Strumenti di sincronizzazione nella libreria LinuxThread e nel linguaggio Java

I semafori nelle librerie `pthread` e `LinuxThreads`

- La libreria **`pthread`** definisce soltanto il ***semaforo binario*** (mutex).
- La Libreria Linuxthread, implementa comunque il semaforo esternamente alla libreria pthread, conformemente allo standard POSIX 1003.1b

pthread: MUTEX

- Lo standard POSIX 1003.1c (libreria **<pthread.h>**) definisce i **semafori binari** (o lock, mutex, etc.)
  - sono semafori il cui valore puo` essere 0 oppure 1 (*occupato* o *libero*);
  - vengono utilizzati tipicamente per risolvere problemi di **mutua esclusione**
  - **operazioni fondamentali:**
    - **inizializzazione**: `pthread_mutex_init`
    - **locking**: `pthread_mutex_lock`
    - **unlocking**: `pthread_mutex_unlock`

  - **Per operare sui mutex:**

    `pthread_mutex_t` : tipo di dato associato al mutex; esempio:

    ```
    pthread_mutex_t mux;
    ```

MUTEX: inizializzazione

- L'inizializzazione di un `mutex` si puo` realizzare con:

```
int pthread_mutex_init(pthread_mutex_t* mutex, const
    pthread_mutexattr_t* attr)
```

attribuisce un valore iniziale all'intero associato al semaforo (default: *libero*):

- `mutex` : individua il mutex da inizializzare
- `attr` : punta a una struttura che contiene gli attributi del mutex; se NULL, il mutex viene inizializzato a *libero* (default).

- in alternativa , si puo` inizializzare il mutex a default con la macro:
  `PTHREAD_MUTEX_INIZIALIZER`

- **esempio**: `pthread_mutex_t mux= PTHREAD_MUTEX_INIZIALIZER;`

MUTEX: lock/unlock

- Locking/unlocking si realizzano con:

```
int pthread_mutex_lock(pthread_mutex_t* mux)
int pthread_mutex_unlock(pthread_mutex_t* mux)
```

  - lock: se il mutex `mux` e` occupato,il thread chiamante si sospende; altrimenti occupa il mutex.
  - unlock: se vi sono processi in attesa del mutex, ne risveglia uno; altrimenti libera il mutex.

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX 10
pthread_mutex_t M; /* def.mutex condiviso tra threads */
int DATA=0; /* variabile condivisa */
int accessi1=0; /*num. di accessi del thread 1 alla sez critica */
int accessi2=0; /*num. di accessi del thread 2 alla sez critica */

void *thread1_process (void * arg)
{   int k=1;
    while(k)
    {
        pthread_mutex_lock(&M); /*prologo */
        accessi1++;
         DATA++;
        k=(DATA>=MAX?0:1);
        printf("accessi di T1: %d\n", accessi1);
        pthread_mutex_unlock(&M); /*epilogo */
    }
    pthread_exit (0);
}
```

Esempio

```
void *thread2_process (void * arg)
{    int k=1;
   while(k)
    {
        pthread_mutex_lock(&M); /*prologo sez. critica */
        accessi2++;
        DATA++;
        k=(DATA>=MAX?0:1);
        printf("accessi di T2: %d\n", accessi2);
       pthread_mutex_unlock(&M); /*epilogo sez. critica*/
    }
    pthread_exit (0);
}
```

Esempio:

```
main()
{ pthread_t th1, th2;
  /* il mutex e` inizialmente libero: */
  pthread_mutex_init (&M, NULL);
   if (pthread_create(&th1, NULL, thread1_process, NULL) <
   0)
         { fprintf (stderr, "create error for thread 1\n");
           exit (1);
         }
  if (pthread_create(&th2, NULL,thread2_process,NULL) < 0)
  { fprintf (stderr, "create error for thread 2\n");
    exit (1);
  }
   pthread_join (th1, NULL);
   pthread_join (th2, NULL);
}
```

Test

```
$
$ gcc -D_REENTRANT -o tlock lock.c -lpthread
$ ./tlock
accessi di T2: 1
accessi di T1: 1
accessi di T2: 2
accessi di T1: 2
accessi di T1: 3
accessi di T1: 4
accessi di T1: 5
accessi di T1: 6
accessi di T1: 7
accessi di T1: 8
accessi di T2: 3
$
```

LinuxThreads: Semafori

Memoria condivisa: uso dei semafori (POSIX.1003.1b)

- Semafori: libreria <semaphore.h>
  - `sem_init`: **inizializzazione di un semaforo**
  - `sem_wait`: implementazione di *P*
  - `sem_post`: implementazione di *V*

- `sem_t` : tipo di dato associato al semaforo; esempio:

```
static sem_t my_sem;
```

Operazioni sui semafori

- `sem_init`: inizializzazione di un semaforo

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

attribuisce un valore iniziale all'intero associato al semaforo:

- `sem`: individua il semaforo da inizializzare
- `pshared` : 0, se il semaforo non e` condiviso tra task, oppure non zero (sempre zero).
- `value` : e` il valore iniziale da assegnare al semaforo.

- `sem_t` : tipo di dato associato al semaforo; esempio:

