Il Monitor

Monitor

- Definizione del monitor
- Esempi d'uso
- Realizzazione del costrutto monitor
- Realizzazione di politiche di gestione delle risorse
- Chiamate innestate a procedure del monitor
- Realizzazione del monitor con Pthread e Java

Il costrutto monitor

Definizione: Costrutto sintattico che associa un insieme di operazioni (*public* o *entry*) ad una struttura dati comune a più processi.

- Le operazioni entry **sono le sole operazioni permesse** su quella struttura.

- Le operazioni entry sono **mutuamente esclusive**: un solo processo per volta può essere attivo nel monitor.

```
monitor tipo_risorsa {
      <dichiarazioni variabili locali>;
      <inizializzazione variabili locali>;

      public void op1 ( ) {
      <corpo della operazione op1 >;
      }
      ----------
      public void opn ( ) {
      <corpo della operazione opn>;
      }
      <eventuali operazioni non public>
}
```

- Le **operazioni public** (o entry) sono **le sole operazioni** che possono essere utilizzate dai processi per accedere alle variabili locali. L'accesso avviene in **modo mutuamente esclusivo**.

- Le **variabili locali** mantengono il loro valore tra successive esecuzioni delle operazioni del monitor (variabili permanenti).

- Le variabili locali sono **accessibili solo entro il monitor**.

- Le operazioni **non** dichiarate **public** non sono accessibili dall'esterno. Sono usabili solo all'interno del monitor (dalle funzioni public e da quelle non public).

Uso del monitor

```
tipo_risorsa ris;
```

- crea una istanza del monitor, cioè una struttura dati organizzata secondo quanto indicato nella dichiarazione dei dati locali.

```
ris.opi(...);
```

- chiamata di una generica operazione dell'oggetto **`ris`**.

Uso del monitor

Scopo del monitor è controllare l'assegnazione di una risorsa tra processi concorrenti in accordo a determinate politiche di gestione. Le variabili locali definiscono lo stato della risorsa associata al monitor.

L'assegnazione della risorsa avviene secondo **due livelli di controllo**:

1. Il **primo** garantisce che un solo processo alla volta possa aver accesso alle variabili comuni del monitor. Ciò è ottenuto garantendo che le operazioni *public* siano eseguite in modo **mutuamente esclusivo** (eventuale sospensione dei processi nella ***entry queue***).
2. Il **secondo** controlla **l'ordine** con il quale i processi hanno accesso alla risorsa. La procedura chiamata verifica il soddisfacimento di una condizione logica che assicura l'ordinamento (eventuale sospensione del processo in una ***coda associata alla condizione*** e liberazione del monitor).

- Nel caso in cui la condizione non sia verificata, la sospensione del processo avviene utilizzando variabili di un nuovo tipo, detto **condition** (condizione).

- La **condizione di sincronizzazione** è costituita da variabili locali al monitor e da variabili proprie del processo passate come parametri.

Variabili tipo condizione

La dichiarazione di una variabile cond di tipo condizione ha la forma:

```
condition cond;
```

- Ogni variabile di tipo condizione rappresenta una **coda** nella quale i processi si sospendono.

Operazioni sulle variabili condition:

- Le operazioni del monitor agiscono su tali variabili mediante le operazioni:

```
wait(cond);
signal(cond);
```

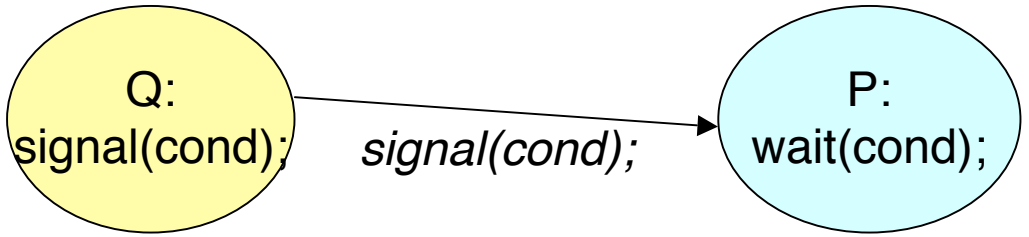
wait:

- L'esecuzione dell'operazione `wait(cond)` sospende il processo, introducendolo nella coda individuata dalla variabile cond, e il monitor viene liberato.

signal:

- L'esecuzione dell'operazione `signal(cond)` rende attivo un processo in attesa nella coda individuata dalla variabile cond.

Semantiche dell'operazione signal



Come conseguenza della signal entrambi i processi, quello segnalante Q e quello segnalato P, **possono concettualmente proseguire la loro esecuzione**.

Possibili strategie:

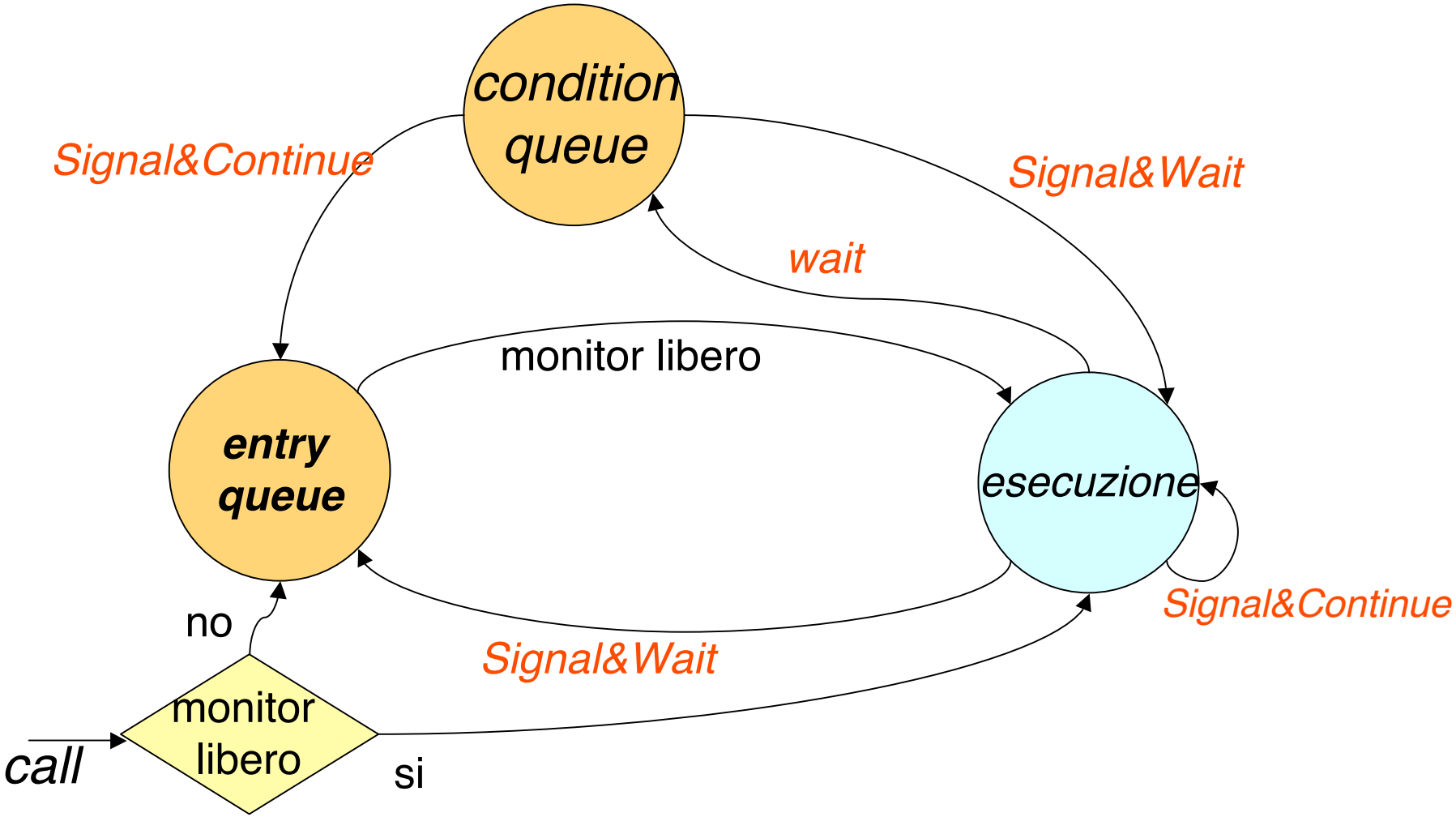
signal_ and_ wait. P riprende immediatamente l'esecuzione ed il processo **Q viene sospeso**.

signal_ and_ continue. **Q prosegue** la sua esecuzione mantenendo l'accesso esclusivo al monitor, dopo aver risvegliato il processo .

- Con la ***signal_ and_ wait*** si evita la possibilità che Q, proseguendo, possa modificare la condizione di sincronizzazione rendendola non più vera per P.

- Q si sospende nella coda dei processi che attendono di usare il monitor (***entry queue***).

