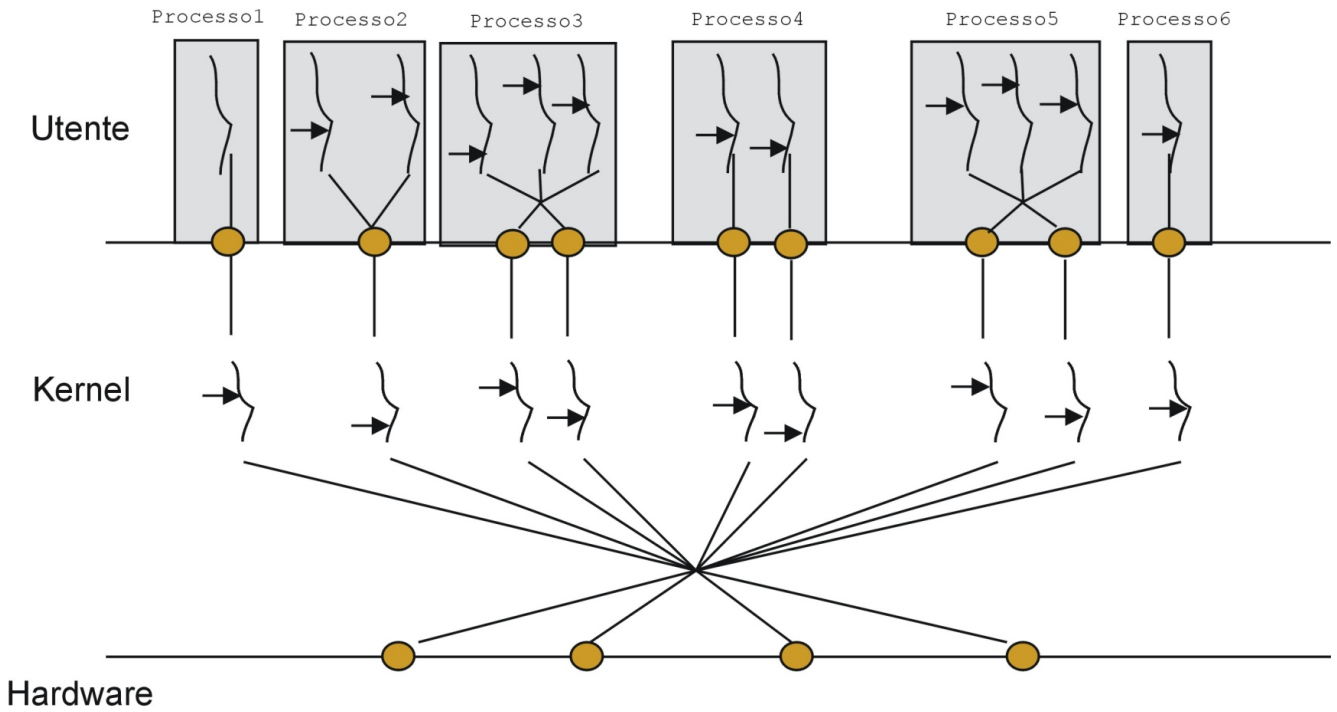
Mutua esclusione

Concorrenza

- Ci sono due tipi di programmazione concorrente:
  1. la programmazione concorrente a **processi concorrenti**;
  2. la programmazione concorrente a **thread concorrenti** ('94).
- Vantaggi dei thread:
  - schedulazione molto piú efficiente dei processi;
  - il blocco di un thread non provoca il blocco di un intero processo;
  - i thread vedono lo stesso ambiente, quindi i dati vengono visti da tutti i thread. Questo vuol dire che la comunicazione tra thread é in generale piú agevole. Bisogna stare attenti alle interferenze.
- Tipi di threads:
  - Da molti (thread utente) a uno (thread sistema) (Solaris2)
  - Da uno a uno (come Linux, WinNT, Win2000)
  - Da molti a molti (come Solaris2, Irix, HP-UX)

