Sincronizzazione dei processi
(seconda parte)

Semafori: notazioni

- Alcune notazioni comunemente utilizzate per indicare le operazioni di acquisizione e rilascio di un semaforo ***S***

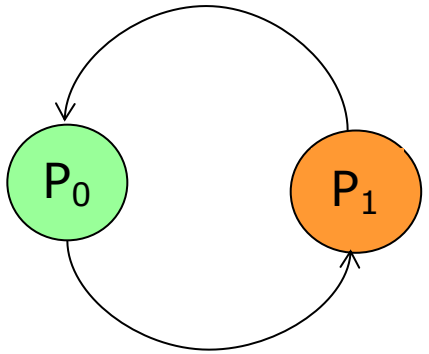
wait(S)	***signal(S)***
S.wait()	***S.signal()***
S.P()	***S.V()***

- Concetto proposto da Edsger Dijkstra e implementato nel sistema operativo THE (1968).
  - ***P*** indica *proberen* (test) e ***V*** indica *verhogen* (incrementare).

Deadlock e Starvation

- **Deadlock** – due o più processi (o thread) sono indefinitamente in attesa per un evento che può essere causato da uno soltanto dei processi in attesa.
- Siano ***S*** e ***Q*** due semafori inizializzati a 1:

P_0	P_1
S.wait();	Q.wait();
Q.wait();	S.wait();
⋮	⋮
S.signal();	Q.signal();
Q.signal();	S.signal();



- **Starvation** – blocking indefinito di un processo **(diversa dal deadlock).**
- Un processo potrebbe non essere mai rimosso dalla coda del semaforo in cui è sospeso.

Semafori binari

- Il semaforo definito in precedenza è chiamato *semaforo contatore*, in quanto il suo valore intero può spaziare su un dominio specificato.
- Si definisce *semaforo binario* un semaforo il cui valore intero può valere solo 0 e 1.
- È possibile implementare un semaforo binario a partire da un semaforo contatore.

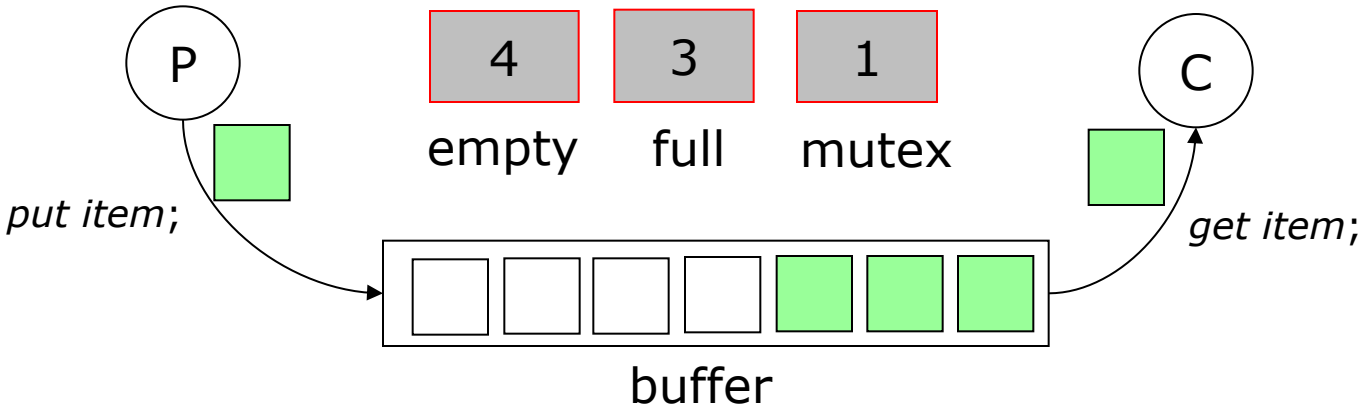
La classe Semaphore di Java

- **java.util.concurrent.Semaphore**
- Implementa un semaforo contatore simile a quello definito in precedenza
- Presenta tuttavia alcune differenze:
  - Al posto dei metodi **P**() e **V**() sono definite le operazioni (*metodi*)

    acquire*(…)* e ***release****(…)*
  - Il valore massimo del contatore è fissato nel costruttore.
  - Un thread può incrementare o decrementare il contatore del semaforo di una quantità non necessariamente unitaria.

