
    -- produce VALUE --
    wait (emptyspots); {wait for space}
    DEPOSIT(VALUE);
    release(fullspots); {increase filled}
    end loop;
end producer;
task consumer;
    loop
    wait (fullspots);{wait till not empty}}
    FETCH(VALUE);
    release(emptyspots); {increase empty}
    -- consume VALUE --
    end loop;
end consumer;
```

Producer – Consumer sa 3 semafora

```
semaphore access, fullspots, emptyspots;
access.count = 1;   {binary semaphore}
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
   loop
   -- produce VALUE --
   wait(emptyspots); {wait for space}
   wait(access);     {wait for access}
   DEPOSIT(VALUE);
   release(access); {relinquish access}
   release(fullspots); {increase filled}
   end loop;
end producer;
```

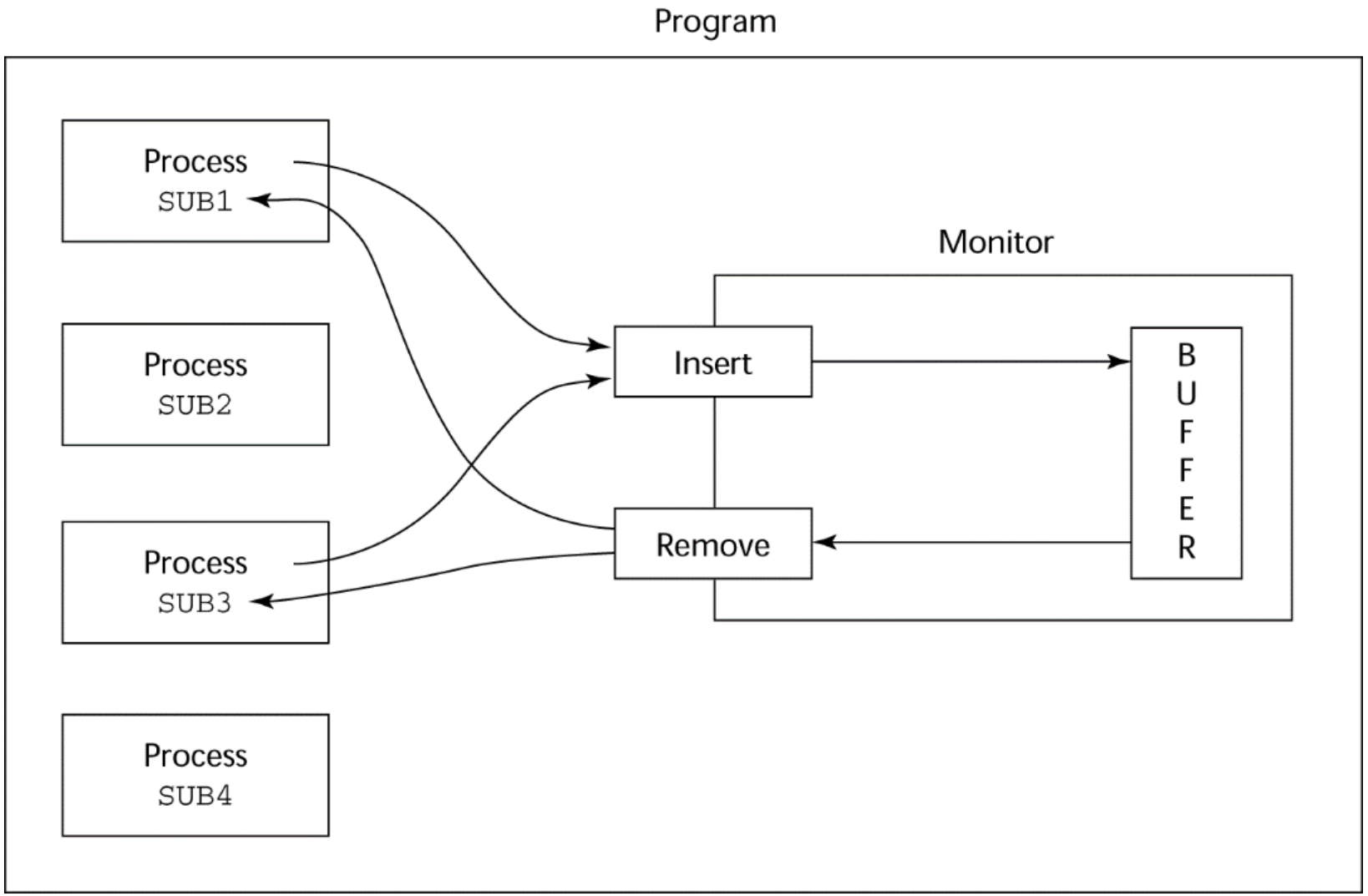
Producer – Consumer sa 3 semafora