Una visione dei threads



Riepilogo: la Concorrenza in Java

```
class classeA extends Thread {
    public void run(){
       while(true)
          System.out.println("Sono la classe A");
    }
}
class classeB extends Thread {
    public void run(){
       while(true)
          System.out.println("Sono la classe B");
    }
}
public class main {
    public static void main(String argv[]){
      classeA A = new classeA(); // istanzia
      classeB B = new classeB();
      A.start(); A.join();     // esegue
      B.start(); B.join();



    }
}
[CUT]
Sono la classe B
Sono la classe A
Sono la classe B
Sono la classe A
Sono la classe B
```

```
class classeA implements Runnable {
   public void run(){
     while(true)
       System.out.println("Sono la classe A");
   }
}
class classeB implements Runnable {
   public void run(){
     while(true)
       System.out.println("Sono la classe B");
   }
}
public class main {
   public static void main(String argv[]){
     classeA A = new classeA(); // istanzia
     classeB B = new classeB();
     Thread t1=new Thread(A);
     t1.start(); t1.join();
     Thread t2=new Thread(B);
     t2.start(); t2.join();
    }
 }
 [CUT]
 Sono la classe B
 Sono la classe A
 Sono la classe B
 Sono la classe A
 Sono la classe B
```

Riepilogo: condivisione variabili tra thread

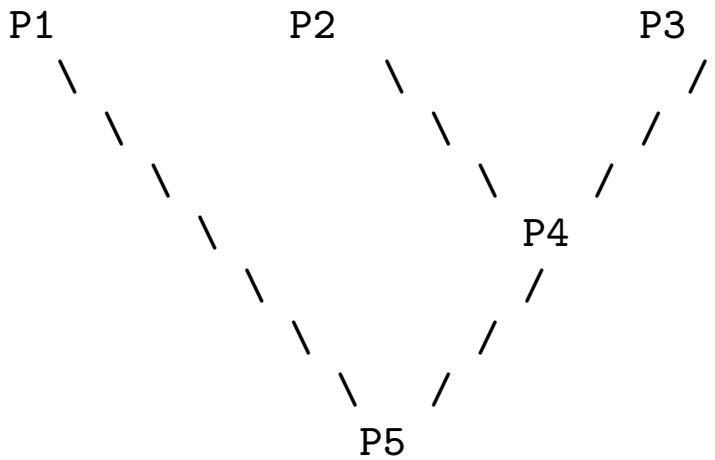
```
public class z{
  float b[]= new float[10];
  void put(int i, float f){ b[i]=f; }
     float get(int i){ return(float)b[i]; }
}
public class pi extends Thread{
  z buf;
  pi(z buf){ this.buf=buf;} //costruttore
  public void run(){
   while(true) {
      try{ Thread.sleep(800);}
        catch(InterruptedException e) {}
      System.out.print("leggo ");
      for (int i=0; i<5; i++)
         System.out.print( "+buf.get(i));
      System.out.println();
   }
  }
}
```

```
public class po extends Thread{
  z buf; Random r=new Random();
  po(z buf){ this.buf=buf; } //costruttore
  public void run(){
    while(true) {
      try{ Thread.sleep(990);}
        catch (InterruptedException e){}
      System.out.print("\tscrivo ");
      for(int i=0; i<5; i++) {
         buf.put(i,r.nextFloat());
         System.out.print(" "+buf.get(i));
      }
    }
  }
}
```

```
public class pth{
  public static void main(String[] a){
    z buf=new z();
    pi c=new pi(buf); po t=new po(buf); c.start(); t.start();
  }
}
```