Produttore-Consumatore con buffer limitato

- L'uso dei semafori può garantire la consistenza dei dati assicurando l'esecuzione ordinata dei processi concorrenti.
- Due processi: *produttore* (P) e *consumatore* (C)
- Tre semafori: *empty (*posiz. vuote*), full* (posiz. piene) e *mutex* (mutua esclusione sul buffer)



Produttore- Consumatore

Produttore

```
while (true) {
      /* produce un elemento e lo inserisce in
         nextProduced */
     wait(empty);
     wait(mutex);
     buffer [in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     signal(mutex);
     signal(full);
}
```

Consumatore

```
while (true) {
     wait(full);
     wait(mutex);
     nextConsumed =  buffer[out];
     out = (out + 1)%BUFFER_SIZE;
     signal(mutex);
     signal(empty);
     /*  consuma l'item in nextConsumed */
}
```

inizializzazione

```
empty= BUFFER_SIZE;
full = 0;
mutex = 1;
```

Problema dei Lettori-Scrittori

- Processi **lettori** (accedono) e processi **scrittori** (modificano) dati condivisi.
- Più lettori possono leggere insieme, **soltanto uno** scrittore alla volta più modificare i dati.
- Dati condivisi:

```
Semaphore mutex, wrt;
int readerCount;
```

Inizialmente:

```
mutex = 1;          // per la mutua esclusione dei lettori su readerCount
wrt = 1;            // per la mutua esclusione degli scrittori
readerCount = 0;    // contatore dei lettori correnti
```

Lettori-Scrittori: thread Scrittore

Scrittore

```
while (true)
{
   wrt.acquire(); //wait
     …
     <scrive>
     …
   wrt.release(); //signal
}
```

Un solo processo scrittore per volta (sezione critica).

Lettori-Scrittori: thread Lettore

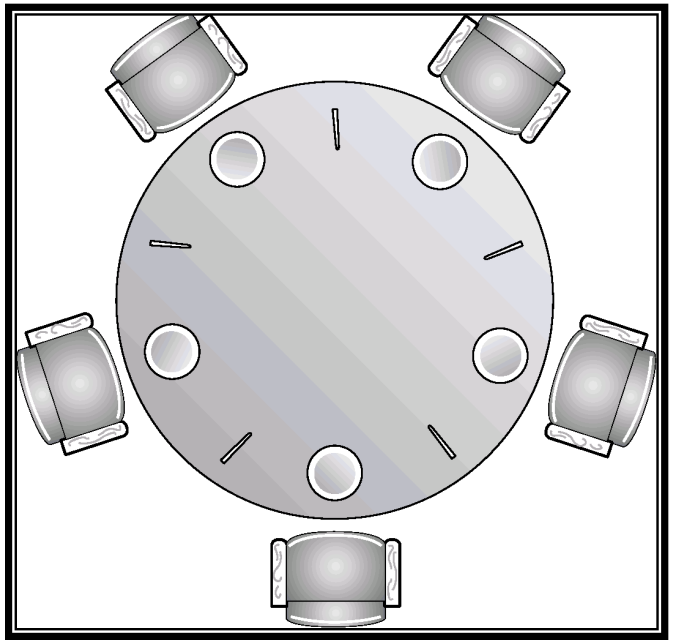
Lettore

```
while (true)
{
   mutex.acquire();
   readerCount++;
   if (readerCount == 1)
      wrt.acquire();
   mutex.release();
         …
      <legge>
         …
   mutex.acquire();
   readerCount--;
   if (readerCount == 0)
     wrt.release();
 mutex.release();
}
```

Il primo e l'ultimo lettore devono gestire la mutua esclusione con gli scrittori.

Problema dei Cinque Filosofi

- 5 Filosofi mangiano usando due bacchette.
- Rappresenta problemi di allocazione di risorse multiple.
- Si può rappresentare ciascuna bacchetta con un semaforo.
- Ogni filosofo tenta di afferrare una bacchetta con un'operazione di ***wait*** e la rilascia eseguendo ***signal*** sul semaforo appropriato.



- Dati condivisi:

```
Semaphore chopstick[] = new Semaphore [5];
```

Inizialmente tutti i 5 semafori valgono 1.

