Sincronizzazione fra thread

- Problemi della memoria condivisa fra thread
- Le sezioni critiche
- I Semafori
- Alcuni problemi di Sincronizzazione
- I Monitor

Background

- L’accesso a dati condivisi può generare inconsistenza nei dati.
- Per mantenere la consistenza nei dati abbiamo bisogno di meccanismi per assicurare che alcune operazioni dei thread cooperanti avvengano in un ordine corretto.
- Esempio: prenotare un posto in uno scompartimento di treno

Prenotazione posto

Due thread T1 e T2 cercano di prenotare un posto in uno scompartimento di un treno.

Abbiamo una variabile condivisa numero_posti=6

Cosa può succedere

Il primo thread prenota un posto:

- numero_posti=numero_posti-1 può essere implementato come
  register1 = numero_posti
  register1 = register1 -1
  numero_posti = register1

Il secondo thread cerca anche lui di prenotare un posto:

- numero_posti=numero_posti-1 può essere implementato come
  register2 = numero_posti
  register2 = register2 - 1
  numero_posti = register2
- Considera questa esecuzione:

  Passo 1: T1 esegue register1 = numero_posti {register1 = 6}
  Passo 2: T2 esegue register2 = numero_posti {register2 = 6}

  Passo 3: T1 esegue register1 = register1-1 {register1 = 5}
  Passo 4: T2 esegue register2 = register2-1 {register2 = 5}
  Passo 5: T1 esegue numero_posti= register1 {numero_posti = 5}
  Passo 6: T2 esegue numero_posti= register2 {numero_posti = 5}

Le sezioni critiche: proprietà

1. Mutua Esclusione – Se un processo P_i sta eseguendo la sua sezione critica, allora nessun altro processo può entrare nella sezione critica

2. Progresso – Se nessun processo sta eseguendo una sezione critica e esiste qualche processo che vuole entrare nella sezione critica, allora ad uno di tali processi deve essere consentito di entrare

3. Attesa Limitata - Deve esistere un limite al numero di volte che altri processi possono entrare nelle loro sessioni critiche dopo che un processo P_i ha fatto richiesta di entrare (in modo tale che P_i possa entrare nella sua sezione critica)

Algoritmo 1

- I Thread condividono una variabile interna turn
- Se turn==i, allora il thread i può entrare

- Codice per il Thread i

```
while (turno!=i) ;  // aspetta il suo turno
//esegue la sezione critica
numero_posti--;
turno = j;
//esegue la sezione non critica
```

- Questo algoritmo non soddisfa la proprietà di progresso
  - Perchè?

Algoritmo 2

- Utilizziamo una variabile booleana per indicare che il prcesso vuole entrare nella sezione critica

- Codice per il Thread i

```
pronto[i]=true;
while (pronto[j]) ;  // aspetta il suo turno
//esegue la sezione critica
numero_posti--;
pronto[i]=false;
//esegue la sezione non critica
```

- Questo algoritmo non soddisfa ancora la proprietà di progresso (si ha un’attesa infinita se entrambi mettono la booleana a true)

Algoritmo 3

- Combiniamo le idee degli algoritmi 1 e 2

- Stavolta soddisfa I requisiti della sezione critica?

Algoritmo 3 (1)

```
public class Algorithm_3
{
    private boolean flag0;
    private boolean flag1;
    private int turno;
    public Algorithm_3() {
      flag0 = false;
      flag1 = false;
      turno = 0;
    }
    // continua sull'altro lucido
```

Algoritmo 3 (2)

```
public void entroSezioneCritica(int t) {
  int altro = 1 - t;
  turno = altro;
  if (t == 0) {
        flag0 = true;
        while(flag1 == true && turno == altro)
              Thread.yield();
  }
  else {
        flag1 = true;
        while (flag0 == true && turno == altro)
              Thread.yield();
  }
}
```

// Continued on Next Slide

Algoritmo 3 – (3)

```
  public void lasciaSezioneCritica(int t) {
        if (t == 0)
              flag0 = false;
       else
              flag1 = false;
  }
}
```

Algoritmo del fornaio

- Se invece di avere due processi ne abbiamo n, si usa l’algoritmo del fornaio

- Ognuno ha un numerino, entra chi ha il numero giusto (se vuole).