Riepilogo: metodi start/join

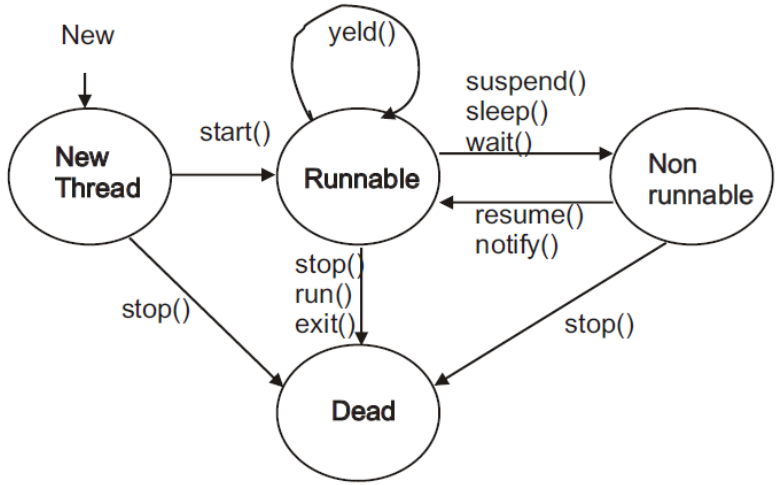
- Valutazione multithread dell'espressione $3 * a^3 + (2 * b^3) * (3 * d^3)$
- dove $P1 \rightarrow x = 3 * a^3$, $P2 \rightarrow y = 2 * b^3$, $P3 \rightarrow z = 3 * d^3$, $P4 \rightarrow t = y * z$, $P5 \rightarrow k = x + t$
- Il grafo delle precedenze si puo' rappresentare nel modo seguente



Riepilogo: uso dei metodi start/join

```
\\codice
p1.start(); p2.start(); p3.start();
try { p2.join(); } catch(InterruptedException e)
                     {System.out.println("Errore P2");}
try { p3.join(); } catch(InterruptedException e)
                     {System.out.println("Errore P3");}
p4.start();
try { p1.join(); } catch(InterruptedException e)
                     {System.out.println("Errore P1");}
try { p4.join(); } catch(InterruptedException e)
                     {System.out.println("Errore P4");}
p5.start();
try { p5.join(); } catch(InterruptedException e)
                     {System.out.println("Errore P5");}
p1.stop();p2.stop(); p3.stop(); p4.stop();p5.stop();
System.out.println("  Fine programma ");
```

Stati di un thread java



Alcuni metodi della classe Thread:

- public void start() //lancia il thread
- public void join() //aspetta la terminazione
- public void run() //esegue il codice
- public final void stop() //termina il thread
- public final void suspend() //sospende il thread
- public final void resume() //riattiva il thread
- public static void yield() //rischedula
- public final native boolean isAlive() //esce con true se il thread vivo

Stati di un thread java

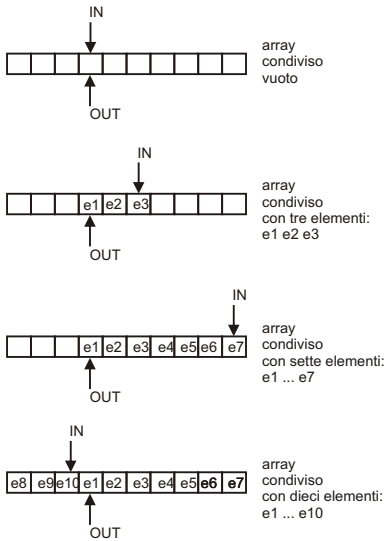
Un Thread di Java inizia chiamando il metodo start() e termina con il metodo stop().

- Stato NonRunnable: il thread é sospeso, cosa che puó avvenire per: attesa del completamento di una operazione di I/O o del periodo di una sleep(), chiamata del metodo suspend() o wait().
- Viceversa il risveglio puó aversi per la fine della operazione di I/O o dell'intervallo di sleep(), chiamata dei metodi notify(), notifyAll() o resume().

La schedulazione dei thread é a prioritá, che é un valore che inizia con quello del thread padre ma puó essere variata con il metodo setPriority().

Un problema di mutua esclusione

Si consideri un array circolare condiviso tra un thread produttore e due consumatori: i consumatori devono prelevare elementi consecutivi senza ripetizioni o cancellazioni




```
prod(){
 while(true)
 {
    el=produci();
    while((IN+1)%N==out)){};
    buf[in]=el;
    in=(in+1)%N;
  }
}
```

```
cons() {
 while(true)
 {
    while(in==out){};
    el=buf[out];
    out=(out+1)%N;
    consuma(el);
  }
}
```

```
cons1() {
   while(true)
   {
      while(in==out)();
      el=buf[out];
      out=(out+1)%N;
      consuma(el);
    }
 }
```

```
prod(){
 while(true)
 {
    el=produci();
    while((IN+1)%N==out)){};
    buf[in]=el;
    in=(in+1)%N;
  }
}
```

```
cons() {
  while(true)
  {
     while(in==out){};
     el=buf[out];
     out=(out+1)%N;
     consuma(el);
  }
}
```

```
cons1() {
   while(true)
   {
      while(in==out)();
      el=buf[out];
      out=(out+1)%N;
      consuma(el);
    }
 }
```

