SISTEMI OPERATIVI

(MODULO DI INFORMATICA II)

Sincronizzazione dei processi

Prof. Luca Gherardi

Prof.ssa Patrizia Scandurra (anni precedenti)

Università degli Studi di Bergamo

a.a. 2012-13

Sommario

- Il problema della sincronizzazione
- Il problema della *sezione critica*
- Soluzioni per due processi
- Hardware per la sincronizzazione
- Semafori
- Problemi classici di sincronizzazione
- Monitor
- Esempi di sincronizzazione
- Transazioni atomiche

Il problema della sincronizzazione (1)

- Il funzionamento di un processo cooperante influenza ed è influenzato dal comportamento dei processi che cooperano con lui
- L'accesso concorrente a dati condivisi può portare alla loro inconsistenza
- Per mantenere la consistenza dei dati sono necessari dei meccanismi che assicurino l'esecuzione ordinata di processi cooperanti

Il problema della sincronizzazione (2)

- Ritorniamo all'esempio del produttore/consumatore
- Frammento di codice eseguito dal produttore con la chiamata *insert(item)* :

```
while (count == BUFFER_SIZE)
        ; // non fare nulla
// aggiungi un elemento al buffer
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
count ++;
```

Il problema della sincronizzazione (3)

Frammento di codice eseguito dal consumatore con la chiamata *remove()* :

```
while (count == 0)
        ; // non fare nulla
// rimuovi un elemento dal buffer
Item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
count --;
```

Il problema della sincronizzazione (4)

- count ++ può essere implementata in linguaggio macchina nel modo seguente:

  register1 = count

  register1 = register1 + 1

  count = register1

- count -- può essere implementata come segue:

  register2 = count

  register2 = register2 - 1

  count = register2

- Se le due istruzioni di alto livello vengono eseguite contemporaneamente le istruzioni a livello macchina possono essere interfogliate secondo qualsiasi ordine

Il problema della sincronizzazione (5)

- Si consideri questo ordine di esecuzione partendo da count = 5:

S0: produttore esegue register1 = count	{register1 = 5}
S1: produttore esegue register1 = register1 + 1	{register1 = 6}
S2: consumatore esegue register2 = count	{register2 = 5}
S3: consumatore esegue register2 = register2 - 1	{register2 = 4}
S4: produttore esegue count = register1	{count = 6 }
S5: consumatore esegue count = register2	**{count = 4} scorretto!**

Il problema della sincronizzazione (6)

- Si consideri questo ordine alternativo di esecuzione:

S0: produttore esegue register1 = count	{register1 = 5}
S1: produttore esegue register1 = register1 + 1	{register1 = 6}
S2: consumatore esegue register2 = count	{register2 = 5}
S3: consumatore esegue register2 = register2 - 1	{register2 = 4}
S4: consumatore esegue count = register2	{count = 4 }
S5: produttore esegue count = register1	**{count = 6} scorretto!**

Il problema della sincronizzazione (7)

- Se le istruzioni delle due chiamate di alto livello non fossero interfogliate otterremmo il risultato corretto:

S0: produttore esegue register1 = count	{register1 = 5}
S1: produttore esegue register1 = register1 + 1	{register1 = 6}
S2: produttore esegue count = register1	{count = 6}
S3: consumatore esegue register2 = count	{register2 = 6}
S4: consumatore esegue register2 = register2 - 1	{count = 5 }
S5: consumatore esegue count = register2	**{count = 5} corretto!**

- Occorre quindi assicurare che l'accesso alla variabili condivise sia eseguito da un processo alla volta
  - Si usano tecniche di sincronizzazione e coordinamento dei processi
  - Questo vale a maggior ragione nei moderni sistemi multicore

Corse critiche

- Si arriva a questi stati di inconsistenza quando permettiamo ai thread di accedere e manipolare dati condivisi in maniera concorrente
  - il risultato dell'esecuzione dipende dall'ordine in cui è avvenuto l'accesso

- Queste situazioni sono dette **corse critiche** (race condition) ed il segmento di codice in cui un thread accede a dati condivisi è detto **sezione critica** (critical section)

- Per evitarle, bisogna fare in modo che un solo thread per volta possa manipolare i dati condivisi, ovvero
  - occorre garantire che **l'esecuzione delle sezioni critiche** dei thread sia **mutuamente esclusiva nel tempo**

Protocolli di accesso alla sezione critica

- Questa condizione può essere garantita progettando dei **protocolli di accesso alla sezione critica**
  - I processi cooperanti adottano questo processo
- Un processo può accedere alla sua sezione critica solo dopo aver ricevuto un permesso
- È utile introdotte le seguenti sezioni

Il problema della sezione critica

Una soluzione del problema della sezione critica deve soddisfare queste condizioni

- **Mutua esclusione**: se un thread sta eseguendo la sua sezione critica, nessun altro thread può eseguire la propria
- **Progresso**: se nessun thread esegue la sua sezione critica e alcuni thread vogliono entrare nella loro, allora la decisione su chi sarà il prossimo a entrare nella propria sezione critica non può essere ritardata indefinitamente
  - E spetta ai processi che sono **fuori** dalle proprie sezione **non** critiche
- **Attesa limitata** (bounded waiting): esiste un limite al numero di volte in cui gli altri thread possono entrare nelle loro sezioni critiche dopo che un thread ha fatto richiesta di entrare nella propria e prima che tale richiesta sia esaudita
- **Velocità relative**: ciascun thread è in esecuzione a velocità non nulla, e non si può fare alcuna ipotesi sulle velocità relative degli n thread e dei processori

Kernel con e senza diritto di prelazione

- Il codice del kernel deve regolare l'accesso ai dati condivisi
  - Ad esempio deve garantire che un processo alla volta acceda alla lista dei file aperti
- I kernel possono essere distinti in due categorie in base a come gestiscono le sezioni critiche:
- Kernel con diritto di prelazione: permettono che un processo che sta lavorando in modalità kernel sia bloccato
- Kernel senza diritto di prelazione: non permettono che un processo che sta lavorando in modalità kernel sia bloccato
- La seconda categoria è immune ai problemi dovuti all'accesso condiviso ai dati (un solo processo può essere in esecuzione in modalità kernel)
- Tuttavia la prima categoria è preferibile per sistemi real-time e offre una maggiore prontezza nelle risposte

Variabile di lock

- Qualunque soluzione al problema della sezione critica richiede l'uso di un lock.
- Variabile condivisa che definisce **lo stato di uso di una risorsa**
- Cioè quando è in uso da parte di un processo (ovvero quando un processo è nella sua sezione critica)

Lock = 0 → **risorsa libera**

= 1 → **in uso**

```
do{
        acquisisce il lock
            sezione critica
        restituisce il lock
            sezione non critica
} while (true);
```

Variabile di turno (implementazione algoritmica del concetto di lock)

- Soluzioni algoritmiche: la sincronizzazione di due processi concorrenti può avvenire mediante **variabile di turno**

- Variabile condivisa che definisce **il turno di uso della risorsa**
- Cioè a quale processo o thread spetta il diritto di uso in un certo istante

Soluzioni per due processi in Java (1)

- Due thread, T0 and T1 (Ti e Tj con j = 1 - i)
- Implementano questa interfaccia **MutualExclusion**:

```
public interface MutualExclusion {
  public static final int TURN_0 = 0;
  public static final int TURN_1 = 1;
  public abstract void enteringCriticalSection(int
    turn);
  public abstract void leavingCriticalSection(int
    turn);
}
```

Prima di eseguire la propria sezione critica, ogni thread chiama il metodo *enteringCriticalSection()* passando il proprio valore di turno (o 0 o 1).

Un thread non terminerà il metodo *enteringCriticalSection()* sinchè non sarà in grado di entrare nella sua sezione critica **in modo esclusivo**.

Struttura generale di un thread (o processo) in Java

```
do {
    enteringCriticalSection()
        ... //sezione critica
    leavingCriticalSection()
    ... // sezione non critica
}while(true);
```

PROTOCOLLO DI ACCESSO ALLA SEZIONE CRITICA

Soluzioni per due processi (2)

- Il thread worker:

```
public class Worker implements Runnable {
  private String name;
  private int id;
  private MutualExclusion mutex;
  public Worker(String name, int id, MutualExclusion mutex) {
    this.name = name;
    this.id = id;
    this.mutex = mutex;
  }
  public void run() {
    while (true) {
      mutex.enteringCriticalSection(id);
      System.out.println("Thread name: " + name + "--- Critical
                  Section Accessed");
      mutex.leavingCriticalSection(id);
      System.out.println("Thread name: " + name + "--- Critical
                  Section Leaved");
  }}
}
```

Soluzioni per due processi (3)

- Utilizziamo la classe **AlgorithmFactory** per creare due thread e per testare ciascun algoritmo

```
public class AlgorithmFactory {
  public static void main(String args[]) {
    MutualExclusion alg = new Algorithm_3();
    Thread first = new Thread(new Worker("Worker
          0", 0, alg));
    Thread second = new Thread(new Worker("Worker
          1", 1, alg));
    first.start();
    second.start();
  }
}
```