Problema dei Cinque Filosofi

- Filosofo *i*:

```
while (true) {
  chopstick[i].acquire();
  chopstick[(i+1) % 5]).acquire();
    …
    <mangia>
    …
  chopstick[i].release();
  chopstick[(i+1) % 5].release();
    …
    <pensa>
    …
}
```

- Semplice ma con possibilità di deadlock!
- Soluzioni più complesse risolvono il problema.

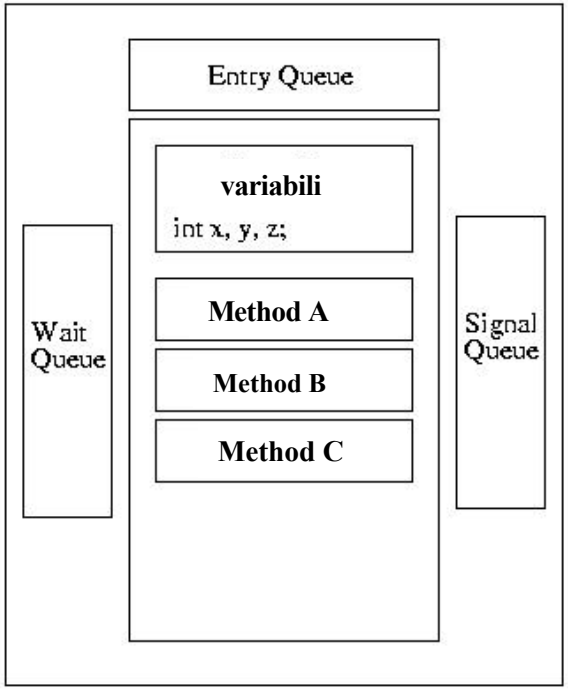
Soluzioni?

Monitor

- Il **Monitor** è un costrutto di alto livello che consente la condivisione sicura di un tipo di dati astratto tra più thread **garantendo la mutua esclusione**.

- Può essere implementato come una classe che racchiude *variabili* condivise e *condition* ed esporta *metodi* che accedono a tali variabili in modo atomico.

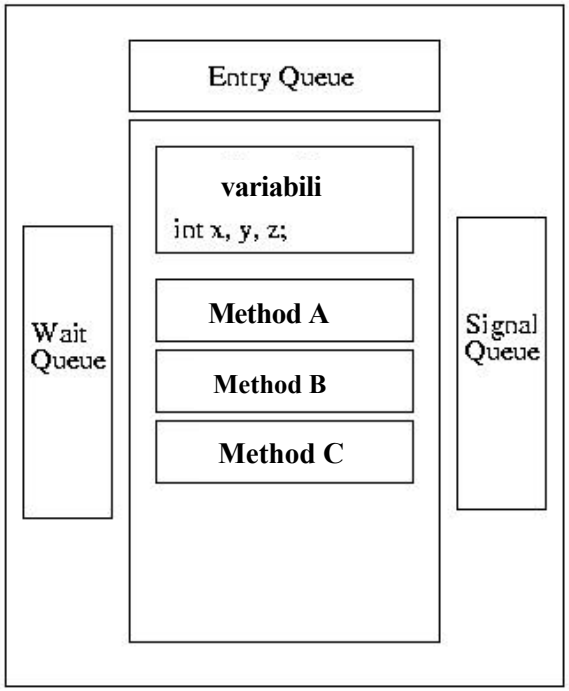
NB: Il costrutto monitor garantisce che soltanto un processo per volta può essere attivo nel monitor!



Monitor

- Le variabili del monitor possono essere utilizzate solo mediante i metodi di accesso e **solo un thread per volta può accedere al monitor**.
- Ad ogni istanza del monitor è associata una coda di thread in attesa di eseguire uno dei metodi di accesso.

NB: Il costrutto monitor garantisce che soltanto un processo per volta può essere attivo nel monitor!