```
static sem_t my_sem;
```

➢ ritorna sempre 0.

Operazioni sui semafori: sem_wait

- P su un semaforo

```
int sem_wait(sem_t *sem)
```

dove:

- `sem`: individua il semaforo sul quale operare.

e` la *P* di Dijkstra:

- se il valore del semaforo e` uguale a zero, sospende il thread chiamante nella coda associata al semaforo; altrimenti ne decrementa il valore.

Operazioni sui semafori: sem_post

- *V* su un semaforo:

```
int sem_post(sem_t *sem)
```

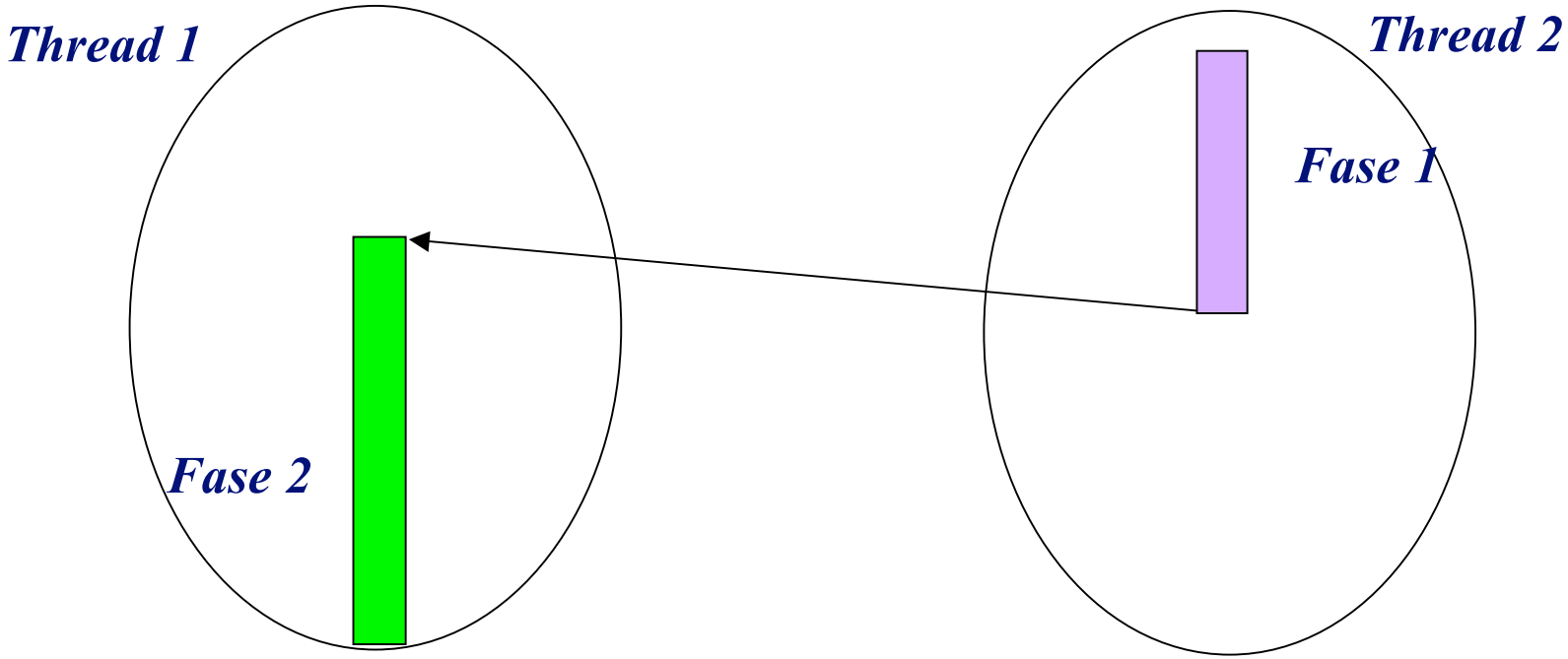
dove:

- `sem`: individua il semaforo sul quale operare.

e` la *V* di Dijkstra:

- se c'e` almeno un thread sospeso nella coda associata al semaforo sem, viene risvegliato; altrimenti il valore del semaforo viene incrementato.

Esempio: Semaforo Evento



- Imposizione di un vincolo temporale: la FASE2 nel thread 1 va eseguita dopo la FASE1 nel thread2.

Esempio: sincronizzazione

```
/* la FASE2 nel thread 1 va eseguita dopo la FASE1 nel thread 2*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

sem_t my_sem;
int V=0;

void *thread1_process (void * arg)
{
  printf ("Thread 1: partito!...\n");
   /* inizio Fase 2: */
  sem_wait (&my_sem);
  printf ("FASE2: Thread 1:  V=%d\n", V);
  pthread_exit (0);
}
```

```
void *thread2_process (void * arg)
{  int i;

    V=99;
     printf ("Thread 2: partito!...\n);
     /* inizio fase 1: */
     printf ("FASE1: Thread 2:  V=%d\n", V);
    /* …
     termine Fase 1: sblocco il thread 1*/
     sem_post (&my_sem);
     sleep (1);
  pthread_exit (0);
}
```

