# Sincronizzazione tra processi

## Sincronizzazione tra processi

- Background
- Il problema della sezione critica
- Soluzione di Peterson
- Hardware per la sincronizzazione
- Semafori
- Problemi classici di sincronizzazione
- Monitor
- Esempi di sincronizzazione
- Transazioni atomiche

## Background

- L'accesso concorrente a dati condivisi può portare all'inconsistenza dei dati
- Sono pertanto necessari meccanismi che assicurino un'esecuzione ordinata dei processi cooperanti
- Riconsideriamo il problema produttore-consumatore con buffer limitato

## Produttore – Consumatore con buffer limitato

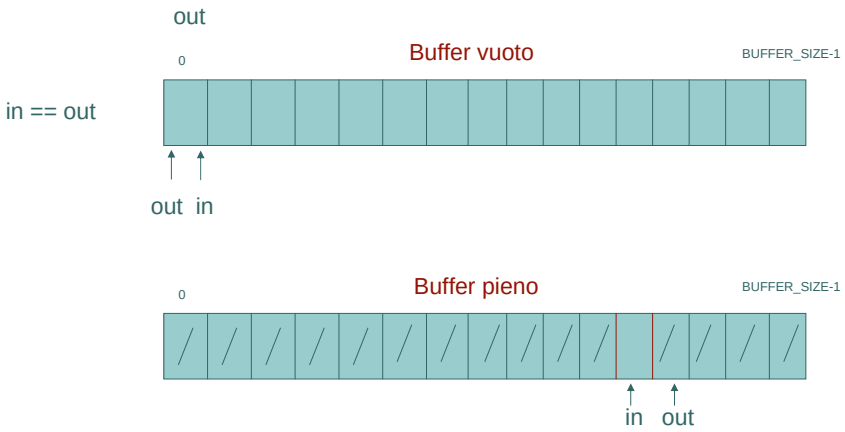
```
item nextProduced;

while (true) {
   /* Produce an item */
   while ((in + 1) % BUFFER_SIZE == out)
        ; // do nothing
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER SIZE;
 }
```

```
item nextConsumed

while (true) {
        while (in == out)
             ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        // consume the item
    }
```

# Produttore – Consumatore con buffer limitato




(in+1) mod BUFFER_SIZE == out

# Produttore – Consumatore con buffer limitato

- Supponiamo di voler fornire una soluzione al problema produttore-consumatore che riempe tutti i buffer.
- Possiamo farlo usando una variabile intera count che tiene traccia del numero di buffer pieni. Inizialmente, count vale 0. Essa viene poi

  - incrementata dal produttore

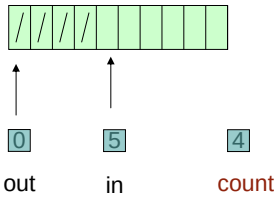
  - decrementata dal consumatore

# Produttore – Consumatore con buffer limitato

- **Dati condivisi**

```
#define BUFFER_SIZE 10
Typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count =0;
```




# Produttore

```
while (true) {

        /*  produce an item and put in nextProduced  */

         while (count == BUFFER_SIZE)
                  ; // do nothing
              buffer [in] = nextProduced;
              in = (in + 1) % BUFFER_SIZE;
              count++;
}
```

## Consumatore

```
while (true)  {
        while (count == 0)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                 count--;

                    /*  consume the item in nextConsumed
 }
```

## Corsa critica (race condition)

- count++ può essere implementata come
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- può essere implementata come
  register2 = count
  register2 = register2 - 1
  count = register2
- Si consideri quest'ordine di esecuzione, con "count = 5" all'inizio:
  $T_0$: produttore esegue register1 = count {register1 = 5}
  $T_1$: produttore esegue register1 = register1 + 1 {register1 = 6}
  $T_2$: consumatore esegue register2 = count {register2 = 5}
  $T_3$: consumatore esegue register2 = register2 - 1 {register2 = 4}
  $T_4$: produttore esegue count = register1 {count = 6 }
  $T_5$: consumatore esegue count = register2 {count = 4}

## Il problema della sezione critica

**n processi $P_1$ ... $P_n$**

**Ciascuno ha un segmento di codice, chiamato sezione critica, in cui può modificare variabili condivise.**

**Occorre progettare un protocollo che i processi possono utilizzare per cooperare**

```
do {
        entry section
                critical section
        exit section
                remainder section

} while (TRUE);
```

## Il problema della sezione critica

- 1. **Mutua esclusione** – se il processo $P_i$ sta eseguendo la sua sezione critica, *nessun altro* processo può eseguire la propria sezione critica.
- 2. **Progresso** – se nessun processo sta eseguendo la propria sezione critica e qualche processo vuole entrare nella sua, la selezione del prossimo processo che entrerà non può essere ritardata indefinitamente. Inoltre, solo i processi che **non stanno** eseguendo la loro *remainder section* possono partecipare alla decisione su chi sarà il prossimo a entrare nella propria sezione critica.
- 3. **Attesa limitata** – esiste *un limite* al numero di volte in cui gli altri processi possono entrare nelle loro sezioni critiche dopo che un processo ha fatto richiesta di entrare nella propria e prima che tale richiesta sia esaudita.
  - I processi sono in esecuzione a velocità non nulla.
  - Non facciamo alcuna ipotesi sulle velocità *relative* degli *n* processi.