Questo codice peró non funziona. Si pensi ad esempio a un context switch tra i due consumatori:

```
cons()
{
   while(true)
    {
       while(in==out){};
       el=buf(out);
       out=(out+1)%N;
       consuma(el);
     }
}
```

context switch!

```
cons1()
{
   while(true)
    {
       while(in==out){};
       el=buf(out);
       out=(out+1)%N;
       consuma(el);
     }
}
```

Si assuma che ci sia 1 elemento nel buffer. Il secondo consumatore lo preleva e incrementa OUT: il primo consumatore legge un dato inesistente. La variabile condivisa che genera il problema é OUT.

Soluzione: eseguire in Mutua Esclusione le sezioni che usano OUT

```
cons() {
  while(true)
  {

      while(in==out){};
    |--------------------|
    |   el=buf[out];     |
    |   out=(out+1)%N;   |
    |--------------------|
     consuma(el);
  }
}
```

```
cons1() {
   while(true)
   {

       while(in==out)();
    |----------------------|
    |    el=buf[out];      |
    |    out=(out+1)%N;    |
    |----------------------|
     consuma(el);
   }
}
```

⇒ Uso un arbitro che controlla l'esecuzione

Soluzioni al problema della Mutua Esclusione

- **Hardware Architetturali Software Linguistiche**

Soluzioni Hardware: disabilitazione degli interrupt

```
while(true){
  disabilita_interrupt();
  sezione_critica;
  abilita_interrupt();
  sezione_noncritica;
}
```

Soluzioni architetturali: basate su Istruzioni Atomiche (vedere gli spinlock)
Due esempi di istruzioni atomiche:
TSL = Test and Set Lock

```
bool TSL(bool *i){
   if(*i==false){*i=true; return false;}
   else {*i=true; return true;}
}                                        }
```

Swap

```
bool Swap (bool *a, bool *b)
{
  bool temp;  temp=*a;  *a=*b;  *b=temp;
}
```

Soluzione software:Primo tentativo

```
Thread1() {
    while(1){
        while( turno == 0 ) ;
        Sezione_Critica1();
        turno = 0;
        Sezione_NON_Critica1();
     }
}
```

```
Thread2(){
      while(1){
        while( turno == 1 ) ;
        Sezione_Critica2();
        turno = 1;
        Sezione_NON_Critica2()
       }
 }
```

Questa soluzione é chiamata 'ALTERNANZA STRETTA CON UNA VARIABILE GLOBALE'. Alcune proprietá di questa soluzione sono:

- soddisfa la muta esclusine
- evita lo stallo tra i processi
- evita la starvation

Problema: Il problema di questa soluzione é che un blocco o un rallentamento in una sezione critica blocca o rallenta l'altro processo.

Soluzione software:Secondo tentativo

```
Thread1() {
    while(1){
        while( turno2 == 0 ) ;
        turno1 = 0;
        Sezione_Critica1();
        turno1 = 1;
        Sezione_NON_Critica1();
     }
}
```

```
Thread2(){
      while(1){
        while( turno1 == 0 ) ;
        turno2 = 0;
        Sezione_Critica2();
        turno2 = 1;
        Sezione_NON_Critica2()
       }
 }
```

Le variabili sono inizializzate come segue: turno1=0, turno2=1. Proprietá:

- se una sezione critica si blocca, non viene influenzato l'altro processo
- non c'é stallo
- non c'é starvation

Ma: i due processi possono trovarsi simultaneamente nella sezione critica!!.

