Operativni Sistemi 2

2. Komunikacija i sinhronizacija između procesa

Zadatak 1. Implementirati monitor koji predstavlja ograničeni bafer koristeći semafore.

```
class Monitor {
public:
  Monitor () : mutex(1), itemAvailable (0),
spaceAvailable(0) {}
  int take ();
  void put (int x);
private:
  Semaphore mutex;

  Semaphore itemAvailable, spaceAvailable;
          // Condition vs. Semaphore
};


int Buffer::take () {
  mutex.wait();

    while (numberInBuffer == 0) {
      mutex.signal();
      itemAvailable.wait();
      mutex.wait();
    }


  //... remove the element
  tmp = ....
  numberInBuffer--;
  spaceAvailable.signal();

  mutex.signal();

  return tmp;
}
```

U opštem slučaju, prikazana implementacija nije potpuno korektna zbog razlike između semantike operacije signal za uslovne promenljive i semafore.

Druga varijanta:

```
class Monitor {
public:
  Monitor () : mutex(1), itemAvailable (0),
spaceAvailable(0) {}
  int take ();
  void put (int x);
private:
  Semaphore mutex;

  Semaphore itemAvailable, spaceAvailable;
  int itemAvailable_c, spaceAvailable_c;
};


int Buffer::take () {
  mutex.wait();

    while (numberInBuffer == 0) {
      //wait(itemAvailable);
      itemAvailable_c++;
      mutex.signal();
      itemAvailable.wait();
      mutex.wait();
    }


  //... remove the element
  tmp = ....
  numberInBuffer--;

  //signal(spaceAvailable);
  if (spaceAvailable_c > 0){
    spaceAvailable.signal();
    spaceAvailable_c--;
  }

  mutex.signal();

  return tmp;
}
```

Problemi

1. Kako rešiti slučaj izlaska iz sredine operacije (sa return ili u slučaju izuzetka)?

```
int Monitor::criticalSection () {
  mutex.wait();
  return f()+2/x;  //  gde pozvati signal()?
}                  //  šta ako dođe do izuzetka?
```

Rešenje:

```
 class Mutex {
public:
  Mutex (Semaphore* s) : sem(s){ if (sem) sem->wait(); }
 ~Mutex ()                     { if (sem) sem->signal(); }
private:
  Semaphore *sem;
};

void Monitor::criticalSection () {
  Mutex dummy(&sem);
  //... telo kritične sekcije
}
```

2. Operacija oslobađanja kritične sekcije (`mutex.signal()`) i blokiranja na semaforu za čekanje na element (`itemAvailable.wait()`) moraju da budu nedeljive inače bi moglo da se dogodi da između ove dve operacije neki proces promeni uslov (npr. stavi element u bafer), a prvi proces se blokira na semaforu `itemAvailable` bez razloga.

Rešenje:

```
signalWait(mutex,itemAvailable); // atomicno - signalizira
                                 // na jednom semaforu,
                                 // a blokira se na drugom
```

Zadatak 2. Februar 2008.

Procesi `Alarm` i `Passengers` prikazani dole treba da se sinhronizuju na sledeći način. Proces `Passengers` signalizira ulazak i izlazak putnika iz neke obezbeđene zone. Proces `Alarm` treba da čeka blokiran sve dok broj putnika koji su trenutno u obezbeđenoj zoni ne pređe neki prag `THRESHOLD`. Tada treba da se deblokira i uključi alarm pozivom operacije `alert`. Realizovati monitor `Signaller` koji obezbeđuje opisanu sinhronizaciju pomoću klasičnih uslovnih promenljivih.

```
process Alarm;
begin
  loop
    Signaller.signal();
    alert;
  end;
end;
```

```
process Passengers;
begin
  loop
    Signaller.entry();
    Signaller.exit();
  end;
end;
```

Rešenje:

```
monitor Signaller;
  export signal, entry, exit;
  var
    count : integer;
    threshold : condition;

  procedure signal ();
  begin
    while (count<=THRESHOLD) wait(threshold);
  end;

  procedure entry ();
  begin
    count := count + 1;
    if (count>THRESHOLD) signal(threshold);
  end;

  procedure exit ();
  begin
    count := count - 1;
  end;

begin
  count := 0;
end;
```

Zadatak 3. Monitori u Javi

Na programskom jeziku Java implementirati ograničeni kružni bafer.

Napomena: Nit koja poziva `notify()` ili `notifyall()` nastavlja sa izvršavanjem u okviru monitora, dok probuđena(e) nit(i) čeka(ju) da prethodna nit oslobodi monitor i nakon toga zajedno sa svim ostalim nitima konkuriše(u) za ulazak u monitor.

Rešenje:

```
public class Buffer{
     int n, kap;

     public synchronized void put(Object obj){
          while (n == kap) {
               try {
                    wait();        // this.wait();
               } catch (InterruptedException e) { }
          }

          ++n;
          // store object

          notifyAll();
     }

     public synchronized Object get(){
          while (n == 0) {
               try {
                    wait();
               } catch (InterruptedException e) { }
          }


          --n;

          Object tmp = // get object

          notifyAll();   // da li moze notify()?

          return tmp;
     }

}
```

Zadatak 4. Oktobar 2015.