Semafori

- Semaforo *S* –(equivale ad una variabile intera)
- 0 semaforo rosso , >0 semaforo verde
- Operazioni fondamentali:
  - wait (S) // aspetto al semaforo
    - equivale a while (S==0); S--;
  - signal(s) // libero il semaforo
    - equivale a S++

Il semaforo binario

- Semaforo Binario– S può vlere solo 0 oppure 1
- Risolve il problema della mutua esclusione

```
Semaforo_Binario S; // valore iniziale di default 1=verde
// codice del Thrad Ti
wait(S);
Esegui Sezione critica
signal(S);
```

Deadlock e Starvation

- Deadlock – due o più processi aspettano per un tempo infinito un evento che può essere causato da uno dei processi che aspettano.
- Esempio: incroncio a 4 vie imboccato contemporaneamente da 4 macchine: chi ha la precedenza?

- S e Q sono due semafori inizializzati a 1

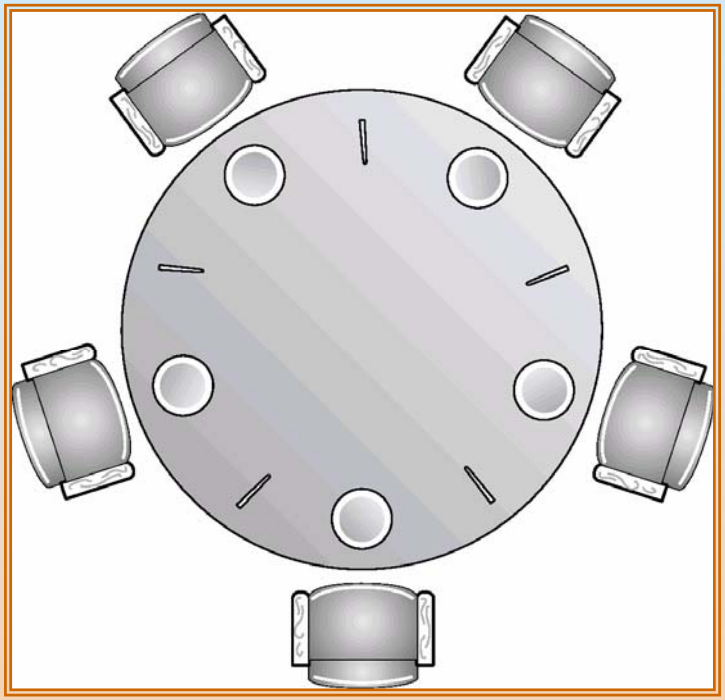
P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
signal(S);	signal(Q);
signal (Q);	signal (S);

- Starvation – Blocco infinito. Un processo non può mai essere rimosso dalla coda del semaforo in cui è bloccato.
- Esempio: cani affamati che non fanno mai mangiare il cane più debole.

Problemi classici di Sincronizzazione

- Buffer Limitato
- Lettori e Scrittori
- Il problema dei 5 filosofi

Problema dei 5 filosofi



- Ogni filosofo deve mangiare con due bacchette.
- Chi mangerà per primo?

Monitor (1)

- Costrutti di sicronizzazione di alto livello che consentono la condivisione sicura di un tipo di dati astratto tra processi concorrenti.

```
monitor monitor-name
{
        dichiarazione di variabili condivise
        procedure entry P1 (...) {
            . . .
        }
        procedure entry P2 (...) {
            . . .
        }
        procedure entry Pn (...) {
            . . .
        }
        monitor-name (...) {
            codice di inizializzazione
        }
}
```

Monitor (2)

- Per consentire ad un processo di attendere all'interno di un monitor, si deve dichiarare una variabile **condition**:

  condition x, y;