```
main ()
{ pthread_t th1, th2;
  void *ret;
  sem_init (&my_sem, 0, 0); /* semaforo a 0 */

if (pthread_create (&th1, NULL, thread1_process, NULL) < 0) {
   fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  if (pthread_create(&th2,NULL, thread2_process, NULL) < 0)
   {fprintf (stderr, "pthread_create error for thread \n");
    exit (1);
  }

  pthread_join (th1, &ret);
  pthread_join (th2, &ret);
}
```

Esempio:

- `gcc -D_REENTRANT -o sem sem.c -lpthread`

- **Esecuzione**:

```
[aciampolini@ccib48 threads]$ sem
Thread 1: partito!...
Thread 2: partito!...
FASE1: Thread 2:  V=99
FASE2: Thread 1:  V=99
[aciampolini@ccib48 threads]$
```

Semafori: esempio

```
/* tre processi che, ciclicamente, incrementano a
   turno (in ordine P1,P2,P3) la variabile V*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX 13
static sem_t s1,s2,s3; /* semafori per imporre
            l'ordine di accesso (P1,P2,P3) alla
            variabile V */
int V=0,F=0;
```

```
void *thread1_process (void * arg)
{   int k=1;
    while(k)
    {   sem_wait (&s1);
       if (V<MAX)
                V++;
        else
        {       k=0;
                printf("T1: %d (V=%d)\n",++F, V);
        }
       sem_post(&s2);
    }
  pthread_exit (0);
}
```

```
void *thread2_process (void * arg)
{   int k=1;
    while(k)
    {   sem_wait (&s2);
       if (V<MAX)
                V++;
         else
         {       k=0;
                 printf("T2: %d (V=%d)\n",++F, V);
         }
       sem_post(&s3);
    }
  pthread_exit (0);
}
```

```
void *thread3_process (void * arg)
{   int k=1;
    while(k)
    {   sem_wait (&s3);
        if (V<MAX)
                V++;
         else
        {       k=0;
                printf("T3: %d (V=%d)\n",++F, V);
        }
        sem_post(&s1);
    }
   pthread_exit (0);
}
```

```
main ()
{ pthread_t th1, th2,th3;

  sem_init(&s1,0,1);
  sem_init(&s2,0,0);
  sem_init(&s3,0,0);
  if (pthread_create(&th1, NULL, thread1_process, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread 1\n");
     exit (1);
  }
  if (pthread_create(&th2, NULL,thread2_process,NULL) < 0)
  { fprintf (stderr, "pthread_create error for thread 2\n");
     exit (1);
  }
  if (pthread_create(&th3,NULL,thread3_process, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread 3\n");
     exit (1);
  }
```

```
    pthread_join (th1, NULL);
    pthread_join (th2, NULL);
    pthread_join (th3, NULL);

}
```

Sincronizzazione in Java

Modello a memoria comune:

I threads di una applicazione condividono lo spazio di indirizzamento.

➔ Ogni tipo di interazione tra thread avviene tramite *oggetti* comuni:

- Interazione di tipo ***competitivo*** (*mutua esclusione*): meccanismo degli **objects locks**.
- Interazione di tipo ***cooperativo***:
  - meccanismo **wait-notify**.
  - **variabili condizione**

Semafori in Java

- Non esiste una classe che implementi i semafori. E` possibile implementarli sfruttando altri strumenti di sincronizzazione (ad es. wait/notify, Variabili Condizione).
- Il linguaggio prevede costrutti specifici per risolvere il problema della mutua esclusione:
  - statement **`syncronized`**

Mutua esclusione

- Ad ogni oggetto viene associato dalla JVM un **lock** (analogo ad un ***semaforo binario***).
- E' possibile denotare alcune sezioni di codice che operano su un oggetto come *sezioni critiche* tramite la parola chiave **`synchronized`**.

➔ Il compilatore inserisce :

  - un prologo in testa alla sezione critica per **l'acquisizione del lock** associato all'oggetto.
  - un epilogo alla fine della sezione critica per **rilasciare il lock**.

Blocchi synchronized

Con riferimento ad un oggetto x si può definire un blocco di statement come una sezione critica nel seguente modo (**synchronized blocks**):

```
synchronized (oggetto x) {<sequenza di statement>;}
```

Esempio:

```
Object mutexLock= new Object;
   …..
   public void  M( ) {
        <sezione di codice non critica>;
        synchronized (mutexlock){
        < sezione di codice critica>;
        }
       <sezione di codice non critica>;
   }
```

- all'oggetto **`mutexLock`** viene implicitamente associato un lock, il cui valore puo` essere:
  - **libero**: il thread può eseguire la sezione critica
  - **occupato**: il thread viene sospeso dalla JVM in una coda associata a **`mutexLock`** *(entry set)*.

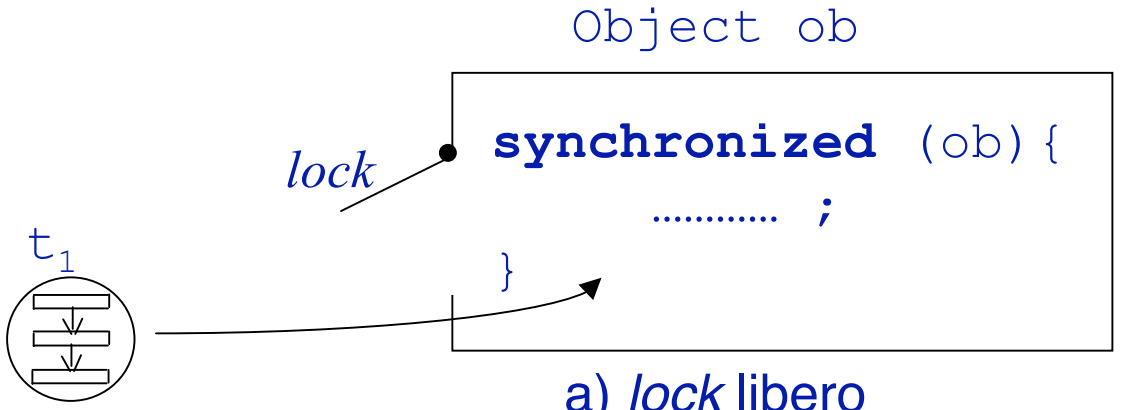
Al termine della sezione critica:

- se *non ci sono thread in attesa*: il lock viene reso libero .
- se *ci sono thread in attesa*: il lock rimane occupato e viene scelto uno di questi .

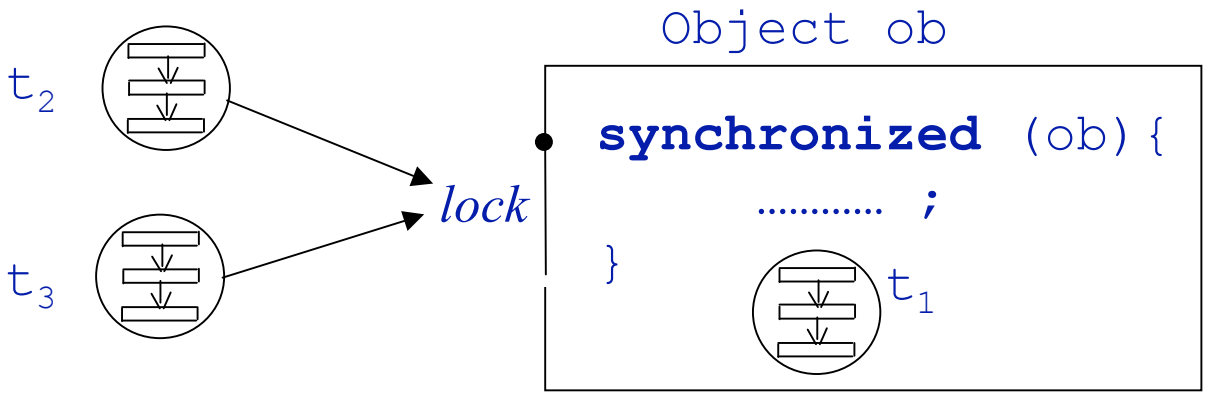
synchronized block

- esecuzione del blocco **mutuamente esclusiva** rispetto:
  - ad altre esecuzioni dello *stesso blocco*
  - all'esecuzione di *altri blocchi* sincronizzati sullo stesso oggetto

Entry set di un oggetto



a) *lock* libero



a) *lock* occupato: t2
e t3 vengono inseriti
nell'entry set di ob

Metodi synchronized

- **Mutua esclusione** tra i metodi di una classe

```
public class intVar {
private int i=0;
public synchronized void incrementa()
{ i ++; }
public synchronized void decrementa()
{i--; }
}
```

- Quando un metodo viene invocato per operare su un oggetto della classe, l'esecuzione del metodo avviene in **mutua esclusione utilizzando il *lock dell'oggetto***.

Esempio: accesso concorrente a un contatore

```
public class competingproc extends Thread
{   contatore r; /* risorsa condivisa */
    int T; // incrementa se tipo=1; decrementa se tipo=-1

    public  competingproc(contatore R,  int tipo)
    {     this.r=R;
          this.T=tipo;
    }

    public void run()
    {     try{
          while(true)
          {   if (T>0)        r.incrementa();
              else if (T<0)   r.decrementa();
          }
          }catch(InterruptedException e){}
    }
}
```

```
public class contatore {
    private int C;

    public contatore(int i)
    {     this.C=i;}

    public synchronized void incrementa()
    {     C++;
          System.out.print("\n eseguito incremento: valore    attuale del contatore:
    "+ C+" ....\n");
    }

    public synchronized void decrementa()
    {     C--;
          System.out.print("\n eseguito decremento: valore
          attuale del contatore: "+ C+" ....\n");
    }
}
```

```
import java.util.*;

public class prova_mutex{ // test

    public static void main(String args[]) {
        final int NP=30;
        contatore C =new contatore(0);
        competingproc []F=new competingproc[NP];
        int i;
        for(i=0;i<(NP/2);i++)
          F[i]=new competingproc(C, 1); // incrementa
        for(i=(NP/2);i<NP;i++)
                F[i]=new competingproc(C, -1); // decrementa
        for(i=0;i<NP;i++)
                F[i].start();
    }