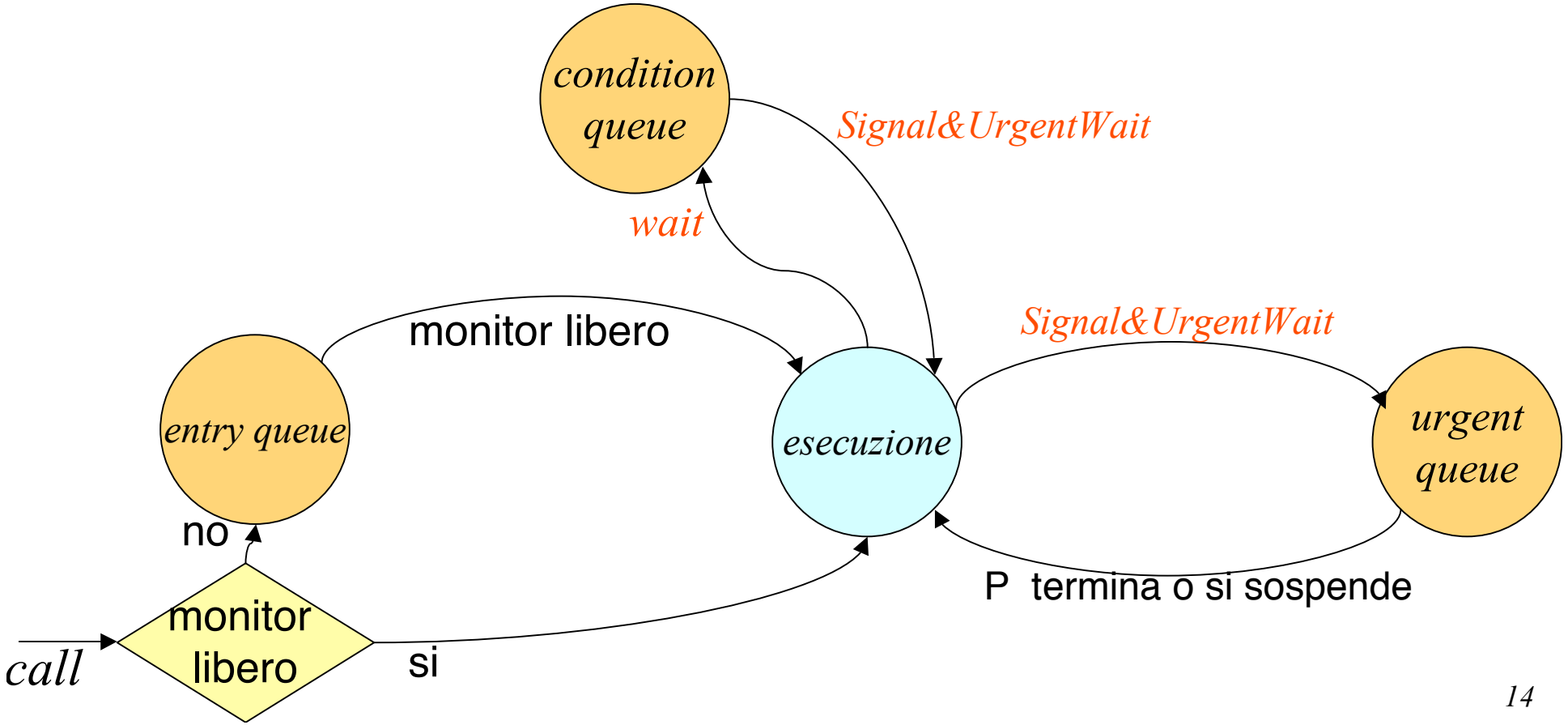
Semantiche della signal



Signal_and_urgent_wait

E` una variante della signal_and_wait.

signal_and_urgent_wait. Q ha la priorità rispetto agli altri processi che aspettano di entrare nel monitor. Viene quindi sospeso in una coda interna al monitor (*urgent queue*). Quando P ha terminato la sua esecuzione (o si è nuovamente sospeso), trasferisce il controllo a Q senza liberare il monitor.



- Un caso particolare della **signal _and_urgent_wait** (e della signal_and_wait) si ha quando essa corrisponde ad una istruzione return: **signal_and_return**.

- Il processo completa cioè la sua operazione con il risveglio del processo segnalato. Cede ad esso il controllo del monitor senza rilasciare la mutua esclusione.

signal_and_continue

- Il processo segnalato P viene trasferito dalla coda associata alla variabile condizione alla entry_queue e potrà rientrare nel monitor una volta che Q l'abbia rilasciato.

- Poiché altri processi possono entrare nel monitor prima di P, questi potrebbero modificare la condizione di sincronizzazione (lo stesso potrebbe fare Q).

- E' pertanto necessario che quando P rientra nel monitor ritesti la condizione:

```
while(!B) wait (cond);
<accesso alla risorsa>
```

- E’ possibile anche **risvegliare tutti i processi** sospesi sulla variabile condizione utilizzando la :

  signal_all

  che è una variante della signal_and_continue.

➔ Tutti i processi risvegliati vengono messi nella entry_queue dalla quale, uno alla volta potranno rientrare nel monitor.

Ulteriori operazioni sulle variabili condizione

Sospensione con indicazione della priorità:

```
wait(cond, p);
```

➔ i processi sono accodati rispettando il valore (crescente o decrescente) di p e vengono risvegliati nello stesso ordine.

Verifica dello stato della coda:

```
empty(cond);
```

➔ fornisce il valore false se esistono processi sospesi nella coda associata a cond , true altrimenti.

Variabili condizione monoprocesso

- E` possibile prevedere un array di variabili condizione, una per ogni processo: **variabili condizione monoprocesso.**
- Ogni processo può sospendersi sulla propria variabile condizione. Ciò consente di risvegliare un ben determinato processo.

```
monitor nome_monitor {
<dichiarazione delle variabili locali>;
<inizializzazione delle variabili locali>;
condition cond [num_max_proc];
    …..
    public void op1 (int k) {
    if (!B)  wait (cond[k]);
    ….. }
    public void op2 ( ) {
    ..
    signal(cond[i]);
    }
}
```

Esempi d'uso: monitor come gestore di risorse

```
monitor buffer_circolare {
        messaggio buffer[N];
        int contatore=0;int testa=0; int coda=0;
        condition non-pieno; condition non-vuoto;
        public void invio (messaggio m)
        {       if (contatore==N)
                        wait(non-pieno);
                buffer[coda] = m;
                coda = (coda+1)% N;
                contatore ++;
                signal(non-vuoto);
        }
        public messaggio ricezione ( )
        {       messaggio m;
                if (contatore==0)
                        wait(non-vuoto);
                m=buffer[testa];
                testa =(testa+1)% N;
                contatore --;
                signal(non-pieno);
                return m; }
        }
```

Esempi d'uso: monitor come allocatore di una risorsa

```
monitor allocatore {
        boolean occupato=false;
        condition libero;
        public void richiesta ( )
        {       if (occupato) wait (libero);
                occupato=true;
        }
        public void rilascio ( )
        {       occupato=false;
                signal(libero);
        }
}
```

Realizzazione del costrutto monitor tramite semafori

Il compilatore assegna ad ogni istanza di un monitor:

- un semaforo **`mutex`** inizializzato a 1 per la mutua esclusione delle operazioni del monitor:
  - la richiesta di un processo di eseguire un'operazione public equivale all'esecuzione di una **`P(mutex)`**.

Il compilatore assegna a ogni variabile **cond** di tipo condition:

- un semaforo **`condsem`** inizializzato a 0 sul quale il processo si può sospendere tramite una **`wait(condsem)`**.
- un contatore **`condcount`** inizializzato a 0 per tenere conto dei processi sospesi su condsem.