Klasa `Server`, čiji je interfejs dat dole, služi kao jednoelementni bafer za razmenu podataka između proizvoljno mnogo uporednih niti proizvođača koji pozivaju operaciju `put` i potrošača koji pozivaju operaciju `get`. Tek kada jedan proizvođač upiše podatak tipa `Data` operacijom `put`, jedan potrošač može da ga pročita operacijom `get`; tek nakon toga neki proizvođač sme da upiše novi podatak, i tako dalje naizmenično. Na jeziku Java implementirati klasu `Server` sa potrebnom sinhronizacijom.

```
class Server {
  public void put
  (Data d); public
  Data get ();

}
```

Rešenje:

```
class Server {
private Data d;
private bool readyToRead = false, readyToWrite = true;
public synchronized void put (Data data) {
   while (!this.readyToWrite) this.wait();
   this.readyToWrite=false;
   this.d=data;
   this.readyToRead=true;
   this.notifyAll();


}
public synhronized Data get () {
   while (!this.readyToRead) this.wait();
   this.readyToRead=false;
   Data data=d;
   this.readyToWrite=true;
   this.notifyAll();
   return data;
   }
}
```

Zadatak 5. Septembar 2014.
Korišćenjem klasičnih uslovnih promenljivih, napisati kod za monitor koji ima dve operacije, *flip* i *flop*, uz sledeću sinhronizaciju: između dva susedna izvršavanja operacije *flip* mora se najmanje *N* puta izvršiti operacija *flop*. Drugim rečima, nakon jednog izvršavanja *flip*, mora doći najmanje *N* (može i više) izvršavanja *flop*, pa onda opet može *flip* itd.
Rešenje:

```
monitor FlipFlop;
export flip, flop;

var i : integer,
        cond : condition;

procedure flip ();
begin
      while i<N do
        cond.wait;
      i:=0;
      (* do flip *)
end;
procedure flop ();
begin
        (* do flop *)
        if i<N then
                i:=i+1;
        cond.signal;
end;

begin
        i:=N;
end; (* monitor *)
```

Zadatak 6. Oktobar 2012.
Jedna varijanta uslovne sinhronizacije unutar monitora je sledeća. Svaki monitor ima samo jednu, implicitno definisanu i anonimnu (bez imena) uslovnu promenljivu, tako da se u monitoru mogu pozivati sledeće dve sinhronizacione operacije:

- `wait()`: bezuslovno blokira pozivajući proces i oslobađa ulaz u monitor;
- `notifyAll()`: deblokira *sve* procese koji čekaju na uslovnoj promenljivoj, ako takvih ima (naravno, mešusobno isključenje tih procesa je i dalje obezbeđeno).

Projektuje se konkurentni klijent-server sistem. Server treba modelovati monitorom sa opisanom uslovnom promenljivom. Klijenti su procesi koji ciklično obavljaju svoje aktivnosti. Pre nego što u jednom ciklusu neki klijent započne svoju aktivnost, dužan je da od servera traži dozvolu u obliku "žetona" (*token*). Kada dobije žeton, klijent započinje aktivnost. Po završetku aktivnosti, klijent vraća žeton serveru. Server vodi računa da u jednom trenutku ne može biti izdato više od N žetona: ukoliko klijent traži žeton, a ne može da ga dobije jer je nema žetona, klijent se blokira. Napisati kod monitora i procesa-klijenta.
Rešenje:

```
monitor server;
export acquireToken, returnToken;
var numOfTokens : integer;

procedure acquireToken (); begin
    while numOfTokens<=0 do wait();
    numOfTokens := numOfTokens - 1;
end;

procedure returnToken (); begin
    numOfTokens := numOfTokens + 1; notifyAll();
end;
begin
   numOfTokens := N;
end; (* server *)
task type client; begin
    loop
        server.acquireToken;
        do_some_activity;
        server.returnToken;
    end;
end; (* client *)
```

Zadatak 7. Februar 2010.

Na programskom jeziku Java, korišćenjem sinhornizovanih metoda (*synchronized*), implementirati ograničeni kružni bafer veličine N celobrojnih podataka tipa int. Pri umetanju podataka u bafer, umeće se po K podataka, gde je K konstanta koja se definiše pri kreiranju bafera. Podaci se u bafer umeću atomično. Ukoliko u baferu nema dovoljno mesta, pozivajuća nit se blokira, ali tako da se obezbedi fer pristup. To znači da nitima treba obezbediti onaj redosled umetanja elemenata u bafer u kojem su prvi put pokušale da umetnu niz od K podataka. Pri dohvatanju podataka iz bafera, dohvata se jedan po jedan podatak. Ukoliko nema dovoljno podataka, pozivajuću nit blokirati. U slučaju dohvatanja podataka nije potrebno voditi računa o fer redosledu dohvatanja podataka. Zadatak rešiti uz ograničenje da jedine sinhronizovane metode mogu biti metode klase koja predstavlja implementirani bafer. Poznato je da je K<N, ali nije poznato da li je N deljivo sa K.

Rešenje:

```
public class buffer {
  private int data[];
  private int K;
  private int N;
  private int free;   //broj slobodnih mesta u baferu
  private int tail;   //pozicija sa koje se uzima sledeci podatak
  private int head;   //pozicija na koju se smesta sledeci podatak
  private int tiket;   //uzimanje rednog broja
  private int sledeci; //redni broj koji je na redu za umetanje u bafer
                       //kada se pojavi dovoljno slobodnog mesta
    public buffer(int N, int K) {
        data = new int[N];
        this.N = N;
        free = N;
        this.K = K;
        head = 0;
        tail = 0;
        tiket = 0;
        sledeci = 0;
    }

    public synchronized void put(int d[]) throws InterruptedException{
        int red;
        if (free<K || sledeci != tiket) {
            red = tiket++;
            while(free<K || sledeci!=red){
                wait();
            }
            sledeci++;
        }
        for(int i =0; i<K; i++){
            data[head]=d[i];
            head = (head+1)%N;
        }
        free -= K;
        notifyAll();
    }

    public synchronized int get() throws InterruptedException{
        while (free == N){
```

```
            wait();
        }
        int ret = data[tail];
        tail = (tail+1)%N;
        free++;
        notifyAll();
        return ret;
    }
}
```

Zadatak 8. Decembar 2014.

Korišćenjem klasičinih monitora i uslovnih promenljivih, realizovati monitor `TaxiDispatcher` koji implementira sledeće ponašanje dispečera taksija. Korisnik (*user*) i taksi vozilo (*taxi*) su procesi koji se prijavljuju dispečeru pozivom procedura `userRequest` i `taxiAvailable`, respektivno.

