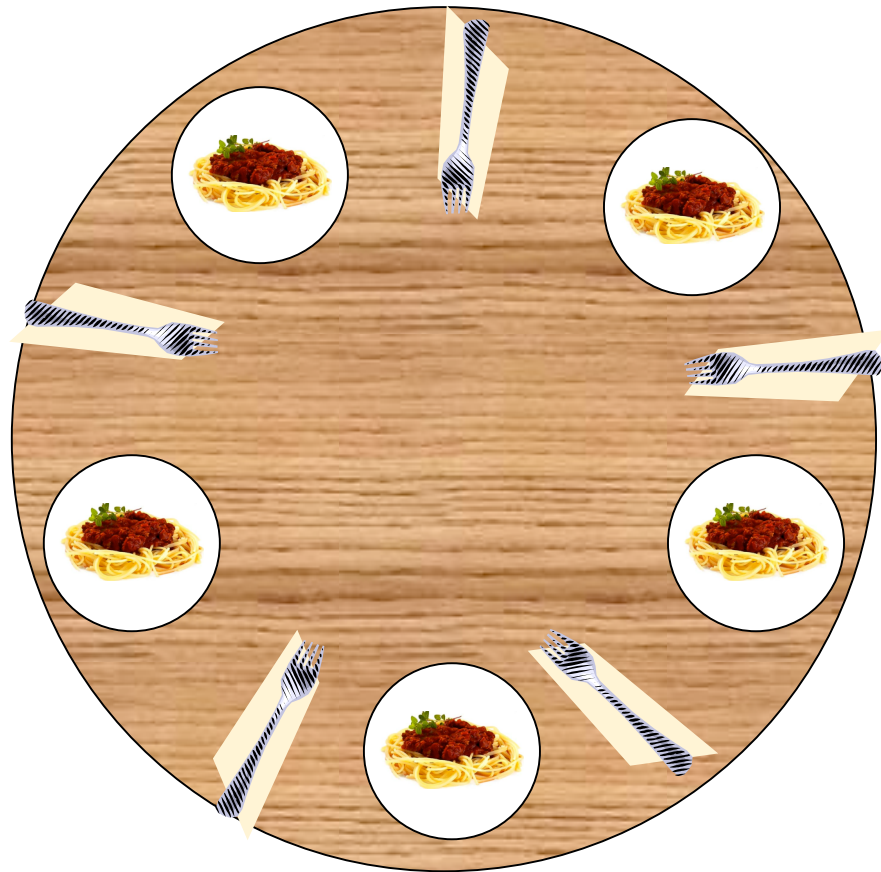


Esercizi sul Monitor in Java

I filosofi a cena (E. Dijkstra, 1965)

- 5 filosofi sono seduti attorno a un tavolo circolare; ogni filosofo ha un piatto di spaghetti tanto scivolosi che necessitano di 2 forchette per poter essere mangiati; sul tavolo vi sono in totale 5 forchette.
- Ogni filosofo ha un comportamento ripetitivo, che alterna due fasi:
 - una fase in cui **pensa**,
 - una fase in cui **mangia**.

Rappresentando ogni filosofo con un thread, realizzare una politica di sincronizzazione che eviti situazioni di deadlock.



Osservazioni

- i filosofi non possono mangiare tutti insieme: ci sono solo 5 forchette, mentre ne servirebbero 10;
- 2 filosofi vicini non possono mangiare contemporaneamente perche' condividono una forchetta e pertanto quando uno mangia, l'altro e' costretto ad attendere

Soluzione n.1

Quando un filosofo ha fame:

1. prende la forchetta a sinistra del piatto
2. poi prende quella che a destra del suo piatto
3. mangia per un po'
4. poi mette sul tavolo le due forchette.

→ **Possibilita` di deadlock:** se tutti i filosofi afferrassero contemporaneamente la forchetta di sinistra, tutti rimarrebbero in attesa di un evento che non si potra` mai verificare.

Soluzione n.2

Ogni filosofo verifica se entrambe le forchette sono disponibili:

- in caso affermativo, acquisisce le due forchette (in modo atomico);
- in caso negativo, aspetta.

→ in questo modo non si può verificare *deadlock* (non c'è possesso e attesa)

Realizzazione soluzione 2

Quali thread?