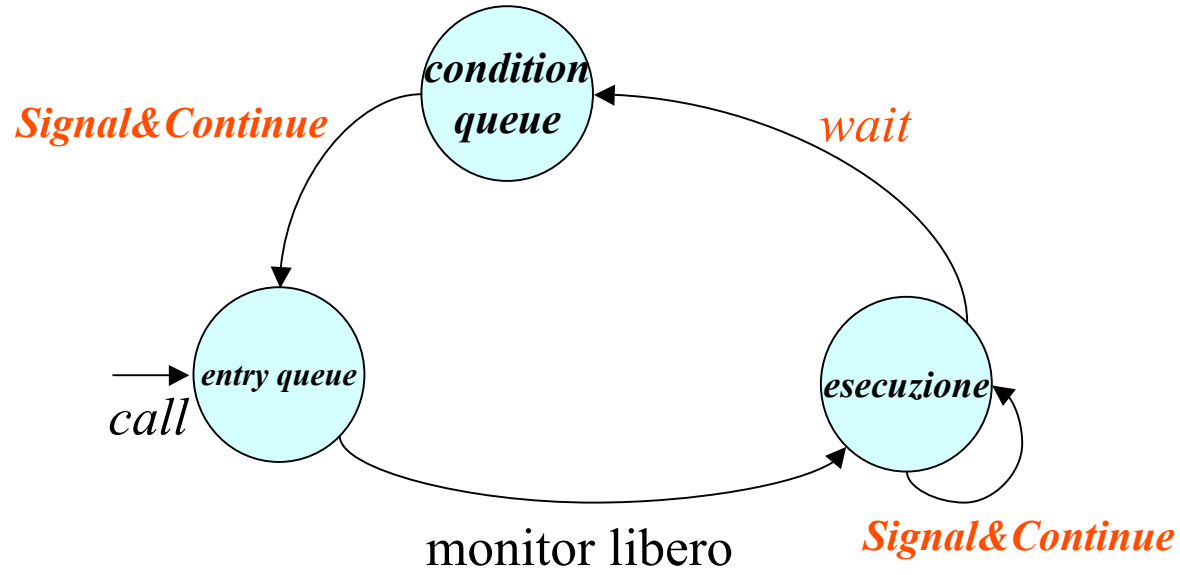
Signal_and_ continue

Prologo di ogni operazione public: `P(mutex);`

Epilogo di ogni operazione public: `V(mutex);`

wait(cond):

```
{    condcount++;
     V(mutex);
     P(condsem);
     P(mutex);
}
```



signal(cond):

```
{    if (condcount>0)
     {      condcount--;
            V(condsem);
     }
}
```

Signal_and_ wait

Prologo di ogni operazione public `P(mutex);`

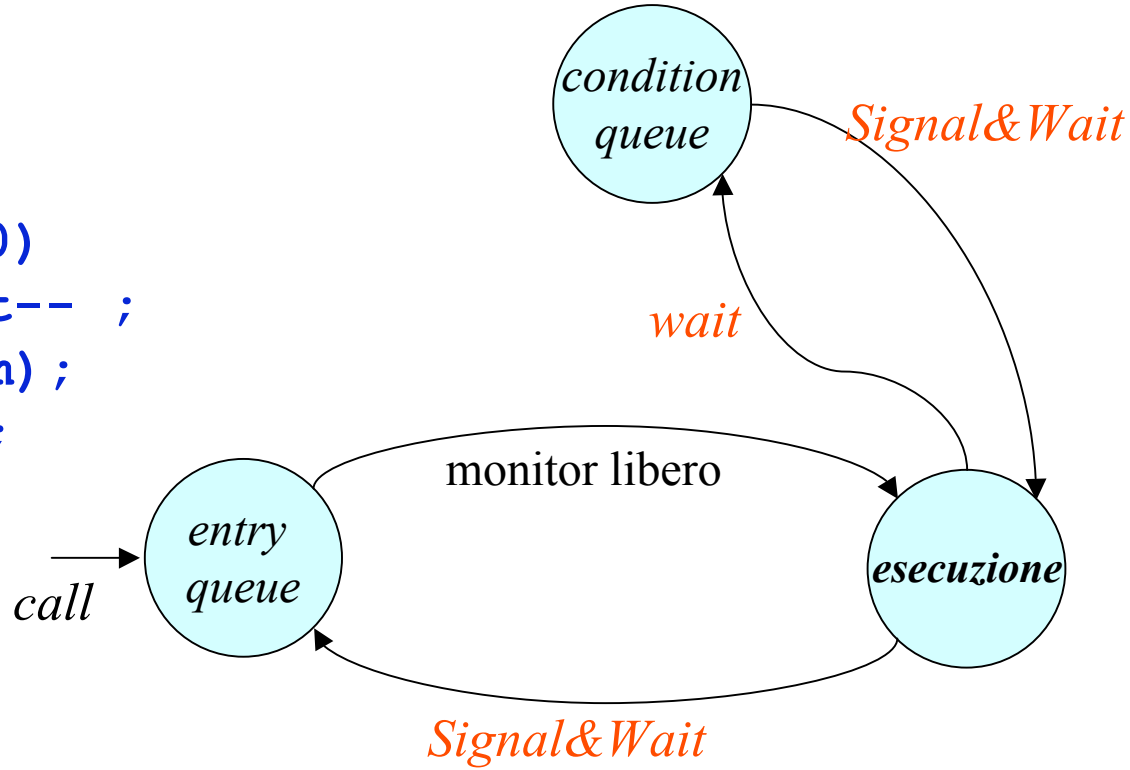
Epilogo di ogni operazione public `V(mutex);`

wait(cond):

```
{       condcount++;
        V(mutex);
        P(condsem);
}
```

signal(cond):

```
{       if (condcount>0)
        {       condcount-- ;
                V(condsem);
                P(mutex);
        }
}
```