Monitor

- Per consentire ad un processo di attendere all'interno di un monitor si possono utilizzare le cosiddette ***condition***:

  condition x, y;

- Una variabile ***condition*** può essere manipolata solo attraverso le operazioni **wait** e **signal**.
  - L'operazione

    x.wait();

    sospende il processo che la invoca fino a quando un altro processo non invoca:

    x.signal();

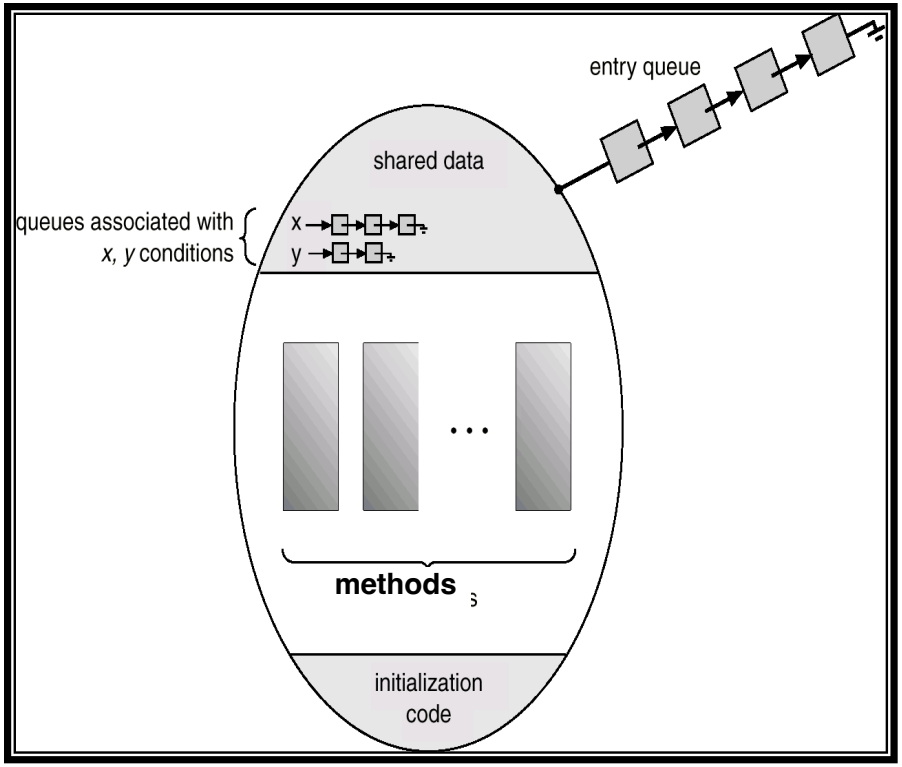
  - L'operazione **x.signal()** risveglia esattamente un processo. Se nessun processo è sospeso l'operazione di **signal** non ha effetto.

Rappresentazione concettuale di un monitor

```
Monitor nome_monitor
{
  dichiarazione variabili
  del monitor e condition;

  metodo m1(…);
    { …  }
  metodo m2(…);
    { …  }
      .
      .
  metodo mn(…);
    { …  }

  begin
    codice inizializzaz.;
   end;
}
```



Monitor di Hoare e di Hansen

Supponiamo che un processo **P** esegua **x.signal()** ed esista un processo sospeso **Q** associato alla variabile condition **x**.

P potrebbe aver eseguito la **x.signal()** non come ultima operazione nel monitor.

Dopo la ***signal***, occorre evitare che **P** e **Q** risultino contemporaneamente attivi all'interno del monitor.

- ***Soluzione di Hansen***: **Q** attende che **P** lasci il monitor o si metta in attesa su un'altra variabile condition diversa da **x**.

- ***Soluzione di Hoare***: **P** attende che **Q** lasci il monitor o si metta in attesa su un'altra variabile condition diversa da **x**.

Monitor di Hoare e di Hansen

P esegue **x.signal()** ed esiste un processo sospeso **Q** associato alla variabile condition **x**.

■**Soluzione di Hansen**: **Q** attende che **P** lasci il monitor o si metta in attesa su un'altra variabile condition.

■**Soluzione di Hoare**: **P** attende che **Q** lasci il monitor o si metta in attesa su un'altra variabile condition.

■***La soluzione di Hoare è preferibile***: infatti consentendo al processo **P** di continuare (soluzione di Hansen), la condizione logica attesa da **Q** potrebbe non valere più nel momento in cui **Q** viene ripreso.

■*Una soluzione di compromesso*: **P** deve lasciare il monitor nel momento stesso dell'esecuzione dell'operazione ***signal***, in quanto viene immediatamente ripreso il processo **Q**.

■Questo modello è meno potente di quello di Hoare, perchè un processo non può effettuare una ***signal*** più di una volta all'interno di una stessa procedura.