Soluzioni per due processi (4)
Algoritmo 1

`volatile`: indica che la variabile sarà modificata da più thread

`Thread.yield()`: il thread dichiara che è disposto a cedere il controllo della CPU

```
public class Algorithm_1 implements MutualExclusion
  {
   private volatile int turn;
   public Algorithm_1() {
     turn = TURN_0;
   }
   public void enteringCriticalSection(int t) {
     while (turn != t)
           Thread.yield(); //attesa attiva
   }
   public void leavingCriticalSection(int t) {
     turn = 1 - t;
   }
}
```

Garantisce mutua esclusione
Impone stretta alternanza dei processi
Non garantisce progresso

Soluzioni per due processi (5)
Algoritmo 1

- Possibile esecuzione

- T1 → `mutex.enteringCriticalSection(0);`
  - All'inizio `turn` ha lo stesso valore dell'ID di T1 (0), quindi T1 accede alla sezione critica
- T2 → `mutex.enteringCriticalSection(1);`
  - `turn` ha lo stesso valore dell'ID di T1 (0), quindi T2 non accede alla sezione critica ma chiama `Thread.yield()`
- T1 → `System.out.println(…);`
- T1 → `mutex.leavingCriticalSection(0);`
  - `turn` viene settata allo stesso valore dell'ID di T2 (1), quindi da questo momento solo T2 potrà accedere alla sezione critica. I tentativi di T1 finiranno con la chiamata `Thread.yield()`

Soluzioni per due processi (6)
Algoritmo 1

- Possibile esecuzione

- T1 → `mutex.enteringCriticalSection(0);`
  - All'inizio `turn` ha lo stesso valore dell'ID di T1 (0), quindi T1 accede alla sezione critica
- T1 → `System.out.println(…);`
- T1 → `mutex.leavingCriticalSection(0);`
  - `turn` viene settata allo stesso valore dell'ID di T2 (1), quindi da questo momento solo T2 potrà accedere alla sezione critica. I tentativi di T1 finiranno con la chiamata `Thread.yield()`
- T2 → `mutex.enteringCriticalSection(1);`
  - `turn` ha lo stesso valore dell'ID di T2 (1), quindi T2 accede alla sezione critica
- T2 → `System.out.println(…);`
- T2 → `mutex.leavingCriticalSection(1);`
  - `turn` viene settata allo stesso valore dell'ID di T1 (0), quindi da questo momento solo T1 potrà accedere alla sezione critica. I tentativi di T2 finiranno con la chiamata `Thread.yield()`

Soluzioni per due processi (7)
Algoritmo 1

- I thread condividono la variabile intera `turn`
- Se `turn==i`, allora il thread Ti è autorizzato ad eseguire la sua sezione critica
- Questa soluzione soddisfa la mutua esclusione ma **non la condizione di progresso**, perché?
- Poiché impone una stretta alternanza dei thread
  - Ad es. se turn == 0 e T1 è pronto ad entrare nella sua sezione critica, non può farlo, anche se T0 è nella sua sezione non critica e non ha necessita di accedere alla sezione critica
  - Promemoria - Progresso: se nessun thread è nella propria sezione critica e un thread vuole entrare nella sua sezione critica, la richiesta non può essere rimandata indefinitamente

Soluzioni per due processi (8)
Algoritmo 2

- Maggiori informazioni sullo stato di ciascun thread
  - Un *flag* (variabile booleana) indica se il thread associato è o non è pronto ad entrare nella sua sezione critica

Soluzioni per due processi (9)
Algoritmo 2

```
public class Algorithm_2 implements MutualExclusion{

   private volatile boolean flag0, flag1;
   public Algorithm 2() {
      flag0 = false; flag1 = false;
   }

   public void enteringCriticalSection(int t) {
      if (t == 0) {
         flag0 = true;
         while(flag1 == true)
           Thread.yield();
      }
      else {
         flag1 = true;
         while (flag0 == true)
           Thread.yield();
      } }
        // Continua …
```

Soluzioni per due processi (10)
Algoritmo 2

```
public void
  leavingCriticalSection(int t) {
   if (t == 0)
      flag0 = false;
   else
      flag1 = false;
   }
} //End class
```

- La condizione di progresso non è ancora rispettata, perché?

- Risp: ciascun thread può rimanere in **attesa all'infinito**
- Supponiamo, ad es., che T0 pone flag0=true, ma che prima di eseguire il ciclo while, avviene un cambio di contesto a favore di T1 che pone flag1=true
  - Ne segue che ciascun thread eseguirà il ciclo while all'infinito poiché troverà il flag dell'altro thread sempre a true

Non impone stretta alternanza dei processi
Non garantisce progresso
Possibile attesa infinita

Soluzioni per due processi (11)
Algoritmo 3 - Soluzione di Peterson

- I processi condividono i seguenti dati:
  - `int turno` – stabilisce di chi è il turno
  - `boolean flag[2]` -- indica se un processo è pronto a entrare nella sezione critica
- Quando un processo cerca di entrare nella sezione critica concede la precedenza all'altro processo

```
//Processo Pi (l'altro è Pj con j = 1 – i)
do {
        flag[i] = true;
        turno = j;                          sezione d'ingresso
        while ( flag[j] && turno==j );
        sezione critica
        flag[i] = false;                    sezione d'uscita
        sezione non critica
}while(true);
```

Soluzioni per due processi (12)
Algoritmo 3 - Soluzione di Peterson

- Può essere inteso come combinazione degli algoritmi 1 e 2

```
public class Algorithm_3 implements MutualExclusion{
  private volatile boolean flag0;
  private volatile boolean flag1;
  private volatile int turn;
  public Algorithm_3() {
    flag0 = false;
    flag1 = false;
    turn = TURN_0;
  }
  //continua …
```

Soluzioni per due processi (13)

Algoritmo 3 - Soluzione di Peterson

```
public void enteringCriticalSection(int t) {
  int other = 1 - t;
  turn = other;
  if (t == 0) { //thread 0
    flag0 = true;
    while(flag1 == true && turn == other)
      Thread.yield(); //attesa attiva
  }
  else { //thread 1
    flag1 = true;
    while (flag0 == true && turn == other)
      Thread.yield(); //attesa attiva
  }
}//continua …
```

Soluzioni per due processi (14)
Algoritmo 3 - Soluzione di Peterson

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

Garantisce mutua esclusione
Garantisce progresso
Attesa limitata

Soluzioni per due processi (15)
Algoritmo 3 - Soluzione di Peterson

- Possibile esecuzione

- T1 → `mutex.enteringCriticalSection(0);`
  - `other=1, turn=1 flag0=true`
  - Essendo all'inizio `flag1==false` T1 riesce ad entrare nella sezione critica
- T2 → `mutex.enteringCriticalSection(1);`
  - `other=0, turn=0 flag1=true`
  - Essendo `flag0==true` (In quanto T1 è nella sezione critica), T2 non riesce ad accedere e chiama `Thread.yield(…);`
- T1 → `mutex.leavingCriticalSection(0);`
  - `flag0=false`
  - Da questo momento solo T2 potrà accedere alla sezione critica in quanto `flag1==true`
  - È garantito il rispetto della condizione di **progresso**
  - Non c'è più stretta alternanza tra i processi

Hardware per la sincronizzazione (1)

- Molti SO forniscono istruzioni HW per risolvere il problema della sezione critica
- Monoprocessori: possono **disabilitare gli interrupt** (sistemi senza diritto di prelazione)
  - Sequenza di istruzioni della "sezione critica" eseguita in ordine senza interruzioni
  - Di solito troppo inefficiente sui sistemi multiprocessore
    - disabilitazione/riabilitazione degli interrupt su ogni processore
  - I SO che lo usano sono carenti in scalabilità
    - L'inefficienza aumenta all'aumentare del numero di CPU

Hardware per la sincronizzazione (2)

- I calcolatori moderni forniscono **istruzioni hardware speciali che funzionano in modo atomico**
  - Atomico = non-interrompibile
  - Esempi:
    - Controllare e modificare il contenuto di una parola `TestAndSet()`
    - Scambiare il contenuto di due parole `Swap()`

Uso ad interruzioni disabilitate

- **Acquisizione della risorsa**:
  - disabilito le interruzioni
  - leggo la variabile di lock
  - se la risorsa è libera (lock=0), la marco in uso ponendo lock=1 e riabilito le interruzioni
  - se la risorsa è in uso (lock=1), riabilito le interruzioni e pongo il processo in attesa che la risorsa si liberi
- **Rilascio della risorsa**:
  - pongo lock=0

Uso dell'istruzione TestAndSet (1)

```
boolean TestAndSet(boolean *obiettivo) {
    boolean valore = *obiettivo;
    *obiettivo = true;
    return valore;
}
```

```
do {
    while (TestAndSet(&lock));
        sezione critica
    lock = false;
        sezione non critica
} while (true);
```

Salve il valore di `lock` in `valore`

Setta `lock` a `true`

Restituisce `valore`, se `lock` era falso entra nella sezione critica

Altrimenti non è cambiato nulla e continua ad attendere

Uso dell'istruzione Swap

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

```
do {

   chiave = true;
   while (chiave == true)
      Swap(&lock, &chiave);

   sezione critica

   lock = false;

   sezione non critica

} while (true);
```

lock: variabile globale
chiave: variabilie locale

Uso dell'istruzione TestAndSet (1)