}
```

Sincronizzazione: `wait` e `notify`

wait set: coda di thread associata ad ogni oggetto, inizialmente vuota.

- I thread entrano ed escono dal ***wait set*** utilizzando i metodi `wait()` e `notify()`.
- `wait` e `notify` possono essere invocati da un thread **solo all'interno di un *blocco sincronizzato* o di un *metodo sincronizzato*** (e` necessario il *possesso* del lock dell'oggetto).

wait, notify, notifyall

`wait` comporta il rilascio del lock, la sospensione del thread ed il suo inserimento in ***wait set***.
(NB. throws InterruptedException)

`notify` comporta l'estrazione di un thread da **wait set** ed il suo inserimento in ***entry set***.

`notifyall` comporta l'estrazione di **tutti** i thread da ***wait set*** ed il loro inserimento in ***entry set***.

NB: `notify` e `notifyall` non provocano il rilascio del lock: ➔ i thread risvegliati devono attendere.

➔ Politica **signal&continue**: il rilascio del lock avviene al completamento del *blocco o del metodo sincronizzato* da parte del thread che ha eseguito la notify.

```
//Esempio: mailbox con capacita`=1
public class Mailbox{
    private int contenuto;
    private boolean pieno=false;

   public synchronized int preleva()
    {   try{
        while (pieno==false)
                wait();
        pieno=false;
        notify();
        }catch(InterruptedException e){}
        return contenuto;
   }

   public synchronized void deposita(int valore)
   {    try{
        while (pieno==true)
           wait();
        contenuto=valore;
        pieno=true;
        notify();
        }catch(InterruptedException e){}
   }
}
```

```
//Mailbox di capacita` N
public class Mailbox {
  private int[]contenuto;
  private int contatore, testa, coda;

  public mailbox(){ //costruttore
    contenuto = new int[N];
    contatore = 0;
    testa = 0;
    coda = 0;
  }
  public synchronized int preleva (){
    int elemento;
    while (contatore == 0)
      wait();
    elemento = contenuto[testa];
    testa = (testa + 1)%N;
    --contatore;
    notifyAll();
    return elemento;
  }
  public synchronized void deposita (int valore){
    while (contatore == N)
      wait();
    contenuto[coda] = valore;
    coda = (coda + 1)%N;
    ++contatore;
    notifyAll();
  }
}
```

wait&notify

Principale limitazione :

- **unica coda (wait set)** per ogni oggetto sincronizzato

➔ non e` possibile sospendere thread su code differenti!

Problema superato nelle versioni più recenti di Java (versione 5.0) tramite la possibilità utilizzare le ***variabili condizione***.

Semafori in Java

- Java non prevede i semafori (versioni precedenti alla 5.0); tuttavia essi possono essere facilmente costruiti mediante i meccanismi di sincronizzazione standard (wait e notify).

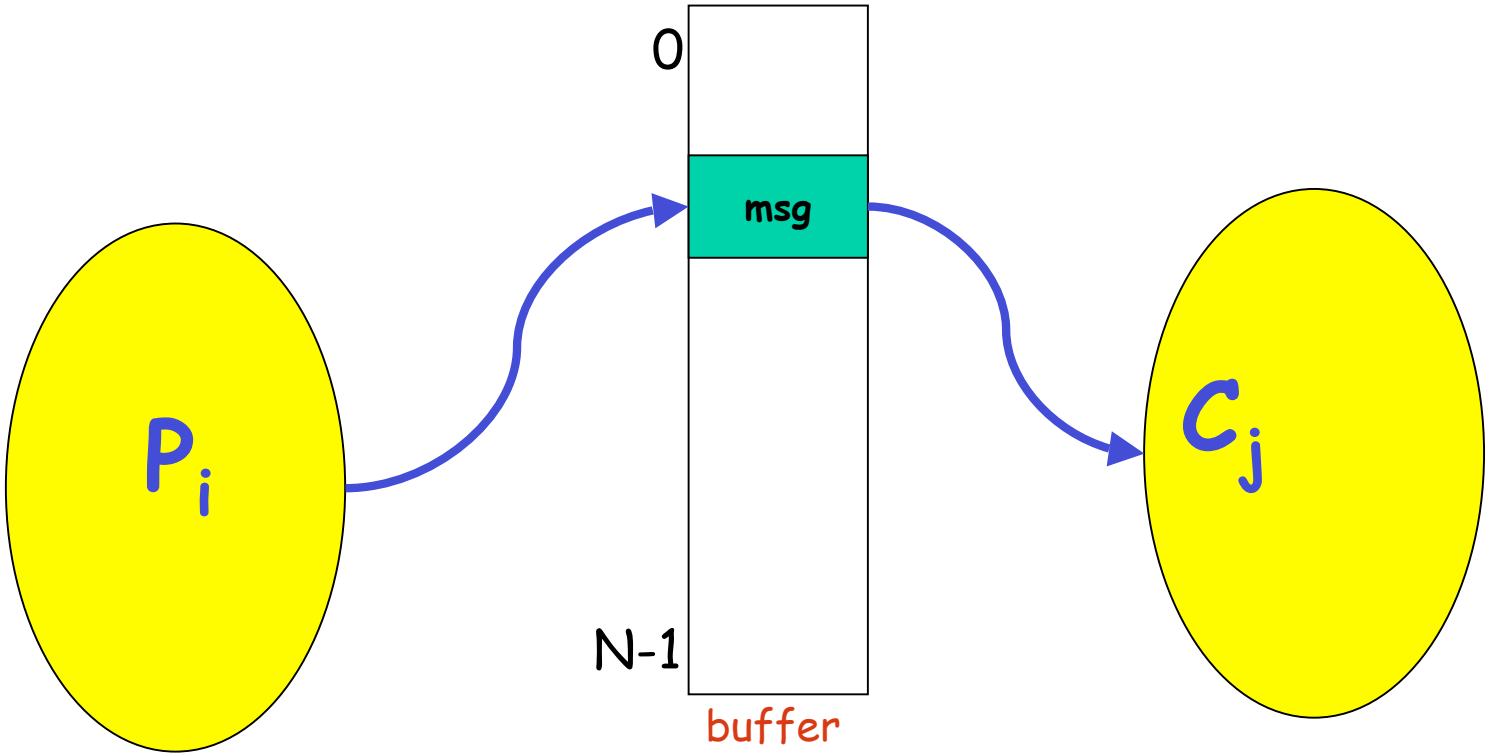
- Le primitive *P* e *V* si possono ottenere dichiarandole come **syncronized methods** all'interno della classe *semaforo*.