Signal_and_urgent_wait

urgent: semaforo per la sospensione del processo segnalante (v.iniz. 0)
urgentcount: contatore dei processi sospesi su urgent

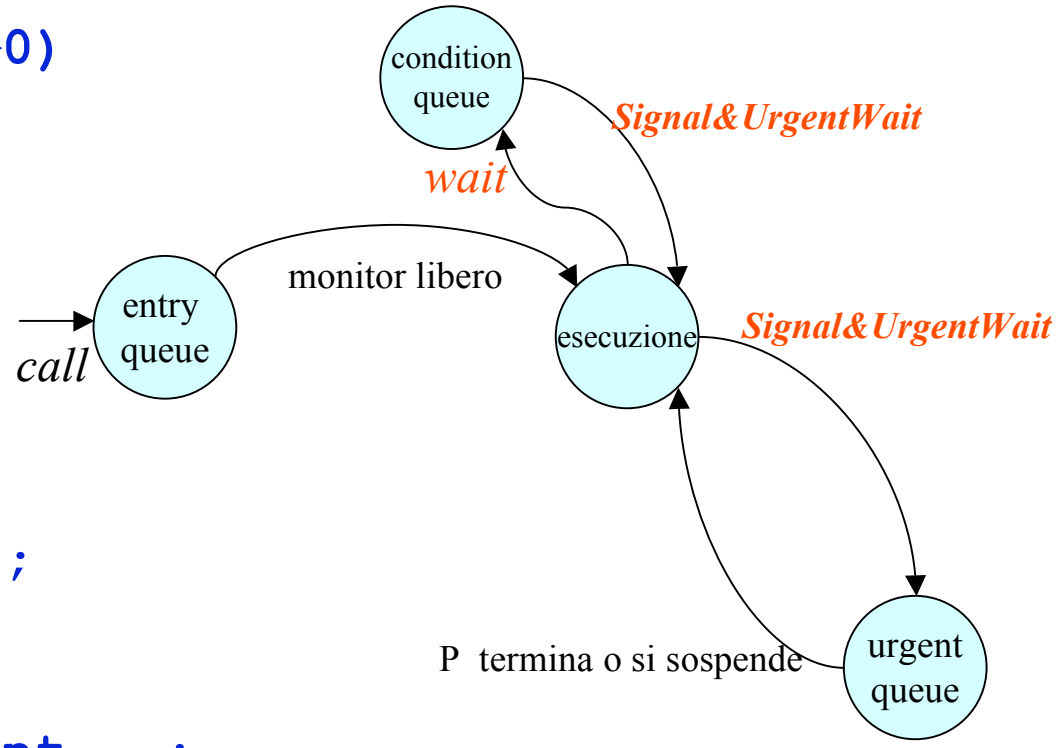
Prologo di ogni operazione: `P(mutex);`
Epilogo di ogni operazione: `if(urgentcount>0) V(urgent) else V(mutex);`

wait(cond):

```
{       condcount++;
        if (urgentcount>0)
              V(urgent);
        else V(mutex);
        P(condsem);
        condcount-- ; }
```

signal(cond):

```
{       if (condcount>0
        {  urgentcount++;
           V(condsem);
           P(urgent);
           urgentcondcount-- ;
}       }
```



Signal_and_return

Prologo di ogni operazione: `P(mutex);`

Epilogo di ogni operazione:

- se la funzione non contiene signal allora : `V(mutex)`
- altrimenti `signal(cond)` (vedi sotto).

wait(cond):

```
{       condcount++;
        V(mutex);
        P(condsem);
        condcount-- ;
}
```

signal(cond):

```
{       if (condcount>0)
                V(condsem);
        else V(mutex);
}
```

Realizzazione di politiche di gestione delle risorse

- Allocazione di una risorsa mediante la strategia Shortest-job-next: più processi competono per l'uso di una risorsa. Quando questa viene rilasciata, essa viene assegnata, tra tutti i processi sospesi, a quello che la userà per il periodo di tempo inferiore.

```
monitor allocatore
{       boolean occupata= false;
        condition non_occupata;

        public void richiesta(int tempo)
        {       if (occupato)
                        wait(non_occupata, tempo);
                occupato = true;
        }

        public void rilascio()
        {       occupato = false;
                signal(non_occupata);
        }
}
```

Problema dei lettori/scrittori

Politica:

- Un nuovo lettore non può acquisire la risorsa se c'è uno scrittore in attesa.

- Tutti i lettori sospesi al termine di una scrittura hanno priorità sul successivo scrittore.

```
monitor lettori_scrittori
{       int num_lettori=0;
        boolean occupato=false;
        condition ok_lettura, ok_scrittura;
        public void inizio_lettura
        {       if (occupato || !empty(ok_scrittura))
                        wait(ok_lettura);
                num_lettori++;
                signal(ok_lettura);             }
        public void fine_lettura
        {       num_lettori--;
                if (num_lettori=0)
                        signal(ok_scrittura); }
        public void inizio_scrittura
        {       if (num_lettori!=0 || occupato=true)
                        wait(ok_scrittura);
                occupato=true;          }
        public void fine_scrittura
        {       occupato= false;
                if (!empty(ok_lettura)) /* priorita` ai lettori*/
                        signal(ok_lettura);
                else signal(ok_scrittura); }
}
```

Il monitor nella libreria pthread

Sincronizzazione dei thread

La libreria pthread non prevede il costrutto monitor, ma offre comunque gli strumenti di sincronizzazione necessari ad implementare il monitor.