```
task consumer;
   loop
   wait(fullspots);{wait till not empty}
   wait(access);   {wait for access}
   FETCH(VALUE);
   release(access); {relinquish access}
   release(emptyspots); {increase empty}
   -- consume VALUE --
   end loop;
end consumer;
```

Monitori

- Monitor - modul koji objedinjuje podatke koji predstavljaju zajednički resurs za više procesa sa procedurama (funkcijama) kojima se realizuje pristup tim podacima.
- Ako se neki od aktiviranih procesa obrati monitoru u trenutku kada neki drugi proces koristi procedure monitora, on se prekida i svrstava u red čekanja pridruzen monitoru.
- Kada neki prooces napušta monitor proverava se red čekanja i ukoliko u njemu ima procesa koji čekaju, prvi od njih se aktivira i dozvoljava mu se pristup podacima.
- U impelementaciji monitora se mogu koristiti semafori.
- Monitori podržani u programsim jezicima: Modula2, Ada, Java, C#...

Producesr – Consumer pomoću monitora



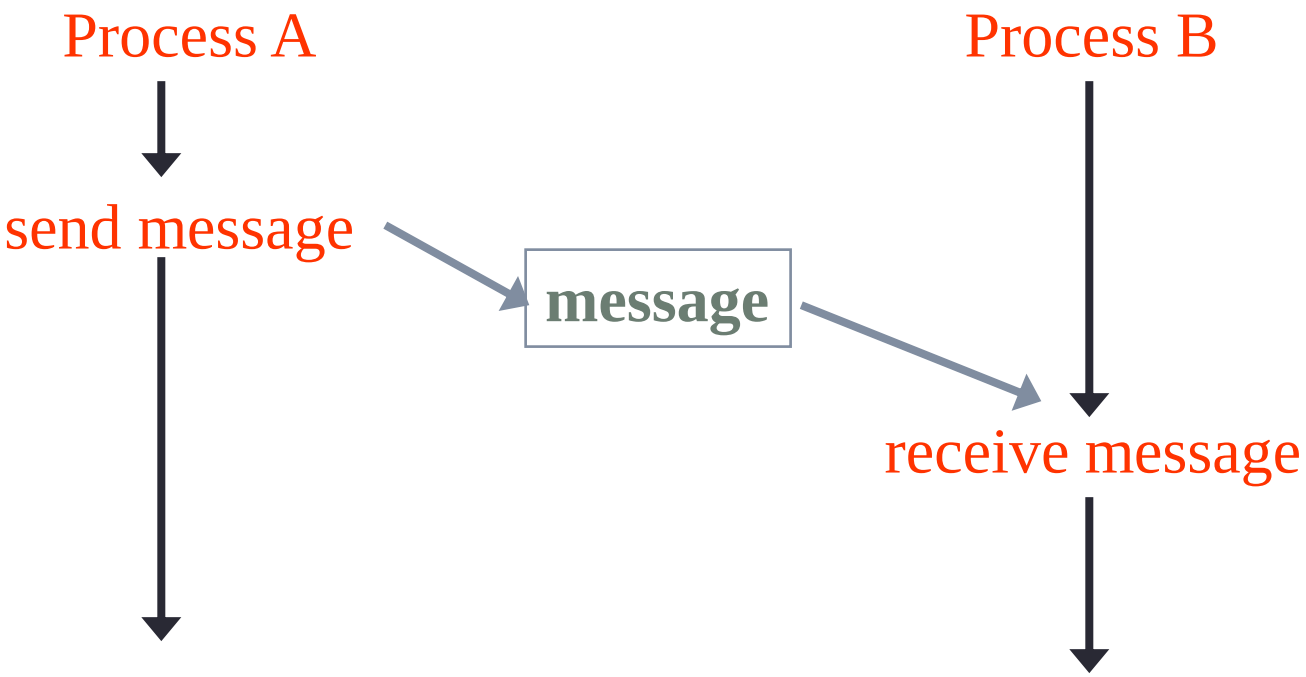
Razmena poruka

- Razmena poruka može biti:
  - Asinhrona
    - Postoji prihvatni bafer koji privremeno čuva pristigle poruke dok primalac ne bude spreman da ih primi.
  - Sinhrona (randevu)
    - Randevu je koncept sinhrone simetrične komunikacije.
    - Procesi se izvršavaju asinhrono dok ne dođu do tačke komunikacije. Ko pre stigne do navedene tačke komunikacije mora da čeka da i drugi proces dođe do te tačke.
    - Ovaj način sinhronizacije je podržan u Adi.

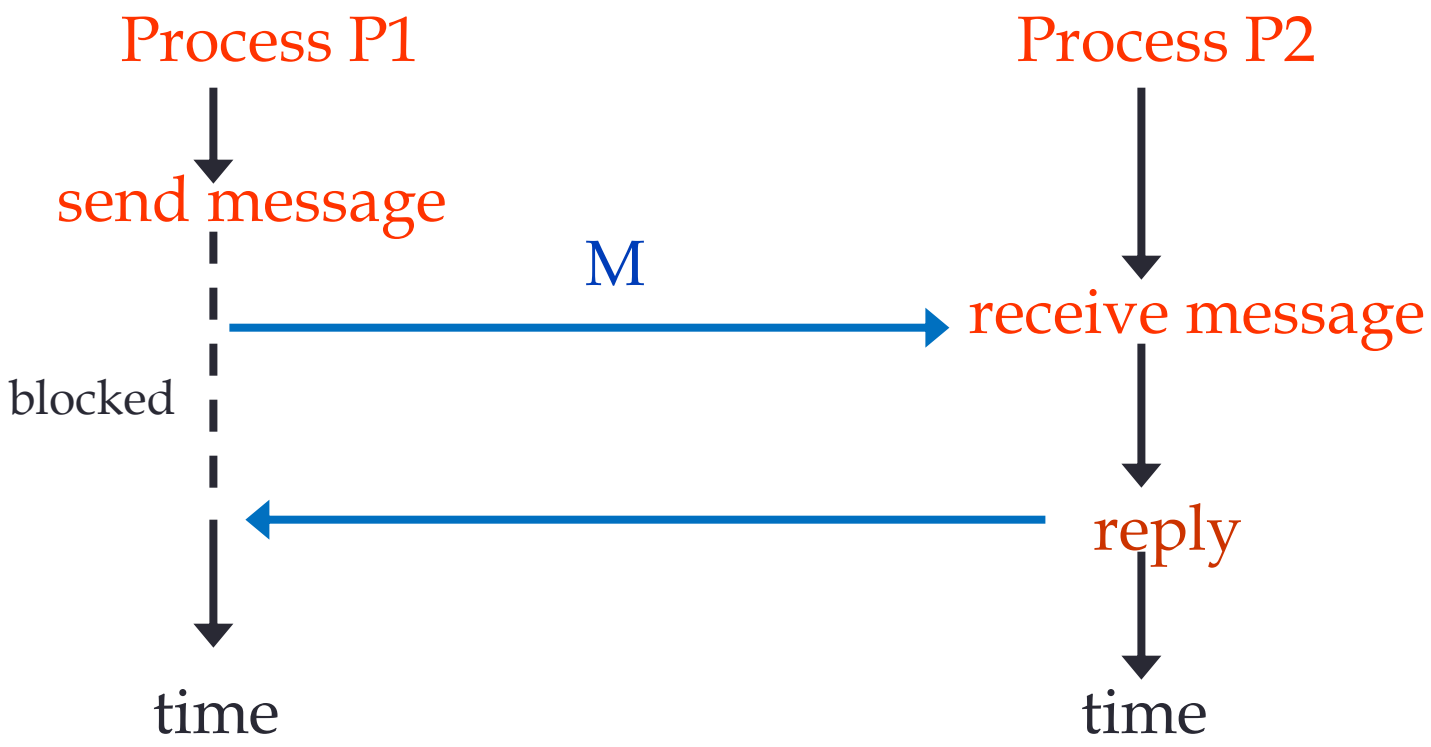
Razmena poruka – Primer u pseudokodu