Soluzione software: Terzo tentativo

```
Thread1() {
    while(1){
        turno1 = 0;
        while( turno2 == 0 ) ;
        Sezione_Critica1();
        turno1 = 1;
        Sezione_NON_Critica1();
     }
}
```

```
Thread2(){
      while(1){
        turno2 = 0;
        while( turno1 == 0 ) ;
        Sezione_Critica2();
        turno2 = 1;
        Sezione_NON_Critica2()
       }
 }
```

Le variabli sono inizializzate come nel caso precedente. Ma: in questa soluzione i processi possono trovarsi in stallo.

Soluzione software: Quarto tentativo

```
Thread1() {
  while(1){
    C1 = 1;
    turno = 0;
    while( C2==1 && turno == 0 );
    Sezione_Critica1();
    C1 = 0;
    Sezione_NON_Critica1();
   }
}
```

```
Thread2(){
  while(1){
    C2 = 1
    turno = 1;
    while( C1==1 && turno == 1 );
    Sezione_Critica2();
    C2 = 0;
    Sezione_NON_Critica2();
   }
}
```

Principali proprietá:

- soddisfa a tutte le richieste (cioé mutua esclusione, no starvation, no stallo)
- la soluzione é estendibile a piú processi

Nota: il test **while(C2 && turno == 0) ;** si puó anche scrivere **while(C2 $\geq$ C1 && turno == 0);** .
Analogamente **while(C1 && turno == 1) ;** si puó anche scrivere **while(C1 $\geq$ C2 && turno == 1) ;** .

```
Strutture dati inizializzate a  0

Thread P1              Thread P2              Thread P3

C1=1;                  C2=1;                  C3=1;
T1=1;                  T1=2;                  T1=3;
while((C2 >= C1        while((C1 >= C2        while((C1 >= C3
    || C3 >= C1)           || C3 >= C2)           || C2 >= C3)
   && T1 == 1);           && T1 == 2);           && T1 == 3);

C1=2;                  C2=2;                  C3=2;
T2=1;                  T2=2;                  T2=3;
while((C2 >= C1        while((C1 >= C2        while((C1 >= C3
   || C3 >= C1)           || C3 >= C2)           || C2 >= C3)
   && T2 == 1);           && T2 == 2);           && T2 == 3);

SezioneCritica0;       SezioneCritica1;       SezioneCritica2;

C1=0;                  C2=0;                  C3=0;
```

Pseudocodice algoritmo di Peterson per N Thread

```
Processo i-esimo

void lock(int i){
  for(j=1;j<N-1;j++){
     C[i] = j;
     T[j] = i;
     for(k=1;(k<=n)&&(k!=i);k++)
        while(C[k] >= C[i] && T[j] == i);
  }
}

void unlock(int i){
   C[i] = 0;
}
```

Codice Java

```
public class MyThread extends Thread{
    Data data;
    float a, b, c, d;
    public MyThread1(Data data){
       this.data = data; //dati condivisi
    }
    public void run() {
      data.c1 = 1; data.t1 = 1;
      while(true){
        while((data.c2 >= data.c1 || data.c3 >= data.c1) && data.t1 == 1)
           try{sleep(10);} catch(InterruptedException e){}; //prima attesa attiva
        System.out.println("Sono il "+Thread.currentThread().getName()+" id: "
                           +Thread.currentThread().getId()+"  c1 c2 c3 t1: "
                           +data.c1+data.c2+data.c3+data.t1);
        data.c1 = 2; data.t2 = 1;
        while((data.c2 >= data.c1 || data.c3 >= data.c1) && data.t2 == 1)
            try{sleep(10);} catch(InterruptedException e){}; //seconda attesa attiva
        //sezione critica SC
        System.out.println("Sono "+Thread.currentThread().getName()+"Entro in SC. c1 c2 c3 t1: "
                            +data.c1+data.c2+data.c3+data.t2);
        data.condivisa++;
        try{sleep(300);} catch(InterruptedException e){};
        System.out.println("Sono il "+Thread.currentThread().getName()+"Valore="+data.condivisa);
        //fine sezione critica SC
        System.out.println("Sono il "+Thread.currentThread().getName()+" esco dalla SC");
        data.c1 = 0;
        a=(b+c)*d;//sezione non critica
      }
    }
```