Strumenti di sincronizzazione pthread:

- mutex (semaforo binario)
- variabili condizione

Variabili condizione (condition)

Consente la sospensione dei thread in attesa che sia soddisfatta una condizione logica.

una variabile condizione è definita dal tipo `pthread_cond_t` che rappresenta una coda per la sospensione dei thread.

Operazioni fondamentali:

- inizializzazione: **p_thread_cond_ init**
- sospensione: **p_thread_cond_ wait**
- risveglio: **p_thread_cond_ signal**

- Una variabile condizione C viene creata e inizializzata nel modo seguente:

```
p_thread_cond_t  C;
p_thread_cond_ init (&C,attr);
```

dove

- **`C`** : individua la condizione da inizializzare
- **`attr`** : punta a una struttura che contiene gli attributi della condizione; se NULL, viene inizializzata a default.

NB: linux non implementa gli attributi !

in alternativa, una variabile condizione può essere inizializzata staticamente con la costante: PTHREAD_COND_INIZIALIZER

esempio:

```
pthread_cond_t C= PTHREAD_COND_INIZIALIZER ;
```

Esempio: accesso di un produttore al buffer condiviso

```
/*variabili globali*/
p_thread_cond_t C=PTHREAD_COND_INITIALIZER;
boolean bufferpieno=false;
….
/* codice produttore:*/
….
while (bufferpieno) <sospensione su C>;
<inserimento messaggio nel buffer>;
```

Ogni operazione di accesso al buffer è una sezione critica.

Alla risorsa buffer va associato un mutex:
→`lock` prologo dell'operazione; `unlock` epilogo dell'operazione.

```
/*variabili globali*/
p_thread_cond_t  C;
p_thread_mutex M;
boolean bufferpieno=0;


...
/* codice produttore:*/
….
p_thread_mutex_lock (&M);
while (bufferpieno) <sospensione su C>;
<inserimento messaggio nel buffer>;
p_thread_mutex_unlock(&M);
```

Sospensione di thread: p_thread_cond_wait

```
p_thread_cond_wait(p_thread_cond_t*C, p_thread_mutex_t*M)
```

- C: variabile condizione;
- M: mutex associato alla condizione C (e` il mutex della risorsa condivisa).

Effetti:

1. il thread chiamante viene sospeso nella coda associata a C
2. M viene liberato
3. al risveglio del thread, M viene rioccupato automaticamente.

Risveglio di un thread: p_thread_cond_signal

```
pthread_cond_signal (pthread_cond_t *C);
```

Effetti:
Se esistono thread sospesi nella coda associata a C, viene risvegliato il primo;

Se non vi sono thread sospesi sulla condizione, la signal non ha alcun effetto.

La politica realizzata dalla signal è del tipo *signal_ and_ continue*: il thread che ha eseguito la signal prosegue l'esecuzione mantenendo il controllo di mutex fino ad esplicito rilascio.

Pthread: simulazione del comportamento del monitor

- Mutua esclusione delle operazioni del monitor: uso di lock e unlock su un mutex all'inizio ed al termine di ogni operazione public.
- Sospensione su una variabile condizione: uso di pthread_ cond_ wait.
- Riattivazione: pthread_ cond_ signal.
- Viene adottata la politica signal_and_ continue.
- Non c'e' controllo da parte del compilatore.

Esempio: produttore e consumatore

Si vuole risolvere il problema del produttore e consumatore, con una metodologia basata sul concetto di monitor.

Progetto del monitor (`prodcons`): dati locali

- buffer circolare di interi, di dimensione data (ad esempio, 16) il cui stato e` dato da:
  - numero degli elementi contenuti: `cont`
  - puntatore alla prima posizione libera: `writepos`
  - puntatore al primo elemento occupato : `readpos`
- il buffer e` una risorsa da accedere in modo mutuamente esclusivo:
  - predispongo un mutex per il controllo della mutua esclusione nell'accesso al buffer: `lock`
- i thread produttori e consumatori necessitano di sincronizzazione in caso di :
  - buffer pieno: definisco una condition per la sospensione dei produttori se il buffer e` pieno (`notfull`)
  - buffer vuoto: definisco una condition per la sospensione dei produttori se il buffer e` vuoto (`notempty`)

➔ Inserisco il tutto all'interno di un tipo struct : `prodcons` (rappresenta l'insieme dei dati locali al monitor)

```
typedef struct
{
  int buffer[BUFFER_SIZE];
  pthread_mutex_t lock;
  int readpos, writepos;
  int cont;
  pthread_cond_t notempty;
  pthread_cond_t notfull;
}prodcons;
```

Produttore e consumatore

Operazioni public sulla risorsa `prodcons`:

- `Inserisci`: operazione eseguita da ogni produttore per l'inserimento di un nuovo elemento.

- `Estrai`: operazione eseguita da ogni consumatore per l'estrazione di un elemento dal buffer.

➔ Inserisci ed estrai sono le operazioni *public* del monitor.

Inoltre, e` necessario prevedere una funzione di inizializzazione dei dati interni al monitor:

- `Init`: inizializzazione del buffer.