```
process A;
var x: podatak;
begin
  B!x;    --Procesu B se šalje podatak x
end ;
process B;
  var y: podatak;
begin
  A?y;   --Zahteva se od procesa A podatak y
end ;
```

Asinhrona razmena poruka



Sinhrona razmena poruka



Niti u Javi

- Procesi se u Java aplikacijama definišu pomoću **niti** (eng. **threads**)
- Nit je jedan tok u izvršenju programa.
- Kad se kreira java aplikacija pokreće se jedna nit – osnovna nit aplikacije.
- Ako želimo da se izvesni delovi aplikacije izvršavaju istovrmeno, kreiraju se nove niti.

Kreiranje niti

1. Definisanje klase koja će predstavljati novu nit u programu
   - Klasa za predstavljanje niti treba da:
     - Bude izvedena iz klase Thread, ili
     - Implementira inteface Runnable.

       ```
       public interface Runnable {  public void run() ; }
       ```
2. Kreiranje objekta tipa navedene klase.
3. Pokretanje izvršenja niti.

Pokretanje izvršenja niti

- Ukoliko je klasa koja predstavlja nit u programu izvedena iz klase Thread:
  - Pozivom metode start() nad objektom kreirane klase.

- Ukoliko je klasa koja predstavlja nit implementira interfejs runnable:
  - Kreira se objekat klase Thread (čijem se konstruktoru porosleđuje objekat kreirane klase)
  - Nad tako kreiranim objektom klase Thread poziva se metoda start().

Kreiranje i pokretanje niti – 1. pristup