Korisnik poziva proceduru `userRequest` kada želi da dobije vožnju taksijem. Ako trenutno ima raspoloživih taksi vozila koja čekaju u redu na zahtev korisnika, korisniku će odmah biti dodeljeno jedno od tih raspoloživih vozila. U suprotnom, korisnik će biti blokiran i čekaće u redu korisnika sve dok mu dispečer ne dodeli vozilo. Broj dodeljenog taksi vozila korisnik dobija u izlaznom parametru `assignedTaxiID`.

Taksi vozač poziva proceduru `taxiAvailable` kada je slobodan da primi i preveze korisnika. Kao ulazni parametar `myID` dostavlja svoj broj vozila. Ako trenutno ima korisnika koji čekaju u redu na raspoloživo vozilo, tom vozilu će biti dodeljen jedan od tih korisnika. U suprotnom, taksi će biti blokiran i čekaće u redu raspoloživih vozila sve dok mu dispečer ne dodeli korisnika.

Semantika uslovnih promenljivih je takva da proces koji je čekao na uslovnoj promenljivoj i oslobođen je ima prioritet u nastavku izvršavanja u odnosu na proces koji je signalizirao tu uslovnu promenljivu. Oba ta procesa, naravno, imaju prioritet u odnosu na druge procese koji čekaju na ulaz u monitor.

```
monitor TaxiDispatcher;
   export userRequest, taxiAvailable;
var ...;
procedure userRequest (var assignedTaxiID : integer); ...
procedure taxiAvailable (myID : integer); ...
begin ... end;
```

Rešenje:

```
monitor TaxiDispatcher;
export userRequest, taxiAvailable;

 var waitingUsers, availableTaxis : integer;
       waitForUser, waitForTaxi : condition;
       taxiID : integer;

procedure userRequest (var assignedTaxiID : integer);
begin
   if (availableTaxis>0)
      begin
         availableTaxis:=availableTaxis-1;
         waitForUser.signal;
         assignedTaxiID:=taxiID;
      end
   else
      begin
         waitingUsers:=waitingUsers+1;
         waitForTaxi.wait;
         assignedTaxiID:=taxiID;
      end;
end;
   procedure taxiAvailable (myID : integer); begin
      if (waitingUsers>0)
         begin
            waitingUsers:=waitingUsers-1;
            taxiID:=myID;
            waitForTaxi.signal;
         end
      else
         begin
            availableTaxis:=availableTaxis+1;
            waitForUser.wait;
            taxiID:=myID;
         end;
   end;

begin
   waitingUsers:=0; availableTaxis:=0;
end;
```

Zadatak 9. Januar 2007.
Neki operativni sistem podržava koncept *poštanskog sandučeta* (*mailbox*) kao sistemskog resursa koji korisnički proces od sistema dobija sistemskim pozivom:

```
MbxHandle mbx_open (char* symbolicName);
```

Ova operacija vraća „ručku“ kojom se identifikuje sanduče koje je otvoreno; ako operacija nije uspela, vraća se `NULL`. Sistem pokušava da pronađe već kreirano sanduče sa zadatim simboličkim imenom, a ako takvo ne postoji, kreira novo sanduče sa tim simboličkim imenom. Na ovaj način omogućeno je deljenje sandučića između procesa.
Operacije nad sandučićima su:

```
void mbx_send(MbxHandle, char* message);  // Asynchronous send
void mbx_receive(MbxHandle, char* message_buffer);  // Synchronous
receive
```

Napisati kod dva procesa koji međusobno komuniciraju posredstvom poštanskog sandučeta. Proces A šalje poruku procesu B sa sadržajem „?“. Na svaku ovakvu primljenu poruku, proces B odgovara procesu A porukom u kojoj je jedan novi ceo broj, formatizovan kao niz znakova – decimalnih cifara (maksimalne dužine 10). Kada je poslao „?“, proces A čeka da dobije ovaj odgovor, štampa dobijeni niz znakova na standardni izlaz, a onda nastavlja dalje sa slanjem nove poruke „?“ i tako ciklično.
Rešenje:

Process A:

```
#include ...  // System header for mailbox
#include <stdio.h>
int main () {
  MbxHandle mbxAB = mbx_open("ABMbx");
  MbxHandle mbxBA = mbx_open("BAMbx");
  if ((mbxAB==NULL)||(mbxBA==NULL)) exit(1); // Error
  while (1) {
    char buf[10];
    mbx_send(mbxAB,"?");
    mbx_receive(mbxBA,buf);
    printf(buf);
  }
}
```

Process B:

```
#include ...  // System header for mailbox
#include <stdio.h>
int main () {
  MbxHandle mbxAB = mbx_open("ABMbx");
  MbxHandle mbxBA = mbx_open("BAMbx");
  if ((mbxAB==NULL)||(mbxBA==NULL)) exit(1); // Error
  for (unsigned long i=0; i<...; i++) {
    char buf[10];
    mbx_receive(mbxAB,buf);
    if (buf[0]<>'?') continue;
    sprintf(buf,"%d",i);
    mbx_send(mbxBA,buf);
  }
  exit(0);
}
```

Java sockets

Tutoriali:
http://docs.oracle.com/javase/tutorial/networking/sockets/index.html
http://www.oracle.com/technetwork/java/socket-140484.html