- Le soluzioni precedenti risolvono il problema della mutua esclusione ma non quello dell'attesa limitata
- Per soddisfare tutte le 4 condizioni è necessario introdurre per ogni processo una nuova variabile booleana `attesa` che viene inizializzata a `false`, cosi come `lock`
- Quando un processo esce dalla sezione critica setta a `false` il campo attesa di un altro processo che è in attesa. In questo esso può accedere alla sezione critica. Questo garantisce la mutua esclusione e il progresso.
- Il processo in uscita scandisce i processi nell'ordine `i+1,i+2,…,n-1,0,…,i-1` e cerca il primo processo in attesa. Ciò garantisce che un processo può attendere al massimo n-1 turni. L'attesa limitata è quindi garantita.

```
do {
    attesa[i] = true;
    chiave = true;
    while (attesa[i] && chiave)
       chiave = TestAndSet(&lock);
    attesa[i] = false;

        sezione critica

    j = (i + 1) % n;
    while ((j != i) && !attesa[i])
      j = (j + 1) % n;

    if (j == i)
      lock = false;
    else
      attesa[j] = false;

        sezione non critica

} while (true);
```

Obiettivo del SO

- Alzare il livello di astrazione portando la gestione della sincronizzazione in **funzioni del sistema operativo**
- Garantire la corretta gestione della sincronizzazione e dell'accesso alle variabili di supporto alla mutua esclusione
- Evitare usi errati delle operazioni di abilitazione e disabilitazione delle interruzioni o degli assegnamenti alle variabili di turno o lock

Semafori

- Meccanismi a livello di funzioni SO che garantiscono un uso corretto delle risorse da parte del programmatore

- Concetto di semaforo:
  - Variabile intera
    - semaforo **binario** (0 o 1) -- **mutex lock**
    - semaforo **generalizzato** (contatore) -- il valore può variare in un dominio senza restrizioni (tipicamente usate per controllare l'accesso ad una risorsa presente in un numero finito di unità)

Semaforo binario

- Un semaforo binario S è una variabile binaria che rappresenta lo stato di uso della risorsa condivisa

  `s = 1` → risorsa libera

  `0` → risorsa in uso

- Il semaforo S è manipolato dalle funzioni:
  - **`acquire(s)`** → acquisisce l'uso della risorsa
  - **`release(s)`** → rilascia la risorsa

- **`acquire`** e **`release`** sono **operazioni atomiche** poiché sono procedure di sistema

Uso del semaforo binario

- Per l'accesso alla sezione critica:
  - `Semaphore S;` //Assumendo che sia inizializzato ad 1
  - `acquire(s);`
  - `criticalSection();`
  - `release(s);`

Semaforo generalizzato

- Un semaforo generalizzato *S* è una variabile intera che rappresenta lo stato di uso di un insieme di risorse omogenee (istanze di una stessa risorsa) condivise

  `S = n` → n risorse libere

  `0` → in uso

- Il semaforo *S* è manipolato dalle funzioni:
  - **`acquire(s)`** → acquisisce l'uso di una risorsa
  - **`release(s)`** → rilascia la risorsa in uso

- Tali funzioni sono spesso chiamate anche **`wait(s)`** e **`signal(s)`** risp. (anche nel libro di teso)

Implementazione dei semafori – con attesa attiva (busy waiting)

```
acquire(S) {
  while S <= 0   //un'attesa attiva (spin lock)
        ;        //nessuna operazione; si cicla
                   continuamente in caso
                   di risorsa non disponibile
  S--;
}

release(S) {
    S++;
}
```

Il busy waiting

- **Contro:** nei sistemi multi-tasking spreca cicli di CPU che altri processi potrebbero usare in modo produttivo!
- **Pro:** non richiede cambio di contesto, quindi è più vantaggioso se i blocchi durano poco e/o nei sistemi multi-processing

Implementazione dei semafori – con sospensione e rischedulazione (1)

- **Strutture dati**: ogni semaforo S ha una **coda dei processi in attesa** di acquisire la risorsa

```
Typedef struct{
  int value;
  struct process *list;
} semaforo;
```

- **`acquire(s)`** sospende il processo in esecuzione in caso di risorsa non disponibile e lo inserisce nella coda di attesa del semaforo
- **`release(S)`** rilascia la risorsa e riattiva il primo processo della coda di attesa cedendogli la risorsa
- **Lo schedulatore** dei processi in attesa della risorsa definisce l'ordine di ottenimento della risorsa in base alla politica adottata per il semaforo

Implementazione dei semafori – con sospensione e rischedulazione (2)

```
acquire(s){
  value --;
  if (value< 0) {
    add this process to list(s)
    block();
  }
}
```

Implementazione dei semafori – con sospensione e rischedulazione (2)

```
release(s){
  value++;
  if (value <= 0) {
    remove a process p from list(s)
    Wakeup(p);
  }
}
```

- Il SO deve fornire due nuove chiamate di sistema di base
  - **`block`**: sospende il processo che la invoca
  - **`wakeup(p)`**: riprende l'esecuzione del processo bloccato P

Implementazione dei semafori

- Occorre ancora garantire che due processi **non** possano eseguire una **acquire() e**d una **release() sullo stesso semaforo nello stesso momento!**
- Quindi l'implementazione crea un nuovo problema di sezione critica
  - il SO ricorre a tecniche di più basso livello come la disabilitazione/riabilitazione degli interrupt e le istruzioni hardware per la sincronizzazione

Stalli e starvation con i semafori (2)

- **Stallo (deadlock)** – due o più processi aspettano un evento che può essere causato solo da uno dei processi in attesa
  - nel nostro caso una release
- Consideriamo un sistema costituito da due processi P_0 e P_1 ciascuno dei quali accede a due semafori S e Q valorizzati a 1

P_0	P_1
acquire(S);	acquire(Q);
acquire(Q);	acquire(S);
.	.
.	.
.	.
release(S);	release(Q);
release(Q);	release(S);

Vedi DeadlockTest.java

Stalli e starvation con i semafori (1)

- **Blocco indefinito (starvation)** – i processi attendono indefinitamente dentro il semaforo
  - Un processo può non essere mai rimosso dalla lista del semaforo nella quale è sospeso (ad es. se gestiamo la lista con una politica LIFO)

Problemi classici di sincronizzazione

- Problema del **buffer limitato** (Producer-consumer)
- Problema dei **lettori-scrittori**
- Problema del **pranzo dei filosofi**

Problemi pensati per rappresentare tipiche situazioni di rischio

- Stallo con blocco (starvation)
- Stallo senza blocco (deadlock)
- Esecuzioni non predicibili (race condition)

- E testare nuovi meccanismi di sincronizzazione

Problema del buffer limitato
Soluzione tramite semafori (1)

Si definiscono

- Un buffer di N locazioni (capacità), ciascuna in grado di contenere un oggetto
- Un semaforo `mutex` inizializzato ad 1 che garantisce la mutua esclusione nell'accesso al buffer
- Un semaforo `full` inizializzato a 0
- Un semaforo `empty` inizializzato al valore N

Problema del buffer limitato

Soluzione tramite semafori (2)

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
       // buffer is initially empty
       in = 0;
       out = 0;
       buffer = new Object[BUFFER_SIZE];

       mutex = new Semaphore(1);
       empty = new Semaphore(BUFFER_SIZE);
       full = new Semaphore(0);
    }
```

Problema del buffer limitato

Soluzione tramite semafori (3)

- Metodo insert(item) – eseguito dal produttore

```
public void insert(Object item) {
    empty.acquire();
    mutex.acquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}
```

Problema del buffer limitato

Soluzione tramite semafori (4)

- Metodo remove() – eseguito dal consumatore

```
public Object remove() {
    full.acquire();
    mutex.acquire();

    // remove an item from the buffer
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}
```

Problema del buffer limitato

Soluzione tramite semafori (5)

- Il produttore

```
public class Producer implements Runnable
{
   private Buffer buffer;

   public Producer(Buffer buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();
         // produce an item & enter it into the buffer
         message = new Date();
         buffer.insert(message);
      }
   }
}
```

Problema del buffer limitato

Soluzione tramite semafori (6)

- Il consumatore

```
public class Consumer implements Runnable
{
   private Buffer buffer;

   public Consumer(Buffer buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();
         // consume an item from the buffer
         message = (Date)buffer.remove();
      }
   }
}
```

Bounded Buffer Problem (Cont.)

- La classe Factory

```
public class Factory
{
    public static void main(String args[]) {
      Buffer buffer = new BoundedBuffer();