Algoritmo di Lamport

L'algoritmo del Peterson é stato pubblicato nel 1981. Semplifica una soluzione software pre-esistente, sviluppata da Lamport nel 1974.
IDEA di Lamport: assegnare un numero (ticket) ai processi e far avanzare quello con ticket minore.
Esempio con 2 processi: C1 e C2 sono inizializzati a zero.

```
Processo P1                    Processo P2
C1=1;                          C2=2;
while(C2 != 0 && C2 < C1);     while(C1 != 0 && C1 < C2);
SezioneCritica1;               SezioneCritica2;
C1=0;                          C2=0;
Sezione Non Critica;           Sezione Non Critica;
```

Il processo P2 aspetta perche' C1 $<$ C2 finché C1 non diventa 0, mentre P1 avanza.
Ma se c'e' un context switch durante la assegnazione C1=1? la mutua esclusione viene negata!!
Bisogna assegnare i valori di C1 e C2 in mutua esclusione: introduco le variabili T1 e T2.

```
Processo P1                    Processo P2
T1=1;                          T2=1;
C1=1;                          C2=2;
T1=0;                          T2=0;
while(T2);                     while(T1);
while(C2 != 0 && C2 < C1);     while(C1 != 0 && C1 < C2);
SezioneCritica1;               SezioneCritica2;
C1=0;                          C2=0;
Sezione Non Critica;           Sezione Non Critica;
```

Algoritmo di Lamport: Assegnazione del ticket

```
Processo P1                      Processo P2
T1=1;                            T2=1;
C1=max(C1,C2)+1                  C2=max(C1,C2)+1;
T1=0;                            T2=0;
while(T2);                       while(T1);
while(C2 != 0 && C2 < C1);       while(C1 != 0 && C1 < C2);
SezioneCritica1;                 SezioneCritica2;
C1=0;                            C2=0;
Sezione Non Critica;             Sezione Non Critica;
```

Ma: ogni processo deve leggere i valori di Ci in mutua esclusione! Potrebbe essere C1==C2.
Allora, introduco un operatore di confronto tra coppie di interi: (a,b) $<$ (c,d) se a $<$ c e, se a $=$ c, b $<$ d.

```
Processo P1                            Processo P2
T1=1;                                  T2=1;
C1=max(C1,C2)+1                        C2=max(C1,C2)+1;
T1=0;                                  T2=0;
while(T2);                             while(T1);
while(C2 != 0 && (C2,2)<(C1,1));       while(C1 != 0 && (C1,1)<(C2,2));
SezioneCritica1;                       SezioneCritica2;
C1=0;                                  C2=0;
Sezione Non Critica;                   Sezione Non Critica;
```

Algoritmo di Lamport per N threadi

```
// valori iniziali variabili condivise
  T: array [1..N] of bool = {false};
  C: array [1..N] of integer = {0}; //tickets
  lock(integer i) {
     T[i] = true;
     C[i] = 1 + max(C[1], ..., C[N]);
     T[i] = false;
     for (j = 1; j <= N; j++) {
         // Attendi finche' il thread j riceve il suo ticket
         while (T[j]) {};
         // Attendi la fine dei thread con ticket minore:
        while ((C[j] != 0) && ((C[j], j) < (C[i], i))) {};
    }
}
unlock(integer i) {
    C[i] = 0;
}
Thread(integer i) {
    while (true) {
        lock(i);
        // Sezione critica
        unlock(i);
        // Sezione NON critica
    }
}
```

Algoritmo di Lamport per 3 thread

```
public class VariabileCondivisa
{
    static volatile private boolean [] T= {false, false, false, false};
    static volatile private int [] C= {0, 0, 0, 0};
    static volatile private int variabile=0;

    static public int getVariabile() { return variabile; }
    static public void incrementaVariabileCondivisa() { variabile=variabile+1;}

    static public void lock(int i)
    {
        T[i]=true;
        int max=0;
        for(int j=1;j<=3;j++){ if(C[j]>=max) max=C[j]; }

        C[i]=1+max;
        T[i]=false;

        for(int j=1;j<=3;j++)
        {
            while(T[j]);
            while( ( C[j]!=0 )  &&  ( ( C[j]<C[i]  ) || (( C[j] == C[i] ) && (j<i) )) ) ;
                try{
                    Thread.currentThread().sleep(1);//sleep for 1000 ms
                }
                catch(InterruptedException ie){
            }
         }
```