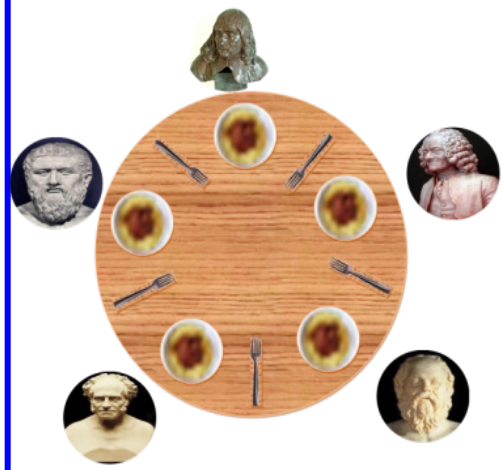
Esempio dei cinque filosofi (con il monitor)

- **Soluzione senza deadlock**: al filosofo è permesso di prelevare le bacchette solo se sono entrambe disponibili.

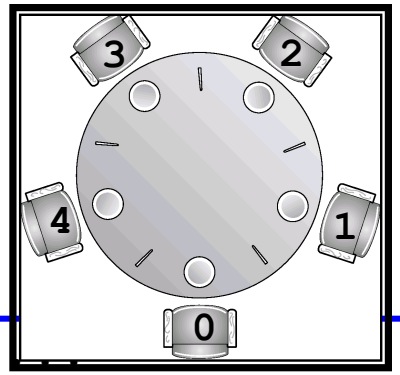
- La distribuzione delle bacchette è controllata dal monitor **`dining_phil`**.

```
monitor dining_phil
{
  var state[5]= {thinking, hungry, eating};
  var self[5] condition;

  init()
  {
    for (int i = 0; i <= 4; i++)
      state[ i ] = thinking;
  }
}
 . . .
```



Esempio dei cinque filosofi (con il monitor)



Ciclo del codice del ***filosofo i***

```
while (true) {
   dining_phil.pickup(i);
      // mangia
   dining_phil.putdown(i);
}
```

```
method pickup(int i) {
  state[i] = hungry;
  test(i); //controlla se può mangiare
  if (state[i] != eating)
    self[i].wait();
}
```

Se almeno una bacchetta è in uso attende

```
method putdown(int i) {
  state[i] = thinking;
  test((i+4)%5); //controlla i vicini
  test((i+1)%5); //destro e sinistro
}                //e se possibile li sblocca
```

```
... //verifica disponibilità delle 2 bacchette
method test(int i) {
  if ((state[(i+4)%5]!=eating)&&
     (state[i]==hungry)&&
     (state[(i+1)%5]!=eating))
    {
      state[i] = eating;
      self[i].signal();
    }
}
```

Il processo può iniziare a mangiare

Monitor in Java

- In Java ogni oggetto ha associato il proprio monitor se contiene del codice sincronizzato (metodo dichiarato come **`synchronized`**).

- Java non implementa fisicamente il concetto di monitor di un oggetto, ma questo è facilmente associabile alla parte sincronizzata dell'oggetto stesso.

- In pratica, se un thread **T** entra in un metodo **`synchronized ms`** di un determinato oggetto **O**, nessun altro thread potrà entrare in nessun metodo sincronizzato dell'oggetto **O**, sino a quando **T**, non avrà terminato l'esecuzione del metodo **`ms`** (ovvero non avrà abbandonato il monitor dell'oggetto).

- Si dice che il thread **T**, ha il "lock" dell'oggetto **O**, quando è entrato nel suo monitor (metodo **`synchronized`**).

Lock e Condition in Java

Java supporta meccanismi di mutua esclusione e sincronizzazione simili a quelli dei monitor, mediante le interfacce **Lock** e **Condition** del package ***java.util.concurrent.locks*;**

■ Un **Lock** è un costrutto per controllare l'accesso ad una risorsa condivisa tra più thread. Un **Lock garantisce accesso esclusivo**: soltanto un thread per volta può acquisire il **Lock** ed accedere alla risorsa condivisa.

■ Una variabile **Condition** può essere associata ad un **Lock** e consente ad un thread di sospendere la propria esecuzione fino a quando sarà notificato (**notify**) da un altro thread del verificarsi di una determinata condizione.