1. client-side

`java.net.Socket`: osnovna klasa za izvršavanje client-side TCP operacija

- Interfejs koji klasa obezbeđuje programeru su tokovi (*stream*) podataka:
  - Slično kao rad sa fajlovima, konzolom, itd.
- Stvarno čitanje i pisanje podataka preko priključnica vrši se pomoću *stream* klasa
- klijentske priključnice se uobičajeno koriste na sledeći način:
  1. program konstruktorom kreira novi priključnicu
  2. priključnica pokušava da se poveže na udaljeni host
  3. nakon što je konekcija uspostavljena, lokalni i udaljeni host dohvataju ulazne (*input*) i izlazne (*output*) tokove podataka (*stream*) priključnice i koriste ih za razmenu podataka. Konekcija je dvosmernog tipa (*full-duplex*), tj. oba hosta mogu da primaju i šalju podatke istovremeno.
  4. kada je prenos podataka završen, jedna ili obe strane zatvaraju konekciju.

```
public Socket(String host, int port) throws
UnknownHostException, IOException
```

Parametri konstruktora: hostname, broj porta

- hostname (String) - ako *domain name server* (DNS) ne može da razreši ovo ime izbacuje se UnkownHostException
- Ukoliko priključnica ne može biti otvorena iz drugih razloga, izbacuje se IOException (npr. host ne prihvata konekcije – nije podignut serverski proces na zadatom portu)

Primer: kreiranje TCP priključnice ka zadatom portu i zadatom hostu:

```
try{
   Socket toOReilly = new Socket("test.etf.bg.ac.rs", 4444);
      // send and receive data …
      // close connection

}catch(UnkownHostException ex){
     System.err.println(ex);
}catch(IOException ex){
     System.err.println(ex);
}
```

```
public InputStream getInputStream() throws IOException
```

- vraća ulazni tok koji može čitati podatke od priključnice
- zbog poboljšanja performansi, vrši se baferisanje pomoću objekta klase *BufferedReader*

```
public OutputStream getOutputStream() throws IOException
```

- analogno za izlazni tok

Primer:

```
// ...
sock = new Socket(host, port);
out = new PrintWriter(sock.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
                                sock.getInputStream()));
// read and write data
String message = in.readLine();
out.println(message);

...
// close connection
sock.close();
```

2. *server-side*

`java.net. ServerSocket`: osnovna klasa za izvršavanje server-side TCP operacija

Primer:

```
try {
    serverSocket = new ServerSocket(4444);
}
catch (IOException e) {
    System.out.println("Could not listen on port: 4444");
    System.exit(-1);
}

Socket clientSocket = null;
try {
    clientSocket = serverSocket.accept(); //block and wait conn.
}
catch (IOException e) {
    System.out.println("Accept failed: 4444");
    System.exit(-1);
}

//na dalje analogno kao na klijentskoj strani
```

Zadatak 10. Školski Web service u Javi

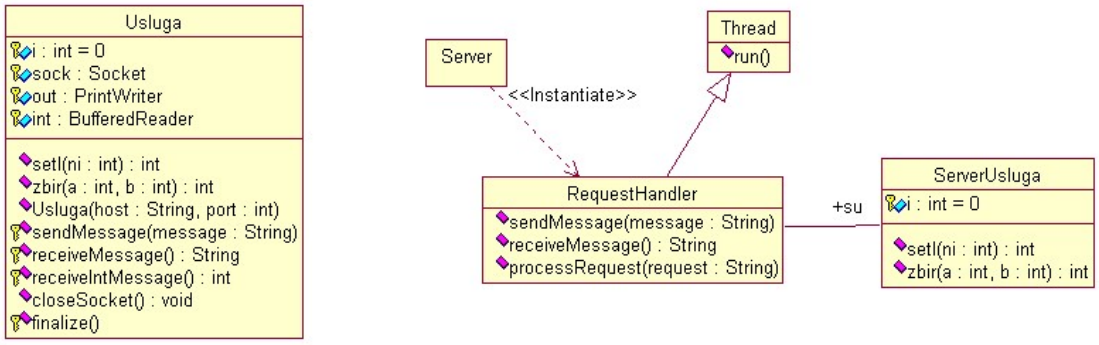
Data je klasa ServerUsluga:

```
public class ServerUsluga {
     protected int i = 0;
     public int setI(int ni) { int tmp = i; i = ni;
          return tmp;  }
     public int zbir(int a, int b) { return a+b+i; }
} // End ServerUsluga.java
```

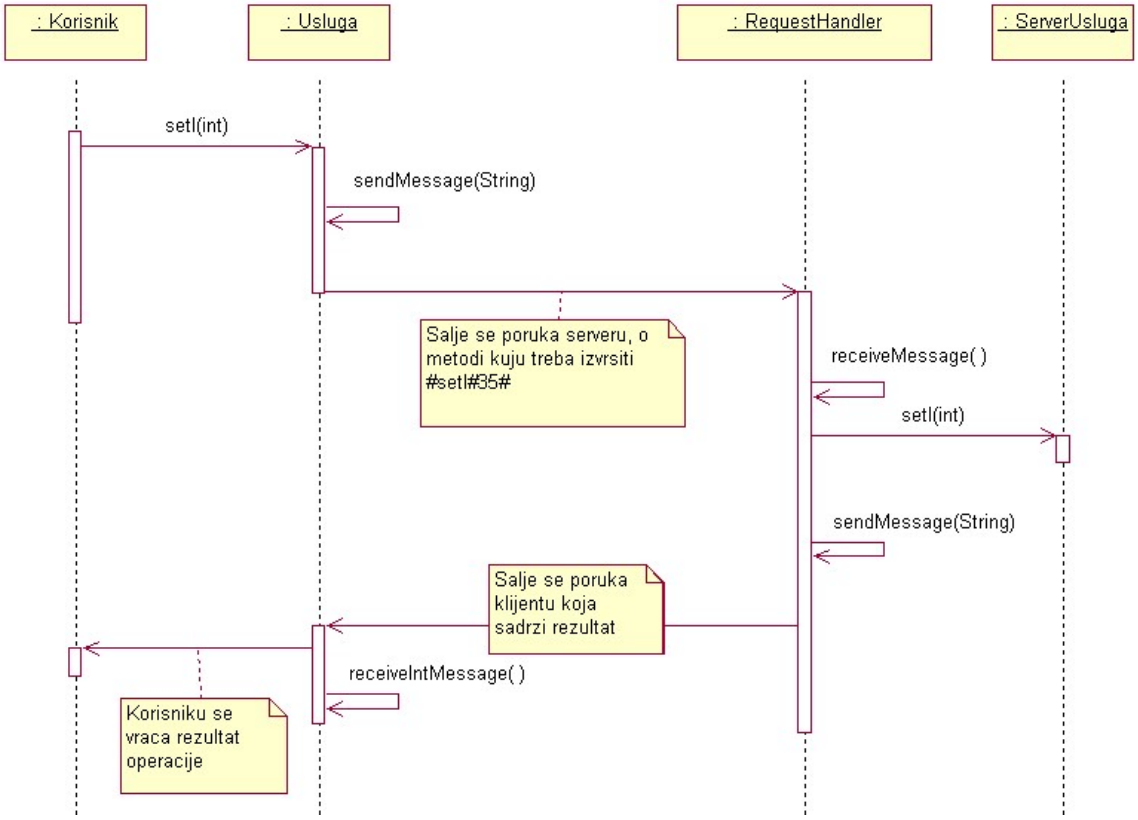
Implementirati potrebne klase koje će krajnem korisniku obezbediti identičan interfejs kao i klasa ServerUsluga i koje će mu omogućiti da sa udaljenog računara kroz obezbeđeni interfejs poziva operacije klase ServerUsluga.