      // now create the producer and consumer threads
      Thread producer = new Thread(new Producer(buffer));
      Thread consumer = new Thread(new Consumer(buffer));

      producer.start();
      consumer.start();
    }
}
```

Il problema dei lettori-scrittori (1)

- Una base di dati è condivisa fra vari processi concorrenti
  - **Lettori**: accedono solo in lettura; non modificano la base di dati
  - **Scrittori**: possono sia leggere che scrivere

- **Problema:**
  - permettere a lettori **multipli** di leggere contemporaneamente (non crea interferenze)
  - permettere ad **un solo scrittore** alla volta di accedere alla base di dati (accesso esclusivo)

Il problema dei lettori-scrittori (2)

- Esistono diverse varianti:

- Variante 1 (*priorità ai lettori*):
  - nessun lettore deve essere tenuto in attesa, a meno che uno scrittore abbia già ottenuto l'accesso alla base di dati condivisa, o anche
  - Nessun lettore deve essere messo in attesa solo perché uno scrittore sta aspettando di scrivere
    - È soggetta però al problema della starvation dei scrittori
- Variante 2 (*priorità agli scrittori*):
  - Uno scrittore deve poter svolgere la scrittura il più presto possibile, o anche
  - Nessun nuovo lettore deve iniziare la lettura se uno scrittore è in attesa di scrivere
    - È soggetta però al problema della starvation dei lettori

Problema dei lettori-scrittori (variante 1)
Soluzione tramite semafori (1)

Si definiscono

- La base di dati
- Un semaforo `db` inizializzato a 1, per la mutua esclusione sulla base di dati (è usato dagli scrittori, dal primo lettore che accede alla sezione critica e dall'ultimo che lascia la sezione critica)
- Una variabile intera `readerCount` inizializzata a 0
- Un semaforo `mutex` inizializzato a 1, per la mutua esclusione sulla variabile `readerCount`

Problema dei lettori-scrittori (variante 1)
Soluzione tramite semafori (2)

- Interfaccia per i metodi che implementano i read-write lock

```
public interface RWLock
{
   public abstract void acquireReadLock();
   public abstract void acquireWriteLock();
   public abstract void releaseReadLock();
   public abstract void releaseWriteLock();
}
```

Problema dei lettori-scrittori (variante 1)
Soluzione tramite semafori (3)

```
public class Database implements RWLock
{
    private int readerCount;
    private Semaphore mutex;
    private Semaphore db;
    public Database() {
            readerCount = 0;
            mutex = new Semaphore(1);
            db = new Semaphore(1);
    }
    public int acquireReadLock() { /* slide successiva */ }
    public int releaseReadLock() {/* slide successiva */ }
    public void acquireWriteLock() {/* slide successiva */ }
    public void releaseWriteLock() {/* slide successiva */ }
}
```

Problema dei lettori-scrittori (variante 1)
Soluzione tramite semafori (4)

```
public void acquireReadLock() {
    mutex.acquire();
    ++readerCount;
    // se sono il primo lettore dite agli altri
    // che il database viene letto
    if (readerCount == 1)
        db.acquire();
    mutex.release();
}
```

```
public void releaseReadLock() {
    mutex.acquire();
    --readerCount;
    // se sono l'ultimo lettore dite agli altri
    // che il database non viene più letto
    if (readerCount == 0)
        db.release();
    mutex.release();
}
```

Priorità ai lettori: Il primo lettore esegue un'operazione di acquire sul db, prevenendo che qualunque scrittore entri nella base di dati

Problema dei lettori-scrittori (variante 1)
Soluzione tramite semafori (5)

```
public void acquireWriteLock() {
    db.acquire();
}

public void releaseWriteLock() {
    db.release();
}
```

Problema dei lettori-scrittori (variante 1)
Soluzione tramite semafori (6)

- Il lettore

```
public class Reader implements Runnable
{
    private RWLock db;

    public Reader(RWLock db) {
       this.db = db;
    }

    public void run() {
       while (true) {
          // nap for awhile
          SleepUtilities.nap();

          db.acquireReadLock();

          // you have access to read from the database
          SleepUtilities.nap();

          db.releaseReadLock();
       }
    }
}
```

Problema dei lettori-scrittori (variante 1)
Soluzione tramite semafori (7)

- Lo scrittore

```
public class Writer implements Runnable
{
    private RWLock db;

    public Writer(RWLock db) {
       this.db = db;
    }

    public void run() {
       while (true) {
          // nap for awhile
          SleepUtilities.nap();

          db.acquireWriteLock();

          // you have access to write to the database
          SleepUtilities.nap();

          db.releaseWriteLock();
       }
    }
}
```

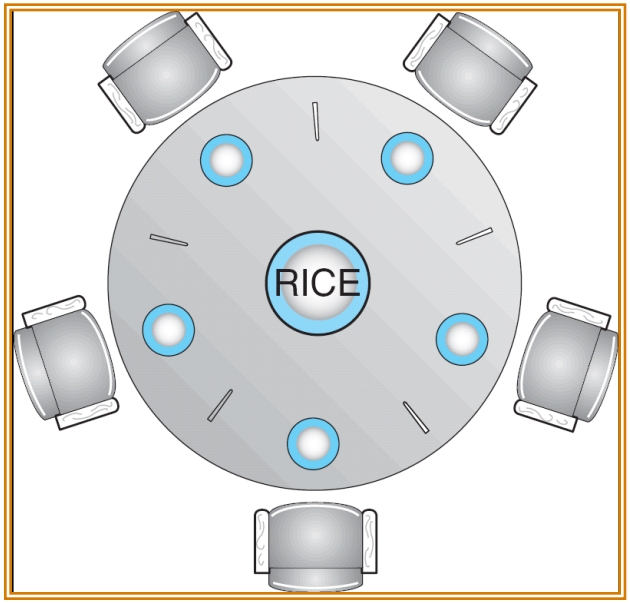
Problema dei lettori-scrittori
Lock di lettura-scrittura

- Alcuni SO forniscono un particolare tipo di lock per affrontare questo problema: i lock di lettura-scrittura

- Se un processo deve solo leggere acquisisce il lock in modalità sola lettura
  - Più processi possono avere questo lock
- Se un processo deve anche scrivere acquisisce il lock in modalità scrittura
  - Solo un processo può avere questo lock (e nessuno deve avere quello in modalità lettura)

Il problema dei filosofi a cena

- 5 filosofi su 5 sedie
- 5 bacchette per prendere il riso posto in una zuppiera condivisa
- Un filosofo può prendere una bacchetta alla volta, solo se è libera e se è posta tra lui e un vicino
- Quando il filosofo non mangia pensa e non interagisce con gli altri
- Quando il filosofo ha due bacchette mangia, dopo di che lascia le bacchette e torna a pensare
- La soluzione prevede l'uso di un array di 5 semafori
  - `chopStick[5]` inizializzati a 1



Problema dei filosofi a cena
Soluzione tramite semafori

- Implementazione del comportamento del filosofo

```
while (true) {
    // prendi la bacchetta di sinistra
    chopStick[i].acquire();
    // prendi la bacchetta di destra
    chopStick[(i + 1) % 5].acquire();
    eating();
    // rendi la bachetta di sinistra
    chopStick[i].release();
    // rendi la bacchetta di destra
    chopStick[(i + 1) % 5].release();
    thinking();
}
```

E' però **soggetta a deadlock** quando tutti i filosofi hanno fame e prendono contemporaneamente la forchetta di sinistra

Il problema dei filosofi a cena – Soluzioni che evitano i deadlock

- Permettere al più a 4 filosofi di sedersi contemporaneamente al tavolo
- Permettere ad un filosofo di **prendere le forchette solo se entrambe disponibili** (la raccolta deve avvenire all'interno di una sezione critica!)
- Soluzione asimmetrica: ad es. prevedendo che un filosofo dispari prenda prima la forchetta di sx e poi quella di dx, mentre uno pari raccoglie prima quella di dx e poi quella sx

Accesso alla sezione critica con i mutex lock

- Ricordiamo lo schema:

```
while (true) {
        acquire lock        mutex.acquire()
            critical section
        release lock        mutex.release()
            remainder section
}
```

Altri usi dei dei semafori per ordinare i flussi di esecuzione

- Si considerino due processi (thread) concorrenti:
  - P1 che esegue in sequenza due istruzioni `s1` e `s2`
  - P2 che esegue `s3`.
  - Supponiamo di volere **eseguire `s3` solo dopo che `s1` è terminata**, e **`s2` solo dopo che `s3` sia terminata**.
- **Soluzione**: Siano `mutex1` e `mutex2` semafori binari inizializzati a 0 (risorsa in uso) e condivisi da P1 e da P2.

```
Istruzioni di P1:  s1;                  Istruzioni di P2: acquire(mutex1);
                   release(mutex1);                       s3;
                   acquire(mutex2);                       release(mutex2);
                   s2;
```

- Poiché mutex1 è inizializzato a 0, P2 esegue `s3` solo dopo che P1 ha eseguito `release(mutex1)`, che si trova dopo `s1`. Un discorso simile vale per `mutex2`.

Possibili errori con i semafori per accedere alla sezione critica

- Scambiando l'ordine di acquire/release, vari thread possono accedere simultaneamente alla propria sezione critica!