```
public class MyClass extends Thread
{
    public void run()
    {
        ...
    }
}
```

```
Thread thread1 = new MyThread();
thread1.start();
```

Kreiranje i pokretanje niti – 2. pristup

```
public class MyClass implements Runnable
{
    public void run()
    {
        ...
    }
}
```

```
Thread thread1 = new Thread( new MyClass() );
thread1.start();
```

Bitnije metode klase Thread

- void sleep(int ms) – uspavljuje (suspenduje) nit na određeno vreme
- void setDeamon(bool d) – postavlja indikator da li je nit demonska ili ne. Demonska nit može da nastavi svoje izvršenje i kada se nit koja ju je kreirala završi. Može da se pozove samo pre startovanja niti
- void join() – čeka se da se nit za koju je metoda pozvana završi
- void yield() – privremeno suspenduje nit – daje šansu da se izvrše niti koje čekaju na izvršenje
- string getName() – vraća ime niti. Ime niti može biti postavljeno u konstruktoru, ako nije, niti dobijaju imena 0,1,2,...

Sinhronizacija niti u Javi

- Pomoću semafora
- Pomoću monitora, tj. sinhronizovanih metoda ili blokova.

Semafori u programskom jeziku Java

- Klasa Semaphore je definisana u paketu:

  java.util.concurent

- Konstruktor klase:

  Semaphore(int permits)

- Bitnije metode:

  void acquire() //wait

  void acquire(int permits)

  bool tryAcquire()

  bool tryAcquire(int permits)

  void release()

  void release(int permits)

  int availablePermits()

Korišćenje semafora u Javi - primer