Rešenje:

Dijagram klasa



Dijagram sekvence




```
// Usluga.java
import java.io.*;
import java.net.*;

public class Usluga {

     // Sve sto zanima korisnika
     public int setI(int ni) {
          //return i = ni;

          String message = "#setI#"+ ni +"#";

          sendMessage(message);

          return receiveIntMessage();
     }
```

```
public int zbir(int a, int b) {
     // return a+b;
     String message = "#zbir#" + a + "#" + b + "#";
     sendMessage(message);

     return receiveIntMessage();
}

public Usluga(String host, int port){
      try {
       sock = new Socket(host, port);
       out = new
          PrintWriter(sock.getOutputStream(), true);
       in = new BufferedReader(new InputStreamReader(
                    sock.getInputStream()));
       System.out.println("Otvoren socket...");
   } catch (UnknownHostException e) {
       System.err.println("Nepoznat host: " + host);
       System.exit(1);
   } catch (IOException e) {
 System.err.println("Ne moze da se okaci na: " +
                host + ":" + port);
       System.exit(1);
   }
}

protected void sendMessage(String message){
     System.out.println("Salje poruku: " + message);
     out.println(message);
}

protected String receiveMessage() {
     try{
          return in.readLine();
     } catch(IOException exc){
          System.out.println(
          "Greska u komunikaciji: " + exc);
          System.exit(1);
     }

     return "";
}

protected int receiveIntMessage(){
     String answer = receiveMessage();

     try{
```

```
                return Integer.parseInt(answer);
            }catch(Exception exc){
                System.out.println(
                "Greska u komunikaciji: " + exc);
                System.exit(1);
            }

            return 0;
        }

        // Stvari koje ne zanimaju krajnjeg korisnika
        protected Socket sock = null;
        protected PrintWriter out = null;
        BufferedReader in = null;

        public void closeSocket(){
            try{
                out.close();
                in.close();
                sock.close();
            } catch(Exception e){}
        }

        protected void finalize() throws Throwable{
            super.finalize();

            closeSocket();
        }
} // End Usluga.java

// ServerUsluga.java
public class ServerUsluga {
        protected int i = 0;

        public int setI(int ni) { int tmp = i; i = ni;
            return tmp;  }

        public int zbir(int a, int b) { return a+b+i; }
} // End ServerUsluga.java


// Server.java
import java.net.*;
import java.io.*;

public class Server extends Thread{
        protected ServerSocket serverSocket = null;
```

```
protected boolean kraj = false;
public Server(int port){

   try {
       serverSocket = new ServerSocket(port);
   } catch (IOException e) {
       System.err.println(
               "Ne moze da osluskuje na: " + port);
       System.exit(1);
   }

   System.out.println(
          "Pokrenut server na portu " + port);
}

public void run(){
     System.out.println(
          "Pokrenut server i ceka zahtev...");
   while(!kraj){

          Socket clientSocket = null;
          try {
               clientSocket = serverSocket.accept();
               System.out.println("Primljen zahtev!");
               Thread t =
                    new RequestHandler(clientSocket);
               t.start();
          } catch (IOException e) {}
   }
   closeSocket();
   System.out.println("Zaustavljen server");
}

public void closeSocket(){
     try{
          serverSocket.close();
     } catch(Exception e){

     }
}

public void stopServer(){
     kraj = true;
}
```

```
     public static void main(String args[]){
          Server s = new Server(6001);
          s.start();


////////////////////////////////////////////////////
           // na pritisak entera zaustavlja server
           try{
             Thread.sleep(1000);
           }catch(Exception e){}

           try{
             System.out.println(
                    "Pritisni enter za zaustavljanje CSa");
             BufferedReader br = new BufferedReader(
                    new InputStreamReader(System.in));
             br.readLine();
           } catch(Exception e){}

////////////////////////////////////////////////////

           s.stopServer();
     }
}

// RequestHandler.java
import java.net.*;
import java.io.*;

public class RequestHandler extends Thread{
     protected Socket sock;
     protected PrintWriter out;
     protected BufferedReader in;

     ServerUsluga su = new ServerUsluga();

     public RequestHandler(Socket clientSocket){
          this.sock = clientSocket;

          try{
             out = new PrintWriter(
                    clientSocket.getOutputStream(), true);
             in = new BufferedReader(new InputStreamReader(
                    clientSocket.getInputStream()));
          } catch(Exception e){
               System.out.println("Greska: " + e);
```

```
            return;
        }
    }


    public void run(){
        System.out.println("Pokrenut handler.");

        try {
            while(!sock.isInputShutdown()){
                String request = receiveMessage();

                if (!request.equals("")){
                    processRequest(request);
                }
            }

        } catch(Exception exc){}
        System.out.println("Zaustavljen handler.");
    }

    protected void processRequest(String request){
        // Trazi ime funkcije koja treba da se pozove
        // request #funcName#arg1#arg2#...

    // Prvi nacin (rucno)
        int ind1 = 1, ind2= request.indexOf("#", ind1+1);
        String functionName =
            request.substring(ind1,ind2);

        ind1 = ind2+1; ind2 =
            request.indexOf("#",ind1+1);
        String arg1 = request.substring(ind1,ind2);;

    // Drugi nacin (kraci)
        StringTokenizer st = new
            StringTokenizer(request,"#");
        String functionName = st.nextToken();
        String arg1 = st.nextToken();

        if (functionName.equals("setI")){
            int ni = Integer.parseInt(arg1);
            int staroi = su.setI(ni);
            sendMessage("" + staroi);
        } else if (functionName.equals("zbir")){
        // Prvi nacin
            ind1 = ind2+1;
```

```
                ind2 = request.indexOf("#",ind1+1);
                String arg2 = request.substring(ind1,ind2);;

            // Drugi nacin (kraci)
                String arg2 =  st.nextToken();

                int a = Integer.parseInt(arg1);
                int b = Integer.parseInt(arg2);

                int res = su.zbir(a,b);
                sendMessage("" + res);
            }
        }

        protected void sendMessage(String message){
            out.println(message);
        }

        protected String receiveMessage() {
            try{
                return in.readLine();
            } catch(IOException exc){

            }

            return "";
        }
}

// Test klijent
public class test {
        public static void main(String args[]){

            Usluga u = new Usluga("localhost",6001);

            System.out.println(u.setI(10));

            System.out.println(u.setI(15));

            System.out.println(u.zbir(5,60));

            System.out.println("Kraj testa...");
            u.closeSocket();

        }
}
```