```
mutex.release()
criticalSection()
mutex.acquire()
```

- Si ha uno stallo con:

```
mutex.acquire()
criticalSection()
mutex.acquire()
```

- Omettendo acquire o release o entrambe, si viola la mutua esclusione o si causa uno stallo

Problemi legati all'uso dei semafori

- L'uso dei semafori può portare ad errori che si manifestano solo in determinate circostanze
- Errori di programmazione
  - Violazioni della mutua esclusione
  - Attese infinite

- La responsabilità della **correttezza è lasciata al programmatore**

- Il SO non ha potere di controllo e gestione

- Motivo?
  Le primitive relative ai semafori sono chiamate di SO e come tali operano solo se chiamate in modo corretto

I Monitor

- **Obiettivo**
  - Innalzare il livello di astrazione per la gestione della sincronizzazione forzandone l'uso corretto
- **Come?**
  - Formulando un costrutto di sincronizzazione formulato a livello di linguaggio di programmazione

- **Monitor**
  - è un costrutto linguistico
  - trasformato nelle corrette chiamate di sistema dal compilatore del linguaggio

Monitor come costrutto linguistico

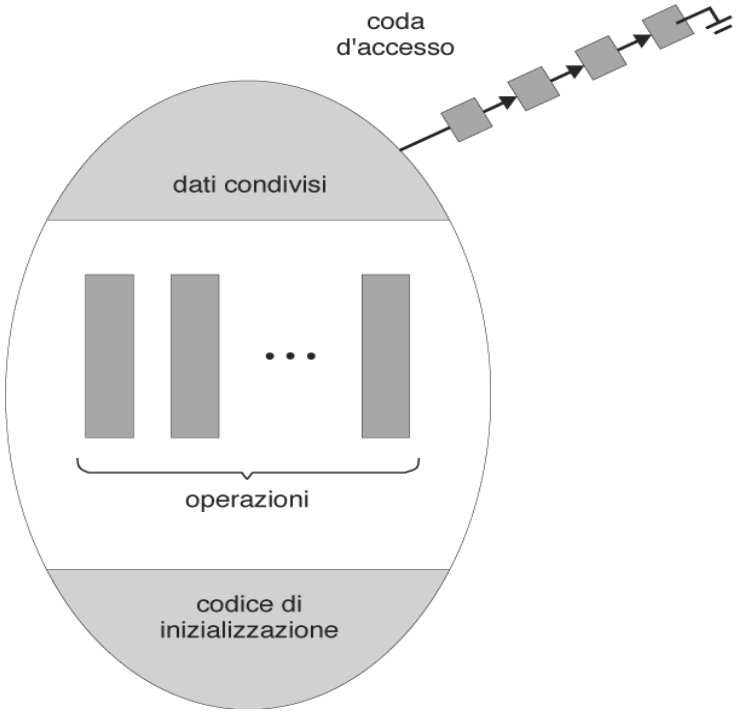
- Tipo astratto che incapsula dati privati mettendo a disposizione dei metodi pubblici per operare su tali dati
- I metodi sono contraddistinti dalla mutua esclusione
- Le variabili che definiscono lo stato di un'istanza del monitor
- Le variabili locali sono visibili solo dalle procedure interne

```
monitor monitor-name {
    // variable declarations
    public entry p1(…) {…}
    public entry p2(…) {…}
}
```

Condivise tra i thread (processi)

Visione schematica del monitor

- Il costrutto monitor assicura che al suo interno possa essere attivo un solo processo per volta (i.e. un solo processo può invocare i metodi)
  - Ciò garantisce la mutua esclusione
- Gli altri processi sono in una cosa di attesa



Le variabili “condizioni” (*condition*)

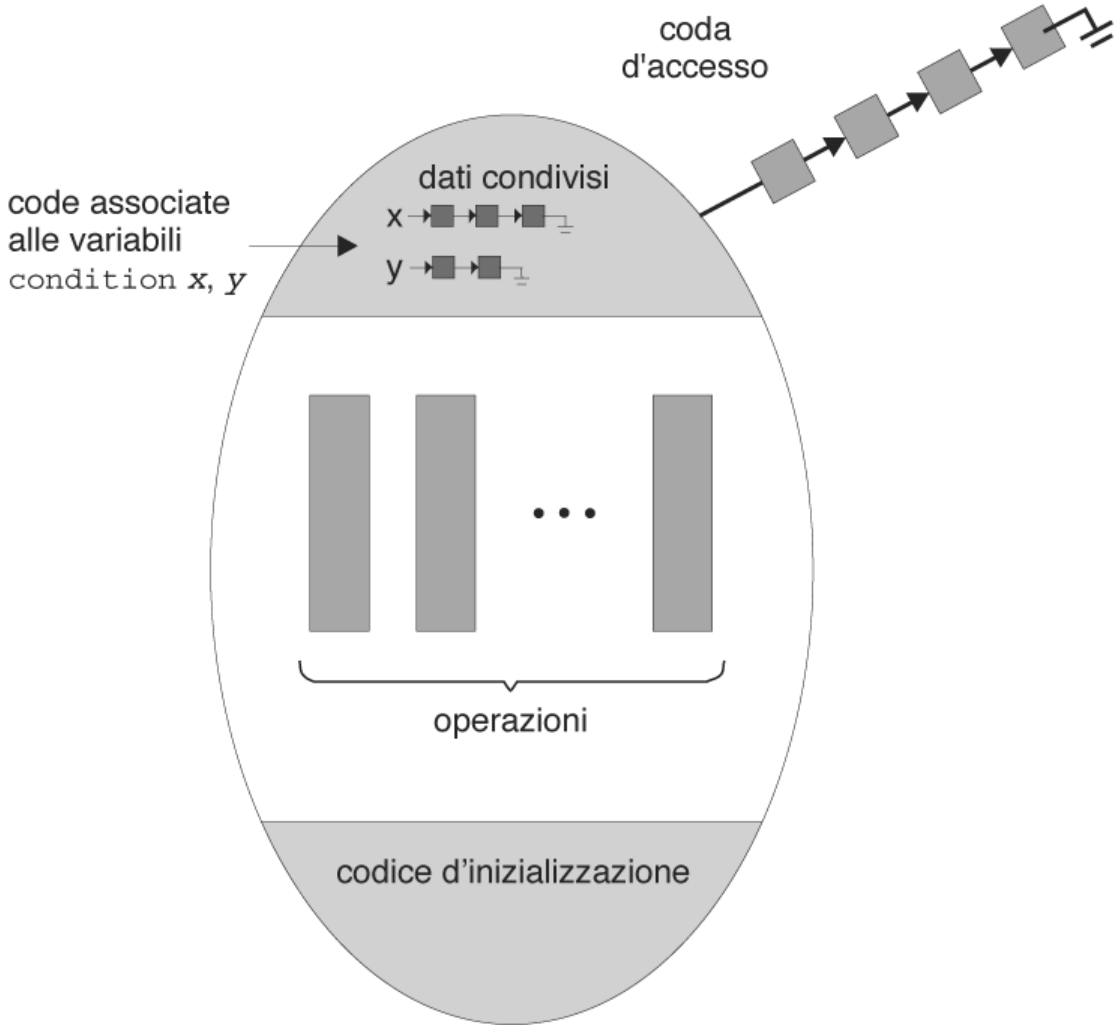
- Questa proposta di monitor garantisce solo la mutua esclusione
- I monitor permettono di esprimere degli schemi di sincronizzazione complessi mettendo a disposizione il costrutto **condizione**

```
condition x, y;
```

- Due operazioni possibili su una variabile “condition”:
  - `x.wait()` – il thread che la invoca viene sospeso nella coda di x
  - `x.signal()` – risveglia *esattamente* un thread in attesa nella coda di x (se ce ne sono, altrimenti lo stato di x rimane invariato)

- Un thread che invoca `x.wait()` è sospeso finché un altro thread non invoca `x.signal()`
- Le due operazioni non sono invocate direttamente ma attraverso i metodi pubblici

Monitor con le variabili “condizioni”



Realizzazione dei monitor

- La chiamata `x.signal()` differisce da quella dei semafori
  - Nei semafori l'operazione `signal()` influisce sempre sullo stato del semaforo
  - Nei monitor solo se c'è un processo in attesa della condizione
- Cosa succede se un thread P invoca `x.signal()` e c'è un thread Q nella coda di x? Quale processo rimane attivo all'interno del monitor?
- Possibili politiche di implementazione:
  - Segnala e aspetta:
    P attende finchè Q lascia il monitor, o aspetta una condizione (in una coda di una variabile "condition")
  - Segnala e continua:
    Q attende finchè P lascia il monitor o aspetta una condizione
  - Segnala e lascia (in *Concurrent Pascal*):
    P esegue x.signal(), lascia immediatamente il monitor e Q viene risvegliato

Problema dei filosofi a cena
Soluzione tramite monitor (1)

- Questa soluzione evita i deadlock lasciando prendere le bacchette solo se entrambe sono libere
- Un filosofo può essere in tre stati: `pensa, affamato, mangia`
- È definita una condizione `auto` per ogni filosofo:
  - Permette al filosofo di ritardare se stesso quando ha fame ma le bacchette non sono libere
- Il problema viene risolto controllando la distribuzione delle bacchette tramite monitor

Problema dei filosofi a cena - Soluzione tramite monitor (2)