Esempio: produttore e consumatore

```
#include <stdio.h>
#include <pthread.h>

#define BUFFER_SIZE 16


typedef struct
{ int buffer[BUFFER_SIZE];
  pthread_mutex_t lock;
  int readpos, writepos;
  int cont;
  pthread_cond_t notempty;
  pthread_cond_t notfull;
}prodcons;
```

Esempio:Operazioni sul buffer

```
/* Inizializza il buffer */
void init (prodcons *b)
{
  pthread_mutex_init (&b->lock, NULL);
  pthread_cond_init (&b->notempty, NULL);
  pthread_cond_init (&b->notfull, NULL);
  b->cont=0;
  b->readpos = 0;
  b->writepos = 0;
}
```

Operazioni sul buffer

```
/* operazione public Inserimento: */
void inserisci (prodcons *b, int data)
{  pthread_mutex_lock (&b->lock);
  /* controlla che il buffer non sia pieno:*/
  while ( b->cont==BUFFER_SIZE)
      pthread_cond_wait (&b->notfull, &b->lock);
  /* scrivi data e aggiorna lo stato del buffer */
  b->buffer[b->writepos] = data;
  b->cont++;
  b->writepos++;
  if (b->writepos >= BUFFER_SIZE)
    b->writepos = 0;
  /* risveglia eventuali thread (consumatori) sospesi */
  pthread_cond_signal (&b->notempty);
  pthread_mutex_unlock (&b->lock);
}
```

Operazioni sul buffer

```
/*operazione public ESTRAZIONE:  */
int estrai (prodcons *b)
{  int data;
  pthread_mutex_lock (&b->lock);
  while (b->cont==0) /* il buffer e` vuoto? */
      pthread_cond_wait (&b->notempty, &b->lock);
  /* Leggi l'elemento e aggiorna lo stato del buffer*/
  data = b->buffer[b->readpos];
  b->cont--;
  b->readpos++;
  if (b->readpos >= BUFFER_SIZE)
    b->readpos = 0;
  /* Risveglia eventuali threads (produttori)*/
  pthread_cond_signal (&b->notfull);
  pthread_mutex_unlock (&b->lock);
  return data;
}
```

Produttore/consumatore: programma di test

```
/* Programma di test: 2 thread
   - un thread inserisce sequenzialmente max interi,
   - l'altro thread li estrae sequenzialmente per stamparli */

#define OVER (-1)
#define max 20

prodcons buffer;

void *producer (void *data)
{ int n;
  printf("sono il thread produttore\n\n");
  for (n = 0; n < max; n++)
    { printf ("Thread produttore %d --->\n", n);
      inserisci (&buffer, n);
    }
  inserisci (&buffer, OVER);
  return NULL;
}
```

```
void *consumer (void *data)
{  int d;
  printf("sono il thread consumatore \n\n");

  while (1)
    {
      d = estrai (&buffer);
      if (d == OVER)
        break;
      printf("Thread consumatore: --> %d\n",  d);
    }
  return NULL;
}
```

```
main ()
{
  pthread_t th_a, th_b;
  void *retval;

  init (&buffer);
  /* Creazione threads: */
  pthread_create (&th_a, NULL, producer, 0);
  pthread_create (&th_b, NULL, consumer, 0);
  /* Attesa teminazione threads creati: */
  pthread_join (th_a, &retval);
  pthread_join (th_b, &retval);
  return 0;
}
```

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**

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)*.

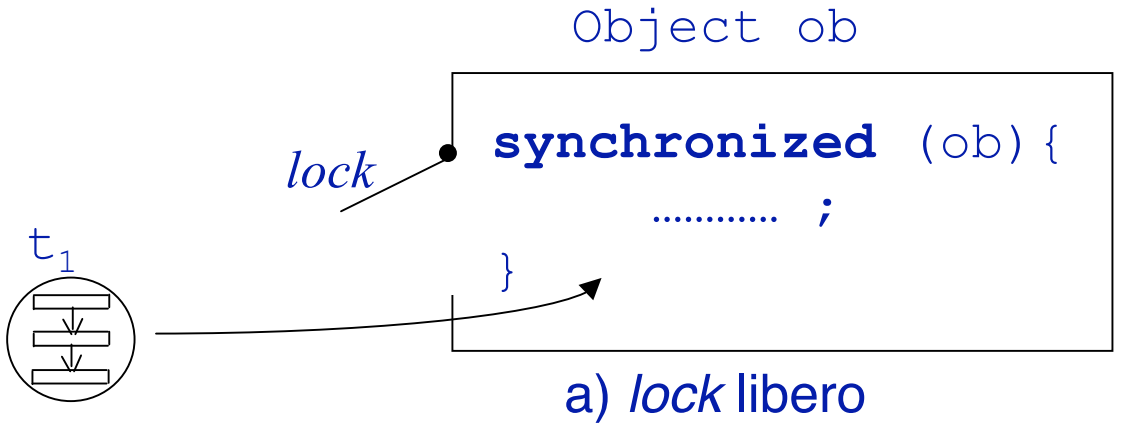
Al termine della sezione critica:

- 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 .

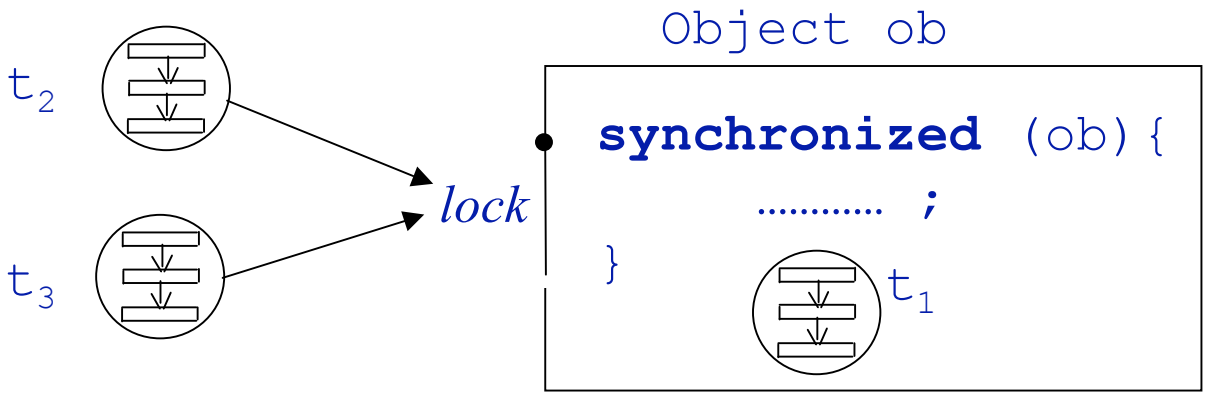
synchronized block

- 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***.