Altra soluzione produttore/consumatore

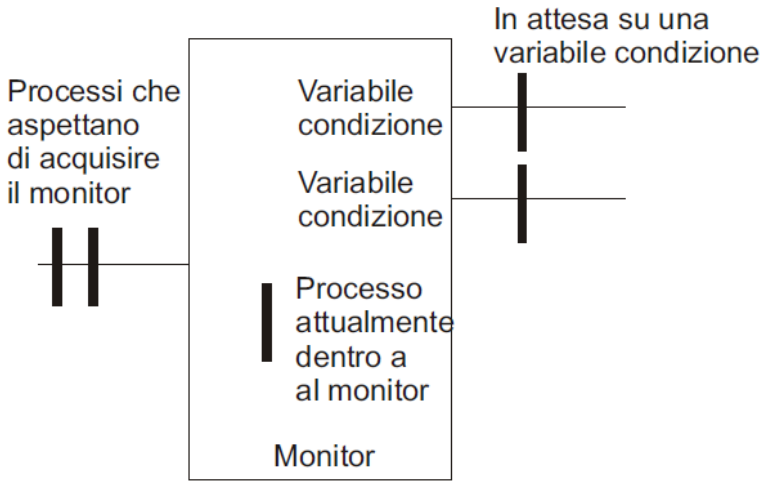
In questo caso usiamo primitive SLEEP() e WAKEUP() che rispettivamente sospendono e risvegliano il processo.

```
void Prod() {
  while(true){
    el=produci();
    while(n==N) sleep();   //se il buffer e' pieno, aspetta
    A[n]=el; n++;
    if(n==1) wakeup(Cons); //1 elemento: sveglia il consumatore
  }
}
void Cons() {
  while(true){
    while(n==0) sleep();   //se il buffer e' vuoto, aspetta
    el=A[n]; n--;
    if(n==N-1) wakeup(Prod); //1 posto libero: sveglia il produttore
  }
}
```

Il problema nasce dal fatto che puó esserci un context switch per esempio prima di sleep() provocando la perdita del risveglio

Una primitiva linguisica fondamentale: MONITOR

- Monitor: costrutto linguistico simile alla definizione di classe, che associa a delle variabili condivise le procedure che le utilizzano.
- Garantisce che i metodi siano usati in mutua esclusione: solo una procedura attiva all'interno del Monitor.
- Sincronizzazione tra i metodi realizzata mediante variabili condizione, (p.e. var_cond) e operatori it var_cond.wait, var_cond.signal.
- var_cond.wait sospende il processo sulla variabile condizione var_cond e rilascia la mutua esclusione per consentire ad altri di modificare le variabili.
- var_cond.signal risveglia processi sospesi su var_cond.
- In definitiva i Monitor si presentano come strutture di variabili interne al monitor, di procedure private, procedure pubbliche (entry), variabili condizione, code d'attesa:



Soluzione Prod/cons con i MONITOR

```
monitor prodcons
  condition pieno, vuoto;

  entry Prod() {
    while(true){
      el=produci();
      while(n==N) pieno.wait;   //se il buffer e' pieno, aspetta
      A[n]=el; n++;
      if(n==1) vuoto.signal; //1 elemento: sveglia il consumatore
    }
  }

  entry Cons() {
    while(true){
      while(n==0) vuoto.wait;   //se il buffer e' vuoto, aspetta
      el=A[n]; n--;
      if(n==N-1) pieno.signal;//1 posto libero: sveglia il produttore
    }
  }
end monitor;
```

Il MONITOR di Java

- **metodi *synchronized*: se un metodo possiede questo attributo cerca di acquisire una proprietá della classe prima di eseguire.**
- **Garantisce che i metodi synchronized siano usati in mutua esclusione: solo un metodo synchronized attivo all'interno della classe.**
- **Metodo wait() ereditato da tutte le classi: rilascia il blocco e mette il thread chiamante in coda d'attesa. Deve essere usata solo dai metodi synchronized altrimenti scatta una eccezione.**
- **Metodo Notify(): libera un thread tra quelli in attesa.**
- **Metodo NotifyAll(): libera tutti i thread in attesa.**