```
monitor fc
{
  enum {pensa, affamato, mangia} stato[5];
  condition auto[5];

  prende(int i) {
    stato[i] = affamato;
    verifica(i);
    if (stato[i] != mangia)
      auto[i].wait();
  }

  posa(int i) {
    stato[i] = pensa;
    verifica((i + 4) % 5);
    verifica((i + 1) % 5);
  }

  verifica(int i) {
    if((stato[(i + 4) % 5] =! mangia) &&
      (stato[i] == affamato) &&
      (stato[(i + 1) % 5] != mangia)) {
        stato[i] = mangia;
        auto[i].signal();
    }
  }

  codice di inizializzazione() {
    for (int i = 0; i < 5; i++)
      stato[i] = pensa;
  }
}
```

Problema dei filosofi a cena - Soluzione tramite monitor (3)

```
prende(int i) {
   stato[i] = affamato;
   verifica(i);
   if (stato[i] != mangia)
     auto[i].wait();
}

posa(int i) {
   stato[i] = pensa;
   verifica((i + 4) % 5);
   verifica((i + 1) % 5);
}
```

```
verifica(int i) {
   if((stato[(i + 4) % 5] =! mangia) &&
      (stato[i] == affamato) &&
      (stato[(i + 1) % 5] != mangia)) {
        stato[i] = mangia;
        auto[i].signal();
   }
}
```

- Si noti che un filosofo i prima di iniziare a mangiare chiama auto[i].signal()
  - In questo caso nessun processo era in attesa della condizione auto[i] quindi non succede nulla
- Allo stesso modo un filosofo i, una volta finito di mangiare, chiama signal() sulle condizioni auto dei due filosofi vicini
  - Questo risveglia i folosofi qualora fossero in attesa delle bacchette

Problema dei filosofi a cena
Soluzione tramite monitor (4)

- Ogni filosofo deve invocare il metodo `prende` prima di mangiare e il metodo `posa` una volta terminato

```
fc.prende(i)
\\ mangia
fc.posa(i)
```

- Questa soluzione è esente da deadlock ma potrebbe portare alla starvation

Esempio di esecuzione

- Supponiamo che `stato = pensa` per tutti i filosofi

0	Pensa	
1	Pensa	
2	Pensa	
3	Pensa	
4	Pensa	

Esempio di esecuzione

- F1 chiama il metodo `prende(1)`
- Lo stato di F1 passa ad `affamato`
- L'esecuzione del metodo invoca `verifica(1)`

0	Pensa	
1	Affamato	
2	Pensa	
3	Pensa	
4	Pensa	

```
        prende(int i) {
          stato[i] = affamato;
          verifica(i);
          if (stato[i] != mangia)
            auto[i].wait();
        }

verifica(int i) {
  if((stato[(i + 4) % 5] =! mangia) &&
     (stato[i] == affamato) &&
     (stato[(i + 1) % 5] != mangia)) {
       stato[i] = mangia;
       auto[i].signal();
  }
}
```

Esempio di esecuzione

- Lo stato di F1 passa a `mangia`

0	Pensa	
1	Mangia	
2	Pensa	
3	Pensa	
4	Pensa	

```
        prende(int i) {
          stato[i] = affamato;
          verifica(i);
          if (stato[i] != mangia)
            auto[i].wait();
        }

verifica(int i) {
  if((stato[(i + 4) % 5] =! mangia) &&
     (stato[i] == affamato) &&
     (stato[(i + 1) % 5] != mangia)) {
       stato[i] = mangia;
       auto[i].signal();
  }
}
```

Esempio di esecuzione

- F2 chiama il metodo `prende(2)`
- Lo stato di F2 passa ad `affamato`
- L'esecuzione del metodo invoca `verifica(2)`
- Siccome lo stato di F1 è `mangia` il metodo non cambia lo stato di F2, che quindi chiama `auto[2].wait()`

0	Pensa	
1	Mangia	
2	Affamato	wait
3	Pensa	
4	Pensa	

```
prende(int i) {
  stato[i] = affamato;
  verifica(i);
  if (stato[i] != mangia)
    auto[i].wait();
}

verifica(int i) {
  if((stato[(i + 4) % 5] =! mangia) &&
    (stato[i] == affamato) &&
    (stato[(i + 1) % 5] != mangia)) {
      stato[i] = mangia;
      auto[i].signal();
  }
}
```

Esempio di esecuzione

- F1 finisce di mangiare e chiama il metodo `posa(1)`
- Lo stato di F1 passa ad `pensa`
- L'esecuzione del metodo invoca `verifica(0)` e `verifica(2)`

0	Pensa	
1	Pensa	
2	Affamato	wait
3	Pensa	
4	Pensa	

```
posa(int i) {
  stato[i] = pensa;
  verifica((i + 4) % 5);
  verifica((i + 1) % 5);
}

verifica(int i) {
  if((stato[(i + 4) % 5] =! mangia) &&
    (stato[i] == affamato) &&
    (stato[(i + 1) % 5] != mangia)) {
      stato[i] = mangia;
      auto[i].signal();
  }
}
```

Esempio di esecuzione

- `verifica(0)` non fa nulla in quanto lo stato di F0 è diverso da `affamato`
- `verifica(2)` imposta lo stato di F2 a `mangia` e chiama `auto[2].signal()`
- F2 può quindi iniziare a mangiare dopo che si era messo in attesa

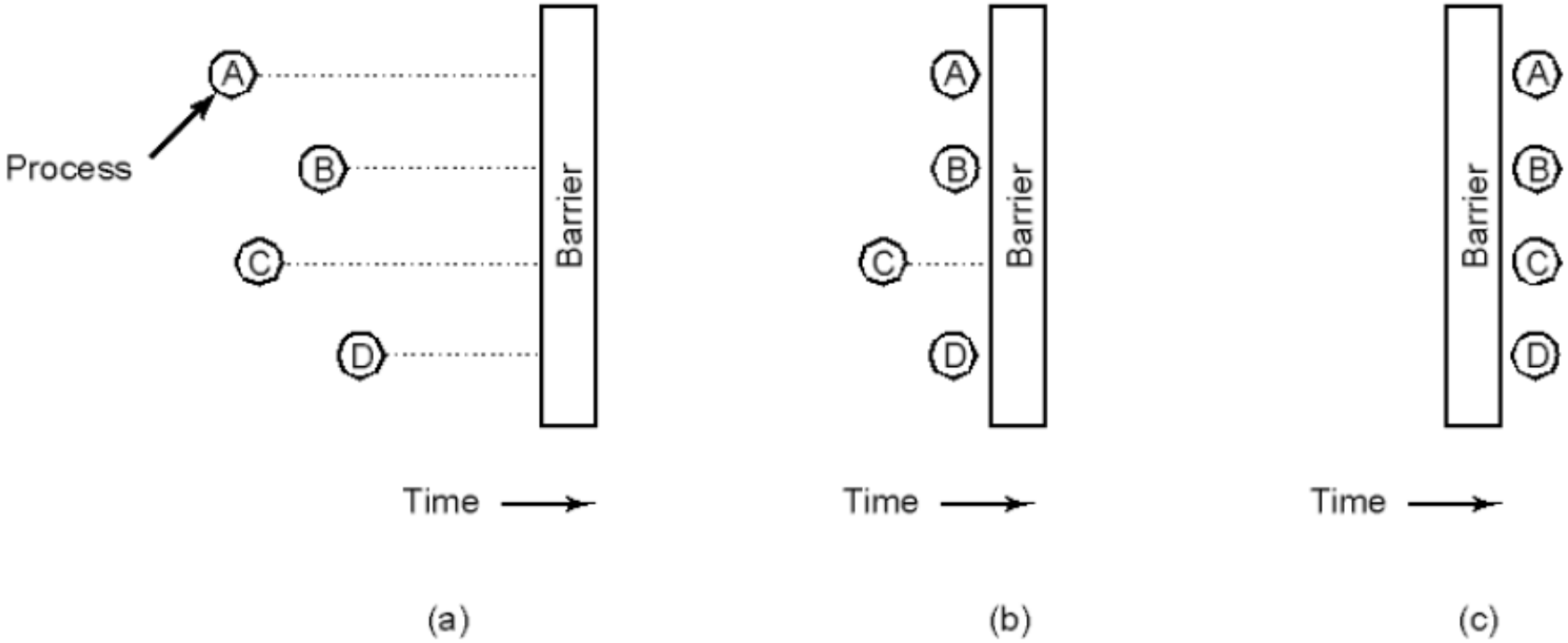
0	Pensa	
1	Pensa	
2	Mangia	
3	Pensa	
4	Pensa	