Sincronizzazione diretta: `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*** (possesso del lock dell'oggetto).

wait, notify, notifyall

`wait` comporta il rilascio del lock, la sospensione del thread ed il suo inserimento nel ***wait set***.

`notify` comporta l'estrazione di un thread da **wait set** ed il suo inserimento nell'***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 che il lock venga liberato.

➔ 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()
    {   while (pieno==false)
                wait ( );
        pieno=false;
        notify();
        return contenuto;
   }

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

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

  public Mailbox(){
    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();
  }
}
```

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.

- Le primitive *P* e *V* (*wait* e *signal* sui semafori) si possono ottenere dichiarandole come **syncronized methods** all'interno della classe *semafor*o.

```
public class Semaphore {
  private int value;
  public Semaphore (int initial){
    value = initial;
  }

synchronized public void V()//signal sul semaforo
{    ++value;
    notify();
}

synchronized public void P() throws InterruptedException //wait
{
        {    while (value == 0) wait();
             --value;
        }
}
```

wait&notify

Principale limitazione :

- **unico wait-set** per un oggetto sincronizzato

➔ non e` possibile sospendere thread su differenti code!

Variabili condizione

- Nelle versioni più recenti di Java (***Java™ 2 Platform Standard Ed. 5.0***) esiste la possibilità utilizzare le **variabili condizione**. Ciò è ottenibile tramite l'uso un'apposita interfaccia (definita in `java.util.concurrent.locks`):

```
public interface Condition{
//Public instance methods
void await ()throws InterruptedException;
void signal();
void signalAll();
}
```

- dove i metodi `await`, `signal`, e `signalAll` sono del tutto equivalenti ai metodi wait, notify e notify_all, (ovviamente riferiti alla coda di processi associata alla condition sulla quale vengono invocati)

Mutua esclusione: lock

- Oltre a metodi/blocchi synchronized, la versione 5.0 di Java prevede la possibilita` di utilizzare esplicitamente il concetto di *lock*, mediante l'interfaccia (definita in `java.util.concurrent.locks`):

```
public interface Lock{
//Public instance methods
void lock();
void unlock();
Condition newCondition();
}
```

Uso di Variabili Condizione

- **Ad ogni variabile condizione deve essere associato un lock**, che:
  - al momento della sospensione del thread mediante **`await`** verra` liberato;
  - al risveglio di un thread, verra` automaticamente rioccupato.

➔ La creazione di una condition deve essere effettuata mediante in metodo newCondition del lock associato ad essa.

In pratica, per creare un oggetto Condition :

```
Lock lockvar=new Reentrantlock(); //Reentrantlock è una
                                  classe che implementa
                          l'interfaccia Lock
Condition C=lockvar.newCondition();
```

Monitor

Con gli strumenti visti, possiamo quindi definire **classi** che rappresentano monitor:

- Dati:
  - le variabili condizione
  - 1 lock per la mutua esclusione dei metodi "entry", da associare a turre le variabili condizione
  - variabili interne: stato delle risorse gestite
- Metodi:
  - metodi public ("entry")
  - metodi privati
  - costruttore

Esempio: gestione di buffer circolare

```
public class Mailbox
{ //Dati:
private int[] contenuto;
private int contatore,testa,coda;
private Lock lock= new ReentrantLock();
private Condition non_pieno= lock.newCondition();
private Condition non_vuoto= lock.newCondition();

//Costruttore:
public Mailbox( ) {
contenuto=new int[N];
contatore=0;
testa=0;
coda=0;
}
```

```
//metodi "entry":

public int preleva()throws InterruptedException
{  int elemento;
   lock.lock();
   try
   {    while (contatore== 0)
                non_vuoto.await();
        elemento=contenuto[testa];
        testa=(testa+1)%N;
        --contatore;
        non_pieno.signal ( );
   } finally{lock.unlock();}
   return elemento;
}
```

```
public void deposita (int valore)throws InterruptedException
{  lock.lock();
   try
   {    while (contatore==N)
               non_pieno.wait();
        contenuto[coda] = valore;
        coda=(coda+1)%N;
        ++contatore;
        non_vuoto.signal( );
   } finally{  lock.unlock();}
}
}
```

Programma di test:

```
public class Produttore extends Thread
{   int messaggio;
    Mailbox m;
    public  Produttore(Mailbox M){this.m =M;}
    public void run()
    {     while(1)
          { <produci messaggio>
            m.deposita(messaggio);
          }

    }
}

public class Consumatore extends Thread
{   int messaggio;
    Mailbox m;
    public  Consumatore(Mailbox M){this.m =M;}
    public void run()
    {     while(1)
          {         messaggio=m.preleva();
                    <consuma messaggio>
          }

    }
```

```
public class BufferTest{

   public static void main(String args[])
   {   Mailbox M=new Mailbox();
       Consumatore C=new Consumatore(M);
       Produttore P=new Produttore(M);
       C.start();
       P.start();
       ...
   }
```