```
public class semaforo {
  private int value;
  public semaforo (int initial){
    this.value = initial;
  }

 synchronized public void v()
{    ++value;
    notify();
}

synchronized public void p() throws
InterruptedException
 {    while(value == 0) {
        try      { wait();
           } catch (InterruptedException e) {}
      }
      value--;
   }
}
```

Esempio: produttori e consumatori con buffer di capacità N



HP: Buffer (mailbox) limitato di dimensione N

```
public class threadP extends Thread{ //produttori
   int i=0;
   risorsa r; //oggetto che rappresenta il buffer condiviso

   public  threadP(risorsa R)
   {    this.r=R;
   }

   public void run()
   {    try{   System.out.print("\nThread PRODUTTORE: il mio
               ID è: "+getName()+"..\n");
               while (i<100)
               { sleep(100);
                 r.inserimento(i);
                 i++;
                 System.out.print("\n"+ getName() +":
         inserito messaggio " +i+ "\n");
               }
       }catch(InterruptedException e){}
   }
}
```

```
public class threadC extends Thread{ //consumatori
   int msg;
   risorsa r;

   public  threadC(risorsa R)
   {    this.r=R;
   }

   public void run()
   {    try{   System.out.print("\nThread CONSUMATORE: il mio
               ID è: "+getName()+"..\n");
               while (true)
               {      msg=r.prelievo();
                      System.out.print("\n"+getName()+"
                      consumatore ha letto il messaggio "+msg
                      + "...\n");
               }
        }catch(InterruptedException e){}
   }
}
```

```
public class risorsa { // definizione buffer condiviso
   final int N = 30; // capacità del buffer
   int inseriti; //numero di messaggi dentro il buffer
   int lettura, scrittura;//indice di lettura
   int []buffer;
   semaforo sP; /* sospensione dei Produttori; v.i. N*/
   semaforo sC; /* sospensione dei Consumatori v.i. 0*/
   semaforo sM; // semaforo di mutua esclusione v.i. 1

   public  risorsa() // costruttore
   {    inseriti=0;
        lettura=0;
        scrittura=0;
        buffer= new int[N];
        sP=new semaforo(N); /* v.i. N*/
        sC=new semaforo(0); /* v.i. 0*/
        sM=new semaforo(1); // semaforo di mutua esclusione
   }  //continua..
```

Semafori risorsa

```
// ..continua classe risorsa

public void inserimento(int M)
{   try{  sP.p();
          sM.p(); //inizio sez critica
          buffer[scrittura]=M;
          inseriti++;
          scrittura=(scrittura+1)%N;
          sM.v(); //fine sez critica
          sC.v();
    }catch(InterruptedException e){}
}

//continua..
```

```
//... Continua

public int prelievo()
   {   int messaggio=-1;
       try{  sC.p();
             sM.p(); //inizio sez critica
             messaggio=buffer[lettura];
             inseriti--;
             lettura=(lettura+1)%N;
             sM.v(); //fine sez critica
             sP.v();
       }catch(InterruptedException e){}
       return messaggio;
   }
} // fine classe risorsa
```

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


  public class prodcons{

   public static void main(String args[]) {

      risorsa R = new risorsa();// creaz. buffer
      threadP TP=new threadP(R);
      threadC TC=new threadC(R);

      TC.start();
      TP.start();
   }
}
```