```
      posa(int i) {
        stato[i] = pensa;
        verifica((i + 4) % 5);
        verifica((i + 1) % 5);
      }

verifica(int i) {
  if((stato[(i + 4) % 5] =! mangia) &&
     (stato[i] == affamato) &&
     (stato[(i + 1) % 5] != mangia)) {
       stato[i] = mangia;
       auto[i].signal();
  }
}
```

Barriera

- Meccanismo di sincronizzazione che forza tutti i thread ad aspettare finché tutti non raggiungono un determinato punto
- I thread raggiungono la barriera ad istanti diversi ma la lasciano insieme



Esempi di sincronizzazione

- Solaris
- Windows XP
- Linux
- Pthreads
- La sincronizzazione in Java
  (vedi lezioni di laboratorio)

La sincronizzazione in Solaris (1)

- Implementa una varietà di lock per supportare multitasking, multithreading (compresi i thread real-time), e multiprocessing

- Usa gli adaptive mutex (semaforo binario *adattivo)*
  - parte come spin lock (attesa attiva) ma può passare al funzionamento con sospensione dei thread in coda di attesa
    - Usa spin lock se il dati sono bloccati da un processo che è in esecuzione
    - Sospende il processo se il dati sono bloccati da un processo che non è in esecuzione
  - usato per proteggere solo dati che vengono richiesti da sezioni di codice brevi (< 100 instruzioni)
    - Altrimenti l'uso dei cicli di attesa è inefficiente

La sincronizzazione in Solaris (2)

- Usa variabili "condition" e semafori per lunghe sezioni di codice
- Usa lock di lettura-scrittura per dati a cui si accede spesso e di solito per sola lettura
  - Solo per sezioni critiche lunghe

- Usa i turnstile (code di attesa del lock) per ordinare la lista dei thread in attesa di ottenere un adaptive mutex o un lock di lettura-scrittura
  - Anziché associare i turnstile agli oggetti condivisi, il SO assegna un turnstile ad ogni thread. Il turnstile del primo thread che si blocca su un oggetto condiviso diventa il turnstile per tutti gli altri thread che competono per lo stesso oggetto condiviso.

Sincronizzazione in Windows XP

- Maschera gli interrupt temporaneamente per proteggere l'accesso alle risorse globali sui sistemi ad un singolo processore
- Usa gli spinlock sui sistemi multiprocessore
- Fornisce anche gli oggetti dispatcher che offrono molti meccanismi differenti, fra cui mutex e semafori
- Gli oggetti dispatcher possono anche fornire eventi
  - Un evento agisce in modo molto simile alle variabili condizione, segnala quindi il verificarsi di una determinata condizione ad una serie di processi in ascolto

Sincronizzazione in Linux

- Per implementare sezioni critiche corte:
  - **enable_irq()/disable_irq()** per abilitare/disabilitare in modo temporaneo gli interrupt
    - Macchine mono-processore
  - **preempt_enable()/preempt_disable()** per abilitare/disabilitare la prelazione a livello kernel (dalla versione 2.6)
    - Macchine mono-processore
  - **Spin lock** (attesa attiva)
    - Macchine multi-processore

- Semafori per implementare le sezioni critiche lunghe

Sincronizzazione in Posix Pthread

- Le API Pthread non è implementata in uno specifico kernel ma è *OS-independent*

- Pthread fornisce:
  - mutex lock (fondamentale)
  - Variabili "condition"
  - read-write lock

- Le estensioni non portabili includono:
  - spin lock
  - semafori (estenzione POSIX SEM)

Equivalenza delle tecniche

- Tutte le tecniche per la comunicazione/sincronizzazione tra processi/thread presentate:
  - send/receive (nella IPC)
  - semafori
  - monitor
  - e (numerose) altre ancora
- Sono equivalenti:
  - ogni problema che si può risolvere con una tecnica si può risolvere con le altre
  - ciascuna tecnica si può implementare con una qualsiasi delle altre

- Un ultima tecnica è quella delle **transazioni atomiche**

Definizione di transazione

- Un insieme di istruzioni che eseguono un'unica funzione logica
  - **Esempio:**