- filosofo

Risorsa condivisa?

la tavola apparecchiata

-> definiamo la classe **tavola**, che rappresenta il monitor allocatore delle forchette

Struttura Filosofo;

```
public class filosofo extends Thread
{
    tavola m;
    int i;
    public filosofo(tavola M, int id){this.m =M;this.i=id;}

    public void run()
    {
        try{
            while(true)
            {
                System.out.print("Filosofo "+ i+" pensa....\n");
                m.prendiForkette(i);
                System.out.print("Filosofo "+ i+" mangia....\n");
                sleep(8);
                m.rilasciaForkette(i);
                sleep(100);
            }
        }catch(InterruptedException e){}
    }
}
```


Monitor

```
public class tavola
{ //Costanti:
    private final int NF=5;          // costante: num forchette/filosofi
    //Dati:
    private int []forchette=new int[NF]; //num forchette disponibili per ogni
    filosofo i
    private Lock lock= new ReentrantLock();
    private Condition []codaF= new Condition[NF]; //1 coda per ogni filosofo i
    //Costruttore:
    public tavola( ) {
        int i;
        for(i=0; i<NF; i++)
            codaF[i]=lock.newCondition();
        for(i=0; i<NF; i++)
            forchette[i]=2;
    }
    // metodi public e metodi privati:...}
```

Metodi public

```
public void prendiForchette(int i) throws InterruptedException
{
    lock.lock();
    try
    {
        while (forchette[i] != 2)
            codaF[i].await();

        forchette[sinistra(i)]--;
        forchette[destra(i)]--;

    } finally{ lock.unlock(); }
    return;
}
```

```

public void rilasciaForchette(int i) throws
    InterruptedException
{
    lock.lock();
    try
    {
        forchette[sinistra(i)]++;
        forchette[destra(i)]++;
        if (forchette[sinistra(i)]==2)
            codaF[sinistra(i)].signal();
        if (forchette[destra(i)]==2)
            codaF[destra(i)].signal();

    } finally{ lock.unlock();}
    return;
}

```

Metodi privati

```
int destra(int i)
{ int ret;
  ret=(i==0? NF-1:(i-1));
  return ret;
}
```

```
int sinistra(int i)
{ int ret;
  ret=(i+1)%NF;
  return ret;
}
```

Programma di test

```
import java.util.concurrent.*;

public class Filosofi {
    public static void main(String[] args) {
        int i;
        tavola M=new tavola();
        filosofo []F=new filosofo[5];
        for(i=0;i<5;i++)
            F[i]=new filosofo(M, i);
        for(i=0;i<5;i++)
            F[i].start();
    }
}
```

Il Bar dello Stadio

In uno stadio e' presente un unico bar a disposizione di tutti i tifosi che assistono alle partite di calcio. I tifosi sono suddivisi in due categorie: tifosi della squadra **locale**, e tifosi della squadra **ospite**.

Il bar ha una capacita` massima **NMAX**, che esprime il numero massimo di persone (tifosi) che il bar puo` accogliere contemporaneamente.

Per motivi di sicurezza, nel bar **non e' consentita la presenza contemporanea di tifosi di squadre opposte**.

Il bar e' gestito da un **barista** che puo` decidere di chiudere il bar in qualunque momento per effettuare la **pulizia** del locale. Al termine dell'attivitaa` di pulizia, il bar verra` riaperto.

Durante il periodo di chiusura non e' consentito l'accesso al bar a nessun cliente.

Nella fase di chiusura, potrebbero essere presenti alcune persone nel bar: in questo caso il barista attendera` l'uscita delle persone presenti nel bar, prima di procedere alla pulizia.

Utilizzando il linguaggio Java, si modellino i clienti e il barista mediante thread concorrenti e si realizzi una politica di sincronizzazione tra i thread basata sul concetto di monitor che tenga conto dei vincoli dati, e che inoltre, **nell'accesso al bar dia la precedenza ai tifosi della squadra ospite**.

Impostazione

Quali thread?