Esempio: la Catena di Montaggio

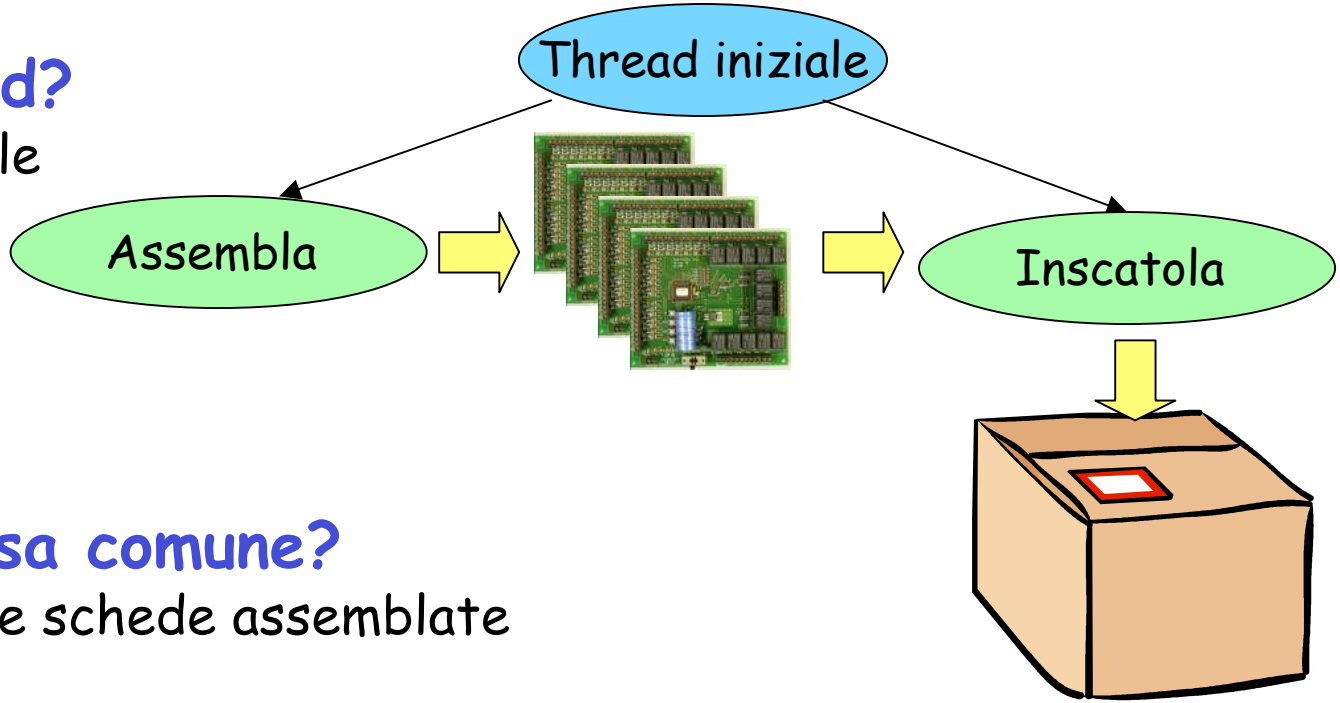
Un'azienda elettronica produce schede a microprocessore. La produzione di ogni scheda avviene in due fasi diverse:
1) **Assemblaggio**: in questa fase avviene l'assemblaggio automatico dei diversi componenti;
2) **Inscatolamento**: le schede assemblate vengono introdotte in scatole di capacita` N.

Si supponga di affidare **ognuna delle 2 fasi a un thread distinto** incaricato di controllare la macchina automatica dedicata alla realizzazione di quella particolare fase.
Utilizzando i **semafori**, si realizzi una politica di sincronizzazione che tenga conto dei seguenti vincoli:
•l'inscatolamento puo` essere attivato soltanto quando N nuovi prodotti sono stati assemblati.
Si supponga inoltre che il thread dedicato all'assemblaggio non possa effettuare una nuova fase di assemblaggio se vi sono ancora **MAX** schede da inscatolare (MAX > N).
La soluzione dovra` consentire un soddisfacente grado di concorrenza tra i threads.

Spunti e suggerimenti (1)

Quali thread?

- thread iniziale
- **Assembla**
- **Inscatola**



Quale risorsa comune?

- l'insieme delle schede assemblate

Il thread iniziale crea i 2 thread ***Assembla***, ***Inscatola***, ognuno con struttura ciclica:

- **Assembla**: al termine di un assemblaggio, soltanto dopo aver confezionato l'N-esimo prodotto, il thread Assembla attiva il thread *Inscatolamento*; puo` sospendersi se ci sono MAX schede ancora da inscatolare.
- **Inscatola**: una volta attivato provvede ad eseguire la fase di inscatolamento.