```
read
manipolazione dei dati
read
manipolazione dei dati
write
manipolazione dei dati
read
manipolazione dei dati
Write
read
manipolazione dei dati
write
commit o abort
```

Si vuole garantire atomicità anche in presenza di guasti o malfunzionamenti del sistema

Atomicità della transazione

- L'effetto della transazione sulle informazioni memorizzate deve essere permanente **solo se tutte le operazioni sono state completate**
  - correttamente e senza interferenze da parte di altri processi (thread)

- La sequenza di operazioni di una transazione deve essere **atomica** come un'unica operazione indivisibile

- Terminazione:

  corretta → commit effetti permanenti

  errata → abort nessun effetto (roll back)

Tipologie di archivi (1)

- Obiettivo: assicurare l'atomicità delle transazioni
  - in modo che una terminazione anomala non produca alcun effetto sulla rappresentazione in memoria dei dati che la transazione ha modificato
- Per definire come il sistema possa garantire l'atomicità è necessario identificare le proprietà dei dispositivi usati per memorizzare i dati

- **Archivio volatile**

  le informazioni non sopravvivono allo spegnimento del sistema
  - memoria cache
  - memoria centrale

Tipologie di archivi (2)

- **Archivio non volatile**

  le informazioni sopravvivono allo spegnimento del sistema

  - dischi magnetici e ottici
  - nastri magnetici

- **Archivio stabile**

  le informazioni non vengono mai perse

  - Ideale, ma inesistente
  - Lo si realizza nella pratica tramite replicazione in molti archivi non volatili

- Ci concentriamo su come è possibile assicurare l'atomicità di una transazione in un ambiente in cui i guasti comportano la perdita di informazione memorizzata in **archivi volatili**

Transazioni atomiche individuali

- Per la gestione delle transazioni atomiche e il ripristino di transazioni non riuscite si usano tipicamente due tecniche:

- **Recupero da log** (registro)
  - Write-ahead logging (registrazione con scrittura anticipata)

- **Check pointing** (punti di verifica)

Write-ahead logging

- Log (registro) delle transazioni in un archivio stabile:
  - registra in un archivio stabile le transazioni e il loro stato di esecuzione

- Ciascun record del log descrive una singola operazione della transazione
  - nome della transazione
  - nome dell'oggetto dei dati
  - vecchio valore dei dati
  - nuovo valore dei dati

- Meccanismo di write-ahead logging
  - Inizio transazione → $<T_i$ starts$>$
  - Prima di ogni operazione della transazione → $<T_i$ operation$>$
  - Fine transazione → $<T_i$ commits$>$

Recupero basato sul log (1)

- L'algoritmo di recupero da fallimento è basato su due procedure:
  - `undo(`T_i`)`: riporta i dati modificati dalla transazione T_i ai vecchi valori
  - `redo(`T_i`)`: assegna ai dati modificati dalla transizione T_i il nuovo valore

Funzioni idempotenti (più esecuzioni dell'operazione devono avere lo stesso risultato di una sola esecuzione)

Recupero basato sul log (2)

L'algoritmo di recupero basato sui log:

- Ripristina i dati usando l'operazione `undo(`T_i`)` se una transizione T_i abortisce

- Se invece accade un fallimento del sistema di elaborazione
  - Per ogni transazione del log,
    - se il log contiene `<`T_i `starts>` ma non `<`T_i `commits>`, esegue `undo(`T_i`)`
    - se il log contiene sia `<`T_i `starts>` sia `<`T_i `commits>`, esegue `redo(`T_i`)`

Check Pointing

- Problema del logging:
  - Tempo di ripristino lungo in caso digrandi log
- Soluzione:
  - Check pointing (punti di verifica)
- Periodicamente si eseguono:
  - scrittura su archivio stabile dei record del log memorizzati su archivio volatile
  - scrittura dei dati modificati su archivio stabile
  - scrittura del record `<checkpoint>` sul log residente nell'archivio stabile

Ripristino basato su check pointing

- Esecuzione del ripristino
  - Si scandisce il log a ritroso fino al primo `<checkpoint>`
  - Per ogni transazione del log a partire dal checkpoint più recente,
    - se il log contiene $\langle T_i \text{ starts} \rangle$ ma non $\langle T_i \text{ commits} \rangle$, esegue $\text{undo}(T_i)$
    - se il log contiene sia $\langle T_i \text{ starts} \rangle$ sia $\langle T_i \text{ commits} \rangle$, esegue $\text{redo}(T_i)$

Transazioni atomiche concorrenti

- Le transazioni atomiche in realtà possono essere eseguite in modo concorrente

- Proprietà della **serializzabilità**:
  - Il risultato dell'esecuzione delle transazioni deve essere equivalente a quello che si otterrebbe
  - Eseguendo le transazioni in modo seriale (cioè sequenziale) in un ordine arbitrario

Tecniche per garantire la serializzabilità

- Scambio di operazioni non in conflitto
- Protocolli di lock
- Protocolli basati su marche di tempo

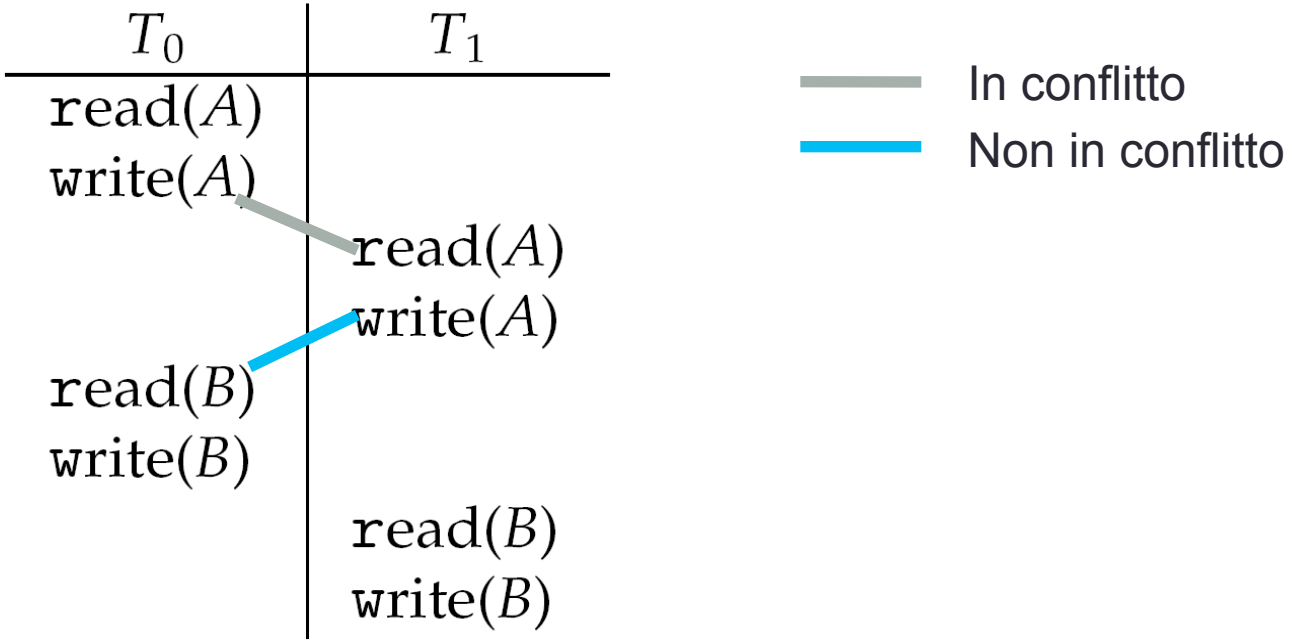
Schedulazione concorrente seriale

- Esecuzione atomica di T0 e T1: T0 è seguita da T1
- È detta **sequenza d'esecuzione seriale**
  - Tutte le istruzioni appartenenti alla stessa transazione sono raggruppate

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedulazione concorrente serializzabile (si consentono le sovrapposizioni)

- Si può generare una schedulazione serializzabile applicando una serie di scambi tra operazioni di T0 e T1 che non sono in conflitto
  - Op_i e Op_j sono in conflitto se accedono agli stessi dati e almeno una lo fa in scrittura
- In questo modo otteniamo una schedulazione seriale identica.



Dimostrazioni (1)

- Applicando le seguenti trasformazioni alla sequenza di sinistra si ottiene quella di destra, ciò dimostra che la sequenza di sinistra equivale ad una sequenza seriale

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Dimostrazioni (2)

- Scambio read(B) di T_0 con write(A) di T_1

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_0	T_1
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Dimostrazioni (3)

- Scambio read(B) di T_0 con read(A) di T_1

T_0	T_1
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

T_0	T_1
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

Dimostrazioni (4)

- Scambio write(B) di T_0 con write(A) di T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

T_0	T_1
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

Dimostrazioni (5)

- Scambio write(B) di T_0 con read(A) di T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Protocollo di lock di base

- T_i esegue il lock su dato Q nella modalità appropriata
- **Se lock disponibile**, T_i accede al dato
- **Se lock non disponibile**,
  - Se il lock richiesto è esclusivo (scrittura), T_i attende finché il dato viene rilasciato
  - Se il lock richiesto è condiviso (lettura),
    - **T_i accede** al dato **se** esso è correntemente **bloccato con lock condiviso**
    - **T_i attende se** il dato è correntemente **bloccato con lock esclusivo**
- **Serializzabilità non garantita**

Protocollo di lock a due fasi

- Fase di crescita (**growing phase**)
  - una transazione può ottenere dei lock, ma non li può rilasciare
- Fase di contrazione (**shrinking phase**)
  - una transazione può rilasciare i lock, ma non ne può ottenere di nuovi
- Assicura la serializzabilità
- Non previene gli stalli

Timestamp (marca di tempo)

- Un Timestamp **TS(T_i)**
  - è un attributo univoco
  - che rappresenta quando la transazione T_i è "entrata nel sistema"

- Generazione del timestamp
  - Clock di sistema
  - Contatore logico
    - incrementato ad ogni entrata di transazione nel sistema

Serializzazione nei protocolli di timestamp

- Si sceglie in anticipo un ordinamento tra le transazioni in base alle marche di tempo (timestamp)
- Ad ogni transazione si associa un timestamp TS
  - $TS(T_i) < TS(T_j)$ se T_i entra nel sistema prima di T_j

- Le marche di tempo delle transazioni determinano l'ordine di serializzazione
  - Se $TS(T_i) < TS(T_j)$,
  - allora la schedulazione prodotta deve essere equivalente alla schedulazione serializzata in cui la transazione T_i appare prima della transazione T_j

Protocollo basato su timestamp (1)

- Due tipi di timestamp per un dato Q
  - W-timestamp(Q) denota la marca di tempo più elevata di ogni transazione che ha eseguito write(Q) con successo
  - R-timestamp(Q) denota la marca di tempo più elevata di ogni transazione che ha eseguito read(Q) con successo

- Idea: ogni operazione read o write in conflitto è eseguita nell'ordine della marca di tempo

Protocollo basato su timestamp (2)

- **read(Q)**
  - Se $TS(T_i) < W\text{-timestamp}(Q)$ (Q è già stato sovrascritto),
    la lettura è negata e T_i esegue roll-back (riceve nuovo timestamp e viene riavviata)
  - Se $TS(Ti) \geq W\text{-timestamp}(Q)$,
    la lettura è eseguita e
    $R\text{-timestamp}(Q) = \max\{R\text{-timestamp}(Q), TS(T_i)\}$

- **write(Q)**
  - Se $TS(T_i) < R\text{-timestamp}(Q)$ (valore di Q necessario in precedenza),
    la scrittura è negata e T_i esegue roll-back
  - Se $TS(T_i) < W\text{-timestamp}(Q)$ (T_i cerca di scrivere un valore obsoleto di Q),
    la scrittura è negata e T_i esegue roll-back
  - Altrimenti, la scrittura è eseguita e

    $W\text{-timestamp}(Q) = \max\{W\text{-timestamp}(Q), TS(T_i)\}$

Schedulazione possibile secondo il protocollo con marche di tempo

- $TS(T_2) < TS(T_3)$

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

R-timestamp(B) = $TS(T_2)$

R-timestamp(B) = $TS(T_3)$

Schedulazione possibile secondo il protocollo con marche di tempo (1)

- $TS(T_2) < TS(T_3)$

T_2	T_3

Timestamp	Valore
R-timestamp(A)	0
W-timestamp(A)	0
R-timestamp(B)	0
W-timestamp(B)	0

Schedulazione possibile secondo il protocollo con marche di tempo (2)

- $TS(T_2) < TS(T_3)$
- $TS(T_2) >$ R-timestamp(B)

T_2	T_3
read(B)	

Timestamp	Valore
R-timestamp(A)	0
W-timestamp(A)	0
R-timestamp(B)	$TS(T_2)$
W-timestamp(B)	0

Schedulazione possibile secondo il protocollo con marche di tempo (3)

- $TS(T_2) < TS(T_3)$
- $TS(T_3) > R\text{-}timestamp(B)$

T_2	T_3
read(B)	
	read(B)

Timestamp	Valore
R-timestamp(A)	0
W-timestamp(A)	0
R-timestamp(B)	$TS(T_3)$
W-timestamp(B)	0

Schedulazione possibile secondo il protocollo con marche di tempo (4)

- $TS(T_2) < TS(T_3)$
- $TS(T_3) >$ R-timestamp(B) && $TS(T_3) >$ W-timestamp(B)

T_2	T_3
read(B)	
	read(B)
	write(B)

Timestamp	Valore
R-timestamp(A)	0
W-timestamp(A)	0
R-timestamp(B)	$TS(T_3)$
W-timestamp(B)	$TS(T_3)$

Schedulazione possibile secondo il protocollo con marche di tempo (5)

- $TS(T_2) < TS(T_3)$
- $TS(T_2) >$ R-timestamp(A)

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	

Timestamp	Valore
R-timestamp(A)	$TS(T_2)$
W-timestamp(A)	0
R-timestamp(B)	$TS(T_3)$
W-timestamp(B)	$TS(T_3)$

Schedulazione possibile secondo il protocollo con marche di tempo (6)

- $TS(T_2) < TS(T_3)$
- $TS(T_3) > R\text{-}timestamp(A)$

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)

Timestamp	Valore
R-timestamp(A)	$TS(T_3)$
W-timestamp(A)	0
R-timestamp(B)	$TS(T_3)$
W-timestamp(B)	$TS(T_3)$

Schedulazione possibile secondo il protocollo con marche di tempo (6)

- $TS(T_2) < TS(T_3)$
- $TS(T_3) >$ R-timestamp(A) && $TS(T_3) >$ W-timestamp(A)

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Timestamp	Valore
R-timestamp(A)	$TS(T_3)$
W-timestamp(A)	$TS(T_3)$
R-timestamp(B)	$TS(T_3)$
W-timestamp(B)	$TS(T_3)$