- barista
- cliente ospite
- cliente locale

Risorsa condivisa?

il bar

-> definiamo la classe **Bar**, che rappresenta il monitor allocatore della risorsa

Struttura dei threads: ospite

```
public class ClienteOspite extends Thread
{
    Bar m;

    public ClienteOspite(Bar M){this.m = M;}

    public void run()
    {
        try{
            m.entraO();
            System.out.print( "Ospite: sto consumando...\n");
            sleep(2);
            m.esciO();

        }catch(InterruptedException e){}

    }
}
```


Struttura dei threads: locale

```
public class ClienteLocale extends Thread
{
    Bar m;

    public ClienteLocale(Bar M){this.m =M;}

    public void run()
    {
        try{
            m.entraL();
            System.out.print( "Locale: sto consumando...\n");
            sleep(2);
            m.esclL();

        }catch(InterruptedException e){}

    }
}
```

Struttura dei threads: barista

```
public class Barista extends Thread
{
    Bar m;

    public Barista(Bar M){this.m =M;}

    public void run()
    {
        try{ while(1)
            {
                m.inizio_chiusura();
                System.out.print( "Barista: sto pulendo...\n");
                sleep(8);
                m.fine_chiusura();
                sleep(10);
            }
        }catch(InterruptedException e){}

    }
}
```

Progetto del *monitor* bar:

```
import java.util.concurrent.locks.*;

public class Bar
{ //Dati:
  private final int N=20; //costante che esprime la capacita` bar
  private final int Loc=0; //indice clienti locali
  private final int Osp=1; //indice clienti ospiti
  private int[] clienti; //clienti[0]: locali; clienti[1]: ospiti
  private boolean uscita;// indica se il bar è chiuso, o sta per chiudere
  private Lock lock= new ReentrantLock();
  private Condition clientiL= lock.newCondition();
  private Condition clientiO= lock.newCondition();
  private Condition codabar= lock.newCondition(); //coda barista
  private int[] sospesi;
  //Costruttore:
  public Bar() {
    clienti=new int[2];
    clienti[Loc]=0;
    clienti[Osp]=0;
    sospesi=new int[2];
    sospesi[Loc]=0;
    sospesi[Osp]=0;
    uscita=false;
  }
```

```

//metodi public:

public void entraL() throws InterruptedException
{ lock.lock();
  try
  {
      while ((clienti[Osp] != 0) ||
              (clienti[Loc] == N) ||
              uscita ||
              (sospesi[Osp] > 0) )
      {
          sospesi[Loc]++;
          clientiL.await();
          sospesi[Loc]--;
      }
      clienti[Loc]++;
      if (sospesi[Loc]) clientiL.signal();
  } finally{ lock.unlock();}
  return;
}

```

```

public void entraO() throws InterruptedException
{
    lock.lock();
    try
    {
        while ((clienti[Loc] != 0) ||
                (clienti[Osp] == N) ||
                uscita )
        {
            sospesi[Osp]++;
            clientiO.await();
            sospesi[Osp]--;
        }
        clienti[Osp]++;
        if (sospesi[Osp]) clientiO.signal ( );
    } finally{ lock.unlock();}
    return;
}

```

```

public void esciO() throws InterruptedException
{ lock.lock();
  try
  { clienti[Osp]--;
    if (uscita && (clienti[Osp]==0))
      codabar.signal();
    else if (sospesi[Osp]>0)
      clientiO.signal();
    else clientiL.signal();
  } finally{ lock.unlock();}
}

```

```

public void esciL () throws InterruptedException
{ lock.lock();
  try
  { clienti[Loc]--;
    if (uscita && (clienti[Loc]==0))
      codabar.signal();
    else if (sospesi[Osp]>0) clientiO.signal();
    else clientiL.signal();
  } finally{ lock.unlock();}
}

```

```

public void inizio_chiusura() throws InterruptedException
{
    lock.lock();
    try
    {
        uscita=true;
        while ((clienti[Loc]>0) || (clienti[Osp]>0))
            codabar.await();
    } finally{ lock.unlock();}
}

```

```

public void fine_chiusura() throws InterruptedException
{
    lock.lock();
    try
    {
        uscita=false;
        if (sospesi[Osp]>0) clientiO.signal();
        else clientiL.signal();
    } finally{ lock.unlock();}
}
} // fine classe Bar

```

Programma di test

```
import java.util.concurrent.*;

public class Bar_stadio {
    public static void main(String[] args) {

        System.out.println("HELLO!!!!");
        int i;
        Bar M=new Bar();
        ClienteOspite []CO= new ClienteOspite[50] ;
        ClienteLocale []CL= new ClienteLocale[50] ;
        Barista B=new Barista(M);
        for(i=0;i<50;i++)
        {
            CO[i]=new ClienteOspite(M);
            CL[i]=new ClienteLocale(M);
        }
        for(i=0;i<50;i++)
        {
            CO[i].start();
            CL[i].start();
        }
        B.start();
    }
}
```