Zadatak 11. Jun 2006.

Implementirati veb servis (*Web Service*), sa svim potrebnim delovima i na klijentskoj i na serverskoj strani, na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
public class Service{
      public Data calc(Data d){…}
      public double calc(Data d1, Data d2) {…}
}
```

Metodi `Service.calc` vrše odgovarajuće obrade nad podacima tipa `Data`, a zatim vraćaju odgovarajući rezultat. Korisnik treba da instancira objekat klase `Service` na svojoj, klijenstkoj strani, koji će predstavljati posrednik (*proxy*) do stvarnog objekta na serverskoj strani koji vrši stvarno izračunavanje. Međuprocesnu komunikaciju realizovati preko priključnica (*socket*) i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama. Interfejs klase `Data` dat je sledećim kodom:

```
public Data{
      public Data(String str){…}
      public String serialize(){…}
      …
}
```

Metod `Data.serialize` služi da napravi string reprezentaciju odgovarajućeg objekta, iz koga se može izvršiti rekonstrukcija tog objekta pomoću konstruktora `Data(String)`. Smatrati da ovaj string neće imati znak `'#'` u sebi.

Rešenje:

Slično prethodnom zadatku, uz sledeće izmene: kada je potrebno poslati podatak tipa `Data`, treba poslati string: d.serialize(). Kada je potrebno primiti podatak tipa Data, onda treba izvršiti konverziju primljene poruke (nastale sa d.serialize()) u objekat tipa Data pomoću postojećeg konstruktora:

```
message = ...;
return new Data(message);
```

Podatke tipa double, slati i primati analogno slanju i prijemu podataka tipa int.

Zadatak 12. April 2006.

Date su klase `Base` i `Derived` na programskom jeziku Java.

```
public class Base {
      public int f(int a, int b) { ... }
      public int f(int a) { ... }
}

public class Derived extends Base {
      public int f(int a, int b) { ... }
}
```

Klasa `Base` ima dve operacije `f` sa preklopljenim imenom (engl. *overloaded*). Klasa `Derived` je izvedena iz klase `Base` i u njoj je redefinisan metod `f(int,int)` (engl. *overrided*). Implementirati klase `BaseProxy` i `DerivedProxy`, koje će se instancirati na klijentskoj strani, a koje će predstavljati posrednike (engl. *proxy*), do stvarnih objekata na serverskoj strani koji izvršavaju stvarno izračunavanje. Međuprocesnu komunikaciju realizovati preko priključnica (engl. *socket*) i razmenom poruka (engl. *message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama.

Rešenje:
Jedina bitna razlika u odnosu na prethodne zadatke je da u poruku pored naziva operacije mora biti uključen i naziv tipa, kako bi se razlikovale metode iz dve prikazane klase. Takođe, u zavisnosti od broja prosleđenih parametara treba voditi računa koja se operacija klase `Base` poziva.

Zadatak 13. Oktobar 2006.

Implementirati Web Service, na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
public class NetSemaphore {
     public NetSemaphore(String host, int port){…}
     public void create(String name, int initialValue){…}
     public void signalS(String name){…}
     public void waitS(String name){…}
}
```

Servis treba da obezbedi sinhronizaciju niti koje se izvršavaju na udaljenim računarima. `NetSemaphore` se povezuje sa odgovarajućim serverom preko koga se vrši sinhronizacija. Metode `create`, `signalS` i `waitS` imaju semantiku primitiva za rad sa standardnim brojačkim semaforima, a služe da kreiraju, signaliziraju i čekaju na semafor `name` koji se nalazi na serveru. Ako je semafor sa imenom `name` već postoji, onda primitiva `create` nema efekta, već se koristi postojeći semafor. Ako se pozivaju primitive `signalS` i `waitS` na semaforima koji ne postoje, ignorisati te pozive. Pretpostaviti da postoji klasa `Semaphore` koja implementira standardni brojački semafor i ima sledeći interfejs:

```
public class Semaphore {
     public Semaphore(int initialValue){…}
     public void signalS(){…}
     public void waitS(){…}
}
```

Međuprocesnu komunikaciju realizovati preko priključnica (*socket*) i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama (kod sa vežbi ne treba prepisivati, nego npr. reći koja klasa ili koji metod se koriste i/ili menjaju, nasleđuju, ...).