```
class MyThread extends Thread {
    Semaphore semaphore;
    MyThread(Semaphore semaphore){
        this.semaphore = semaphore;
    }
    public void run() {
            semaphore.acquire();
            System.out.println("Hello " + this.getName());
            sleep(2000);
            semaphore.release();
            System.out.println("Goodbye "+this.getName());
  }
}
```

Korišćenje semafora u Javi - primer

```
public class SemaphoreTest {
    public static void main(String args[]) {
        Semaphore semaphore=new Semaphore(2);
        MyThread mt1 = new MyThread(semaphore);
        MyThread mt2 = new MyThread(semaphore);
        MyThread mt3 = new MyThread(semaphore);
        MyThread mt4 = new MyThread(semaphore);
        mt1.start();
        mt2.start();
        mt3.start();
        mt4.start();
    }
}
```

Korišćenje semafora u Javi - primer

```
Hello Thread-0
Hello Thread-1
Goodbye Thread-0
Goodbye Thread-1
Hello Thread-2
Hello Thread-3
Goodbye Thread-2
Goodbye Thread-3
```

Sinhronizovani metodi u Javi

- Definicija sinhronizovanih metoda:
  - ključna reč synchronized ispred definicije
- Samo jedan (sinhronizovani) metod se može izvršavati u datom trenutku nad zadatim objektom

- Proces sinhronizacije koristi interni ”lock” koji je pridružen uz svaki objekat. To je neka vrsta flega koji postavlja proces (kada sinhronizovani metod započinje izvršavanje)
- Svaki sinhronizovani metod za objekat proverava da li je neki drugi metod postavio lock, pa ako jeste ne započinje izvršavanje dok se lock ne ukloni

Sinhronizovani metodi u Javi

- Ne postoji uslov da se ne mogu simultano izvršavati sinhronizovani metodi nad različitim objektima klase
- Kontroliše se samo konkurentni pristup jednom objektu
- Metodi koji nisu deklarisani kao sinhronizovani mogu se izvršavati uvek, bez obzira da li je u toku izvršavanje sinhronizovanog metoda nad objektom ili ne u nekoj drugoj niti

Sinhronizovani blokovi

- Moguće je postaviti ”lock” na proizvoljan objekat za dati blok naredbi
- Kada se izvršava blok koji je sinhronizovan za dati objekat, ne može se izvršavati ni jedan drugi blok koji je sinhronizovan za taj objekat
- Definicija snhronizovanog bloka:

```
synchronized ( theObject )
    statement;
```

Metode wait(), notify() i notifyAll()

- Definisane su u klasi Object
- Sve klase su izvedene iz Object pa mogu koristiti ove metode
- Ove metode se mogu pozivati samo iz sinhronizovanog metoda ili bloka
- Ukoliko se pozovi izvan sinhronizovanog dela koda prijavljuju izuzetak: IllegalMonitorStateException

Metode wait(), notify() i notifyAll()

- wait() – suspenduje tekuću nit sve dok se ne pozove notify() ili notifyAll() za objekat za koji je metod pozvan
  - nit oslobađa ”lock” koji ima objekat, tako da drugi sinhronizovan metod ili blok može biti izvršen za isti objekat (ovim se omogućuje da druga nit zove notify(), notifyAll() ali i wait() za isti objekat)
- notify() - restartuje nit koja je pozvala wait() metod za objekat za koji je notify() metod pozvan
  - Ako nema niti koje čekaju, ništa se ne dešava

Producer / consumer u Javi

```
class Buffer {
  private int data;
  private boolean empty;
  public Buffer() {
    this.empty = true;
  }
  public synchronized void produce(int newData) {
    while (!this.empty) {
        this.wait();
      }
    this.data = newData;
    this.empty = false;
    this.notify( );
    System.out.println("Produced:" + newData);
  }
```

Producer / consumer u Javi

```
public synchronized int consume() {
    while (this.empty) {
                this.wait();
      }
    this.empty = true;
    this.notify();
    System.out.println("Consumed:" + data);
    return data;
  }
}
```

Producer / consumer u Javi