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

  - L'operazione

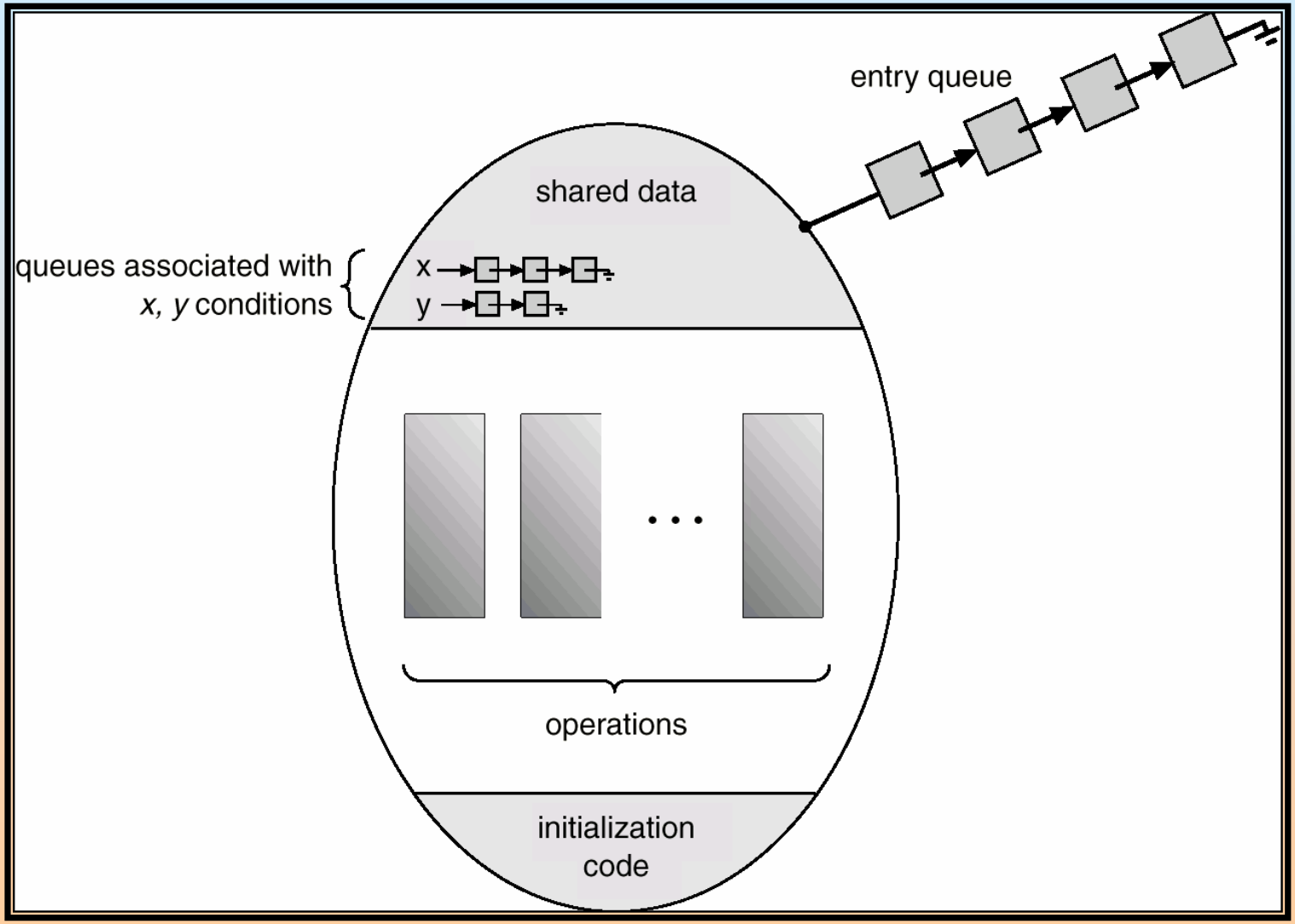
    x.wait();

    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 in Java (1)

Tutti i thread che fanno parte di una determinata applicazione Java condividono lo stesso spazio di memoria, quindi è possibile che più thread accedano contemporaneamente allo stesso metodo o alla stessa sezione di codice di un oggetto.

Per evitare inconsistenze, e garantire meccanismi di mutua esclusione e sincronizzazione, Java supporta la definizione di oggetti "**monitor**".

In Java, un monitor è l'istanza di una classe che definisce uno o più metodi **synchronized**. E' possibile definire anche soltanto una sezione di codice (ovvero un blocco di istruzioni) come "synchronized".

In Java ad ogni oggetto è automaticamente associato un "**lock**": per accedere ad un metodo o a una sezione synchronized, un thread deve prima acquisire il lock dell'oggetto.

Il lock viene automaticamente rilasciato quando il thread esce dal metodo o dalla sezione synchronized, o se viene interrotto da un'eccezione. Un thread che non riesce ad acquisire un lock rimane sospeso sulla richiesta della risorsa fino a quando il lock non diventa disponibile.

Monitor in Java (2)

Ad ogni oggetto contenente metodi o sezioni synchronized **viene assegnata una sola variabile condition**, quindi due thread non possono accedere contemporaneamente a due sezioni synchronized diverse di uno stesso oggetto.

L’esistenza di una sola variabile condition per ogni oggetto rende il modello Java meno espressivo di un vero monitor, che presuppone la possibilità di definire più sezioni critiche per uno stesso oggetto.