Rešenje:

```
public class NetSemaphore extends Usluga {
  public NetSemaphore(String host, int port){
    super(host, port);
  }
  public void create(String name, int initialValue){
    String message = "#create#" + name + "#" + initialValue
      + "#";
    sendMessage(message);
    receiveMessage();
  }
```

```
  public void signalS(String name) {
    String message = "#signal#" + name + "#";
    sendMessage(message);
    receiveMessage();
  }
  public void waitS(String name) {
    String message = "#wait#" + name + "#";
    sendMessage(message);
    receiveMessage();
  }
}
```

Na serverskoj strani u klasi `Server` treba da se dodaju sledeće promenljive:

```
HashMap sems = new HashMap(); // kolekcija semafora
Semaphore mutex = new Semaphore(1); // sinhronizaciona
promenljiva
```

`RequestHandler` treba izmeniti na sledeći način:

```
public class RequestHandler extends Thread {
...
     HashMap semaphores;
     Semaphore mutex;
...
public RequestHandler(Socket clientSocket,
 HashMap semaphores, Semaphore mutex){
  this.sock = clientSocket;
  this.semaphores = semaphores;
  this.mutex = mutex;
          ...
}

protected void processRequest(String request){
  StringTokenizer st = new StringTokenizer(request,"#");
  String functionName = st.nextToken();
  String semName = st.nextToken();

  if (functionName.equals("create")){
    mutex.waitS(); //ulazak u krit. sekciju
```

```
      // pristup deljenoj strukturi
      if (!semaphores.containsKey(semName)){
        int initialValue = Integer.parseInt(st.nextToken());
        semaphores.put(semName, new Semaphore(initialValue));
      }
      mutex.signalS();
    } else if (functionName.equals("signal") ||
                  functionName.equals("wait")){
      Semaphore s = null;
      mutex.waitS();
      if (semaphores.containsKey(semName))
        s = (Semaphore) semaphores.get(semName);
      mutex.signalS();
      if (s != null) {
        if (functionName.equals("signal")) s.signalS();
        else s.waitS();
      }
    }
    sendMessage("done");
  }
```

Zadatak 14. Jun 2007.

Na jeziku Java napisati kod za serverski demonski proces koji će obavljati sledeći posao:

- na portu 1024 „osluškivati“ zahteve sa klijenata;
- za svaki novi zahtev sa klijenta na ovom portu, zauzeće jedan novi port u opsegu 1025..1024+*N* koji do sada već nije zauzet na ovaj način (*N* je konstanta);
- kreirati novu nit koja će sa klijentom obavljati komunikaciju preko ovog novog porta;
- javiti klijentu broj ovog novog porta, a zatim nastaviti da „osluškuje“ nove zahteve.

Nakon kreiranja nove niti, klijent prelazi na komunikaciju sa tom niti na dostavljenom portu. Serverska nit uspostavlja ovu komunikaciju na zahtev klijenta i dalje obavlja neki specifičan posao (samo naglasiti mesto gde se taj posao obavlja).

Rešenje:

```
import java.net.*;
import java.io.*;

private class ChannelServer extends Thread {
  private ServerSocket mySocket;

  public ChannelServer (int port) {
    mySocket = new ServerSocket(port);
  }

  public void run () {
    try {
      Socket client = mySocket.accept();
      //... Ovde se radi specifičan posao
      client.close();
      MainServer.releasePort(port);
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }
}

public class MainServer {
  private final static int N = ...;
  private static bool[] allocatedPorts = new bool[N];

  private static synchronized int getFreePort () {
    for (int i=0; i<N; i++) {
      if (allocatedPorts[i]) continue;
      allocatedPorts[i]=true;
      return 1025+i;
    }
    return -1;
  }
  private static synchronized int releasePort(int i) {
```

```
      allocatedPorts[i-1025]=false;
  }

  public static void main (String[] args) {
    try {
      ServerSocket sock = new ServerSocket(1024);
      while (true) {
        Socket client = sock.accept();
        int newPort = getFreePort();
        if (newPort==-1) break;
        new ChannelServer(newPort).start();
        PrintWriter pout = new
PrintWriter(client.getOutputStream(),true);
        pout.println(new Integer(newPort).toString());
        client.close();
      }
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

Zadatak 15. Prvi kolokvijum 2011.
Na jeziku Java implementirati serverski proces koji predstavlja agenta na aukciji. Ovaj proces treba da „osluškuje“ port 1025 preko koga prima poruku za otvaranje nadmetanja sa početnom cenom. Zatim, nakon zatvaranja prethodne konekcije, po istom portu počinje da prihvata ponude od ostalih učesnika u nadmetanju, pri čemu pamti trenutno najveću ponudu. U svakoj ponudi učesnik se identifikuje svojom vrednošću priključnice (IP adresa i port računara preko koga prima odgovor). Svaka nova ponuda mora biti veća od prethodne, inače se ponuđaču odmah vraća informacija o odbijanju ponude. U suprotnom se vraća poruka da je ponuda prihvaćena i da se čeka krajnji ishod nadmetanja. U slučaju da u međuvremenu pristigne ponuda sa većom vrednošću, vraća se informacija o odbijanju ponude, a ukoliko među prethodnih 5 ponuda ne stigne ni jedna veća, nadmetanje se zatvara, a procesu koji predstavlja učesnika sa najvišom ponudom šalje se poruka o pobedi. Za sinhronizaciju i komunikaciju koristiti priključnice (*sockets*) i mehanizam prosleđivanja poruka (*message passing*).

Rešenje:

```
import java.io.*;
import java.net.*;
import java.util.StringTokenizer;

public class Agent {

static int bestOffer;
static String bestClientHost = null;
static int bestClientPort;
static boolean auctionOver = false;
static int badOfferCounter = 0;
static BufferedReader in;
static PrintWriter out;

public static void main(String[] args) {
 try {
  ServerSocket sock = new ServerSocket(1025);
  Socket clientSocket = sock.accept();
  in = new BufferedReader(new InputStreamReader(
    clientSocket.getInputStream()));
  StringTokenizer st = new StringTokenizer(in.readLine(), "#");
  if (!st.nextToken().equals("StartAuction")) {
    auctionOver = true;
  }else {
    bestOffer = Integer.parseInt(st.nextToken());}

  clientSocket.close();

  while (!auctionOver) {
   clientSocket = sock.accept();

   in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
   out = new PrintWriter(clientSocket.getOutputStream(), true);
   st = new StringTokenizer(in.readLine(), "#");
   String clientHost = st.nextToken();
   String clientPort = st.nextToken();
   int newOffer = Integer.parseInt(st.nextToken());
```

```
   if (newOffer > bestOffer) {
    if (bestClientHost != null)
      sendMsgToBestClient("BetterOfferReceived");
    bestOffer = newOffer;
    bestClientHost = clientHost;
    bestClientPort = Integer.parseInt(clientPort);
    out.println("OfferAccepted");
    badOfferCounter = 0;
   } else {
    out.println("OfferRejected");
    badOfferCounter++;
   }

   clientSocket.close();

   if (badOfferCounter == 5) {
    if (bestClientHost != null) sendMsgToBestClient("YouWon!");
    auctionOver = true;
   }
  }
 } catch (Exception e) { System.err.println(e);}
}

static void sendMsgToBestClient(String msg) throws
UnknownHostException, IOException {
 Socket clientSocket = new Socket(bestClientHost,bestClientPort);
 PrintWriter oldOut = new
PrintWriter(clientSocket.getOutputStream(),true);
 oldOut.println(msg);
 clientSocket.close();
 }
}
```

Zadatak 16. Večera filozofa

Pet filozofa sedi za okruglim stolom na kome se nalazi posuda sa špagetima, jedu i razmišljaju. Svaki filozof ima svoj tanjir, a između svaka dva susedna tanjira stoji po jedna viljuška. Pretpostavlja se da su svakom filozofu, da bi se poslužio, potrebne dve viljuške, kao i da može da koristi samo one koje se nalaze levo i desno od njegovog tanjira. Ako je jedna od njih zauzeta, on mora da čeka. Svaki filozof ciklično jede, pa razmišlja. Kad završi sa jelom, filozof spušta obe viljuške na sto i nastavlja da razmišlja. Posle nekog vremena, filozof ogladni i ponovo pokušava da jede. Potrebno je definisati protokol (pravila ponašanja, algoritam) koji će obezbediti ovakvo ponašanje filozofa i pristup do viljušaka.

Rešenje:

Neophodni uslovi za nastanak mrtvog blokiranja:

- *Međusobno isključenje* (*mutual exclusion*): bar jedan resurs mora biti nedeljiv – samo ga jedan proces može koristiti u jednom trenutku
- *Držanje i čekanje* (*hold and wait*): mora postojati proces koji drži bar jedan resurs i istovremeno čeka na neki drugi
- *Nema preotimanja* (*no preemption*): resursi se ne mogu preotimati; resurs može dobrovoljno osloboditi samo proces koji ga je zauzeo
- *Kružno čekanje* (*circular wait*): mora postojati cikličan lanac procesa tako da svaki proces u lancu drži resurs koga traži naredni proces u lancu

Mrtvo blokiranje može nastati samo ako su sva četiri uslova ispunjena!

```
for(i = 0; i<5; i++) fork[i] = new Semaphore(1);
ticket = new Semaphore(4);

Filozof:
          while(1){
               think();
               ticket->waitS();
               fork[left]->waitS();
               fork[right]->waitS();
               eat();
               fork[right]->signalS();
               fork[left]->signalS();
               ticket->signalS();
          }
```

Zadatak 17. Problem čitača i pisača

Potrebno je pomoću standardnog monitora obezbediti sinhronizaciju za procese koji pristupaju deljenim podacima. Jedna grupa pristupa podacima tako što čita vrednost, dok druga grupa pristupa podacima tako što upisuje novu vrednost. Za potrebe sinhronizacije potrebno je obezbediti po dve metode za obe grupe procesa, dve za traženje dozvole za početak operacije čitanja, odnosno upisa, i dve za signaliziranje krajeva odgovarajućih operacija. Smatrati da procesi koji su bili blokirani unutar procedura monitora imaju prioritet za ulazak u monitor u odnosu na procese koji su zaustavljeni na ulazu u proceduru.

Rešenje:

```
readers_and_writers: monitor;
     var:       readcount: integer;
                busy: boolean;
                OKtoread, OKtowrite: condition;
     procedure startRead;
          begin
                if busy or OKtowrite.queue then
                     OKtoread.wait;

                readcount := readcount + 1;

                OKtoread.signal
          end;
     procedure endRead;
          begin
                readcount := readcount - 1;
                if readcount = 0 then OKtowrite.signal
          end;
     procedure startWrite;
          begin
                if readcount <> 0 or busy then
                     OKtowrite.wait;
                busy := true
          end;
     procedure endWrite;
          begin
                busy := false;
                if OKtoread.queue then OKtoread.signal
                else OKtowrite.signal
          end;

     begin
          readcount := 0;
          busy := false;
     end.
```