```
class Consumer extends Thread {
  private Buffer buffer;
  public Consumer(Buffer buffer) {
    this.buffer = buffer;
  }
  public void run() {
    int data;
    while (true) {
      data = buffer.consume();
    }
  }
}
```

Producer / consumer u Javi

```
class Producer extends Thread {
  private Buffer buffer;
  public Producer(Buffer buffer) {
    this.buffer = buffer;
  }
   public void run() {
    Random rand = new Random();
    while (true) {
      int n = rand.nextInt();
      buffer.produce(n);
    }
  }
}
```

Producer / consumer u Javi

```
public class Main {
  public static void main(String[] args) {
    Buffer buffer = new Buffer();
    Producer p = new Producer(buffer);
    Consumer c = new Consumer(buffer);

    p.start();
    c.start();
  }
}
```

Niti u programkom jeziku C#

- Prostor imena **System.Threading**
  - Sadrzi skup klasa i interfejsa potrebnih za rad sa više niti (*multithread programming*)
  - Glavna klasa u ovom prostoru je **Thread**
  - Sadrzi i skup klasa i interfejsa neophodnih za sinhronizaciju niti

Klasa Thread

- Najbitnija svojstva:
  - **static CurrentThread** – nit koja se trenutno izvrsava
  - **IsAlive** – indikator koji pokazije da li se tred trenutno izvrsava
  - **IsBackground** – indikator da li se radi o pozadinskoj niti
  - **Name** – ime niti (nitima se dodeljuju imena da bi se lakse testirao I debagirao program)
  - **Priority** – prioritet niti (tipa enumeracije ThreadPriority čiji su članovi Highest, AboveNormal, Normal, BelowNormal, Lowest)
  - **ThreadState** – trenutno stanje niti (tipa enumeracije ThreadState čiji su članovi Aborted, AbortRequested, Background, Runing, Stoped, StopRequested, Suspended, SuspendRequested, NotStarted, SleepWaitJoin)

Klasa Thread

- Najbitnije metode:
  - Konstruktori:
    - **Thread(threadMethod)**
  - **Start()** ili **Start(parameter)** – pokrece izvrsenje niti gde je *parameter* parametar metode koju tred izvrsava
  - **static Suspend(thread)** – privremeno suspenduje zadatu nit
  - **static Resume(thread)** – ponovo pokrece suspendovanu nit
  - **Sleep(time)** – nit sama sebe suspenduje na zadato vreme (vreme se zadaje u milisekundama)
  - **Abort()** – trajno prekida izvrsenje niti
  - **Join()** – Zaustavlja izvrsenje niti koja je pozvala metodu dok se ne zavrsi nit za koju je metoda pozvana.

Metoda koju nit izvršava

- Obavezno tipa void
- Bez parametara ili sa parametrom tipa object

Niti u C#-u - primer

```
public static void Print()
{
  for (int i = 0; i < 10; i++)
  {
     Console.WriteLine("Thread {0}: {1}",
              Thread.CurrentThread.Name, i);
     Thread.Sleep(1);
  }
}
```

Niti u C#-u - primer

```
public static void Main(string[] args)
{
  Thread t = new Thread(Print);
  t.Name = "Child thread";
  t.Start();
  Thread.CurrentThread.Name = "Main thread";
  for (int i = 0; i < 10; i++)
  {
     Console.WriteLine("Thred {0}: {1}",
              Thread.CurrentThread.Name, i);
     if (i % 2 == 0)
           Thread.Sleep(1);
  }
  t.Join();
  Console.WriteLine("Chiled thread is finished.");
}
```

Sinhronizacija pristupa zajedničkim resursima u C#-u

- Zaključavanjem resursa
- Korišćenjem monitora (klase Monitor)
- Korišćenjem semafora (klase Semaphore)
- Korišćenjem binarnog semafora (klase Mutex)
- ...

Sinhronizacija niti zaključavanjem resursa

- Koristi se kad neki deo koda u jednom trenutku moze da izvrsava samo jedna nit
  - Tipican primer je deo koda koji radi sa nekim deljivim resursom
- Za zakljucavanje se koristi sinhronizacioni objekat (objekat koji je vidljiv svim nitima)
  - Sinhronizacioni objekat mora biti referentnog tipa
- Sintaksa:

```
lock (<lock_obj>)
{
      /*kod koji izvrsava samo nit
      koja je zakljucala lock_obj*/
}
```

Sinhronizacija niti zaključavanjem - primer

```
public class Racun {
  private double stanje;
   private readonly object lock_obj = new object();
   public Racun( double početniDepozit )
   {
        stanje = početniDepozit;
   }

   public void promeniStanje(double iznos)
   {
        lock (lock_obj)
        {
              stanje += iznos;
        }
   }
}
```

Sinhronizacija niti korišćenjem klase Monitor

- Staticka klasa koja, takodje, koristi sinhronizacioni objekat.
- Pre pocetka koda koji se zakljucava poziva se metoda:

```
Monitor.Enter(<lock_obj>)
```

- Nakon koda koji se zakljucava poziva se metoda:

```
Monitor.Exit(<lock_obj>)
```

Sinhronizacija niti korišćenjem klase Monitor - primer

```
public class Racun {
 private double stanje;
  private double lock_obj;
  public Racun( double početniDepozit )
  {
      stanje = početniDepozit;
  }

  public void promeniStanje(double iznos)
  {
      Monitor.Enter(lock_obj);
      try
      {
        …
      }
      finally
      {
         Monitor.Exit(lock_obj);
      }
  }
}
```

Zaključavanje i Monitor

Zaključavanje

```
lock(buffer) {
  //critical section
}
```

Korišćenje klase Monitor

```
Monitor.Enter(buffer);
try
{
    //critical section
}
finally
{
    Monitor.Exit(buffer);
}
```

Zaključavanje i monitor

- Monitor pruža veće mogućnosti za sinhronizaciju
- Metode monitora koje se koriste pri sinhronizaciji:
  - Monitor.TryEnter
  - Monitor.Wait
  - Monitor.Pulse
  - Monitor.PulseAll

Sinhronizacija niti pomoću mutexa i semafora

- Mutex omogućava da određenu kritičnu sekciju u kodu u jednom trenutku izvršava samo jedna nit.
- Semafori omogućavaju da određenu kritičnu sekciju u kodu u jednom trenutku izvršava najviše N procesa

- Način rada sa muteksom/semaforom:
  1. Kreirati statički objekat klase Mutex/Semaphore.
  2. U svakoj niti, kritičan deo koda (deo koda za pristup zajedničkom resursu) omeđiti pozivima metoda WhaitOne() i ReleaseMutex() /Release() nad tim objektom.

Sinhronizacija pomoću muteksa - primer

```
private static Mutex mut = new Mutex();

static void Main(String[] args)
{
  for (int i = 0; i < 3; i++)
  {
    Thread newThread = new Thread(ThreadProc);
    newThread.Name = String.Format("Thread{0}", i + 1);
    newThread.Start();
  }
}

public static void ThreadProc()
{
  for (int i = 0; i < 5; i++)
    UseResource();
}
```

Sinhronizacija pomoću muteksa - primer

```
private static void UseResource()
{
    Console.WriteLine("{0} is requesting the mutex",
                Thread.CurrentThread.Name);
    mut.WaitOne();
    Console.WriteLine("{0} has entered the protected area",
                Thread.CurrentThread.Name);
    Thread.Sleep(500);
    Console.WriteLine("{0} is leaving the protected area",
                Thread.CurrentThread.Name);
    mut.ReleaseMutex();
    Console.WriteLine("{0} has released the mutex“,
                 Thread.CurrentThread.Name);
}
```

Sinhronizacija pomoću muteksa - primer

```
Thread1 is requesting the mutex
Thread1 has entered the protected area
Thread2 is requesting the mutex
Thread3 is requesting the mutex
Thread1 is leaving the protected area
Thread1 has released the mutex
Thread2 has entered the protected area
Thread2 is leaving the protected area
Thread2 has released the mutex
Thread3 has entered the protected area
Thread3 is leaving the protected area
Thread3 has released the mutex
```