Spunti e suggerimenti (2)

Sincronizzazione:

- mutua esclusione nell'accesso alle risorse condivise (assemblati):
  - definiamo un semaforo di mutua esclusione sM
- sospensione del processo che Assembla:
  - *definiamo un semaforo risorse sA (v.i.=max)*
- sospensione del processo che Inscatola:
  - *definiamo un semaforo risorsa sI (v.i.=0)*

```
public class threadA extends Thread{ // def. Assemblatore
    semaforo sA, sI, sM;
    int i=0;
    risorsa r;

    public  threadA(risorsa R)
    {     this.r=R;
    }

    public void run()
    {     try{
          System.out.print("\nThread ASSEMBLAGGIO: il mio ID
                                        è: "+getName()+"..\n");
          while (i<30)
                    {         sleep(100);
                              r.nuovoA();
                              i++;
                              System.out.print("\n"+ getName() +":
                                        nuovo assemblato...." +i+ "\n");
                    }
          }catch(InterruptedException e){}

    }}
```

```
    public class threadS extends Thread{ //thread che inscatola
    semaforo sA, sI, sM;
    int i, scatole=0;
    risorsa r;

    public  threadS(risorsa R)
    {     this.r=R;}

    public void run()
    {     try{     System.out.print("\nThread INSCATOLAMENTO:
                           il mio ID è: "+getName()+"..\n");
                  while (true)
                  {        r.nuovaS();
                           sleep(100);/*durata inscatolamento...*/
                           scatole++;
                           System.out.print("\n"+getName()+"
                                    inscatolamento "+scatole + ...\n");
                  }
          }catch(InterruptedException e){}
    } //chiude run
}
```

```
public class risorsa {
        final int max=15;
        final int N = 4; // capacità della scatola
        int pronti; // assemblati pronti (al massimo MAX)
        int i;
        semaforo sA; // per la sospensione di TA; v.i max
        semaforo sI; // v.i. 0 per la sospensione di TS
        semaforo sM; // semaforo di mutua esclusione

   public  risorsa(int pronti)
   {    sA=new semaforo(max);     //rappresenta lo spazio
                                  //disponibile
        sI=new semaforo(0);       // rappresenta il numero di
                                  //scatole che possono essere
                                  //confezionate
        sM=new semaforo(1);       // mutua esclusione
        pronti =0;
   }
```

```
public void nuovoA()//deposito assemblato
   {    try{    sA.p();
                sM.p(); //inizio sez critica
                pronti=pronti+1;
                if (pronti%N==0)
                        sI.v();
                sM.v(); //fine sez critica
        }catch(InterruptedException e){}
   }

public void nuovaS() //preleva N assemblati
   {    try{    sI.p();
                sM.p();
                pronti=pronti-N;
                for(i=0; i<N; i++)
                        sA.v();
                sM.v();
        }catch(InterruptedException e){}
   }

}
```

```
public class supplychain{

 public static void main(String args[]) {

      risorsa R=new risorsa(0);
      threadA TA=new threadA(R);
      threadS TS=new threadS(R);

      TA.start();
      TS.start();

              }

}
```

Java Thread: Alcune considerazioni al contorno: i problemi di `stop()` e `suspend()`

stop()

- forza la terminazione di un thread
- tutte le risorse utilizzate vengono immediatamente liberate (lock compresi)

Se il thread interrotto stava compiendo un insieme di operazioni da eseguirsi in maniera atomica, l'interruzione può condurre ad uno stato inconsistente del sistema

I problemi di `stop()` e `suspend()`

suspend()

- blocca l'esecuzione di un thread, in attesa di una successiva invocazione di resume()
- non libera le risorse impegnate dal thread (non rilascia i lock)

Se il thread sospeso aveva acquisito una risorsa in maniera esclusiva (ad esempio sospeso durante l'esecuzione di un metodo synchronized), tale risorsa rimane bloccata

Alcune considerazioni al contorno: priorità dei thread

La classe **`Thread`** fornisce i metodi:

- **`setPriority(int num)`**
- **`getPriority()`**

dove num ∈ [MIN_PRIORITY,MAX_PRIORITY]

In generale un thread rimane in esecuzione fino a quando
(vedi scheduling JVM – prossima settimana):

- ***smette di essere runnable*** (Not Runnable o Dead)
- un thread con ***priorità superiore diviene eseguibile***
- si ***esaurisce il quanto di tempo*** assegnato (time slicing non garantito)
- cede il controllo ***chiamando il metodo `yield()`***

Priorità dei thread

Le specifiche JVM ***suggeriscono*** che

- vengano messi in esecuzione per primi i thread a ***priorità più elevata*** tra quelli in ***stato runnable***
- si proceda secondo una ***politica round-robin*** tra i thread con **identica priorità**

- Alcune implementazioni di ***JVM delegano lo scheduling dei thread al SO***, ove supportati
- Il comportamento reale diviene dunque ***dipendente dalla coppia JVM e SO***

Altri metodi di interesse per Java thread

- **`sleep(long ms)`**
  - sospende thread per il # di ms specificato
- **`interrupt()`**
  - invia un evento che produce l'interruzione di un thread
- **`interrupted()/isInterrupted()`**
  - verificano se il thread corrente è stato interrotto
- **`join()`**
  - ***attende la terminazione del thread specificato***
- **`isAlive()`**
  - true se thread è stato avviato e non è ancora terminato
- **`yield()`**
  - ***costringe il thread a cedere il controllo della CPU***