Il Monitor in Java

```
public class MonitorProdcons{
    int n=0,N=10; int A[] = new int[20];
    int el=0;

    public synchronized void Prod() {
      while(true){
        el=produci();
        try{ Thread.sleep(1); } catch(InterruptedException e) {}

        while(n==N) //se il buffer e pieno, aspetta
          try {wait();} catch (InterruptedException e) {}
        A[n]=el; n++;

        if(n==1) notify(); //1 elemento: sveglia
      }
    }

    public synchronized void Cons() {
      while(true){
        while(n==0) //se il buffer e vuoto, aspetta
          try {wait();} catch (InterruptedException e) {}
        el=A[n]; n--;

        if(n==N-1) notify(); //1 posto libero: sveglia
        return el;
      }
    }
}
```

I SEMAFORI

Dijkstra introduce due operazioni,**down() e up()**, che operano su una variabile intera *s*.
In principio:

```
down(s)
{   while (s<=0); /* loop di attesa: BUSY WAITING */
    s--;
}
up(s)
{ s++; }
```

Ma: le operazioni DEVONO ESEGUIRE ATOMICAMENTE!

I SEMAFORI SPIN LOCK

In pratica:

```
Usando TSL
(s condivisa)
down(s)
{
  while (TSL(&s));
}


up(s)
{ s=0; }
```

oppure:

```
Usando Swap
(s condivisa, b locale)
down(s)

{  b=1;
   while(b==1)
     Swap(s,b);//busy waiting
}
up(s)
{ s=0; }
```

ATTENZIONE: in questi casi non sono le down, up ad essere atomiche, ma le TSL o Swap!
Protezione di una sezione critica:

```
Processo1(){
   while(1){
      down(s);
      Sezione_Critica1();
      up(s);
      Sezione_NON_Critica1();
   }
}
```

```
Processo2(){
   while(1){
      down(s) ;
      Sezione_Critica2();
      up(s);
      Sezione_NON_Critica1();
   }
}
```

```
Processo2(){
   while(1){
      down(s) ;
      Sezione_Critica2();
      up(s);
      Sezione_NON_Critica1();
   }
}
```

I SEMAFORI WAIT LOCK

Per attese lunghe (*wait locks*) **si introduce una coda d'attesa su s**:

```
down(s)
{
   if(s<=0)
       aggiungi il descrittore del processo in coda su s;
   else s--;
 }

up(s)
{
  if(c'e' un descrittore in attesa su s)
      esegui il processo;
  else
      s++;
 }
```

Le wait lock sono piu' lente ma adatte ad attese lunghe.
Anche in questo caso, la protezione di una sezione critica:

```
Processo1(){
   while(1){
      down(s);
      Sezione_Critica1();
      up(s);
      Sezione_NON_Critica1();
   }
}
```

```
Processo2(){
   while(1){
      down(s) ;
      Sezione_Critica2();
      up(s);
      Sezione_NON_Critica1();
   }
 }
```

```
Processo2(){
   while(1){
      down(s) ;
      Sezione_Critica2();
      up(s);
      Sezione_NON_Critica1();
   }
}
```

I SEMAFORO BINARIO in Java

```
class SemaforoBinario {
    int value;
    public Semaforo() //costruttore
    {       value = 1;    }
    public Semaforo(int value) //costruttore
    {          this.value = value;    }

    public synchronized void down()
    {
       if (value == 0)
       { try{wait();} catch(InterruptedException e) { }; }
       value=0;
    }

    public synchronized void up()
    {
       value=1;
       notify();
    }
}
```

I SEMAFORI CONTATORI in Java

```
class SemaforoContatore {
    int value;
    public Semaforo() //costruttore
    {       value = 1;    }
    public Semaforo(int value) //costruttore
    {          this.value = value;    }

    public synchronized void down()
    {
       while (value<=0)
       { try{wait();} catch(InterruptedException e) { }; }
       value--;
    }

    public synchronized void up()
    {
       value++;
       notify();
    }
}
```