## Codice del Kernel

- Il codice che implementa il kernel di un SO è soggetto a molte possibili corse critiche (e.g., modifica tabelle dati globali)
- Kernel: Preemptive e non preemptive
- Monoprocessori – possono *disabilitare gli interrupt*.
  - La sequenza di istruzioni corrente sarà eseguita in ordine senza interruzioni.
  - Di solito inefficiente sui sistemi multiprocessore.
    - I sistemi operativi che la usano sono carenti in scalabilità.

## Soluzione di Peterson

- Soluzione per due processi
- Assumiamo che le istruzioni LOAD e STORE siano atomiche; cioè, non possano essere interrotte.
- I due processi condividono due variabili:
  - int turn;
  - boolean flag[2]
- La variabile turn indica di chi è il turno di entrata nelle sezione critica.
- L'array flag viene usato per indicare se un processo è pronto ad entrare nella sezione critica - flag[i] = true significa che il processo $P_i$ è pronto!

## Algoritmo per il processo $P_i$

```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

              critical section

        flag[i] = FALSE;

              remainder section

 }
```

Attesa attiva

Garantisce:

- mutua esclusione
- progresso
- attesa limitata

## Hardware per la sincronizzazione

- I calcolatori moderni forniscono *istruzioni hardware speciali che funzionano in modo* atomico.
  - Atomico = non-interrompibile.
  - Controllare e modificare il contenuto di una parola.
  - Scambiare il contenuto di due parole.

## TestAndSet

- **Definizione**:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

## Mutua esclusione con TestAndSet

- Variabile booleana condivisa lock, inizializzata a FALSE.
- Soluzione:

Attesa attiva

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing

                //    critical section

        lock = FALSE;

                //    remainder section

}
```

## Swap

- **Definizione**:

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

## Mutua esclusione con Swap

- Variabile booleana condivisa lock, inizializzata a FALSE; Ciascun processo ha una variabile booleana locale key.
- Soluzione:

Attesa attiva

```
while (true)  {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                  //    critical section

        lock = FALSE;

                  //      remainder section
}
```

# Protocollo con TestAndSet

boolean waiting[n];
boolean lock

Inizializzate a FALSE

Garantisce:

- mutua esclusione
- progresso
- attesa limitata

```
do {
  waiting[i] = TRUE;
  key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
  waiting[i] = FALSE;

            // Critical Section

  j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;
  if (j == i)  lock= FALSE
      else waiting [j] = FALSE;

            // Remainder Section

} while(TRUE);
```

# Semafori

- Strumento di sincronizzazione
- Semaforo *S* – variabile intera
- Due operazioni *atomiche* modificano S: wait() e signal()
  - Originariamente chiamate P() (proberen) e V() (verhogen)

```
wait (S) {
      while S <= 0
          ; // no-op
        S--;
  }
```

attesa attiva

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

# Uso dei semafori

- **Semaforo binario (mutex lock)** – il valore può essere 0 o 1;
  - **Mutua esclusione**.

```
semaforo S; // inizializzato a 1

wait(S);
 // Critical section;
signal(S);
```

  - **Sincronizzazione: semaforo *synch*** // inizializzato a 0

```
S1;                          wait(synch);
signal (synch);              S2;
```

- **Semaforo contatore** – il valore può variare in un dominio senza restrizioni.
  Utile in applicazioni in cui ci sono **N** risorse disponibili

# Semaforo senza attesa attiva

- Ad ogni semaforo è associata una coda d'attesa.

```
typedef struct {
    int value;          // valore (di tipo intero)
   struct process *list; // puntatore el. succ. coda
  } semaphore;
```

- Sono necessarie due operazioni (fornite dal SO):
  - block() – sospende il processo che invoca l'operazione
  - wakeup(P) – sveglia il processo P.

## Semaforo senza attesa attiva

- Implementazione di **wait:**

```
wait (semaphore *S){
      S→value--;
      if (S→value < 0) {
            aggiungi il processo a S→list;
             block();  }
 }
```

- Implementazione di **signal**:

```
signal (semaphore *S){
        S→value++;
         if (S→value <= 0) {
                rimuovi un processo P da S→list;
                 wakeup(P);  }
}
```

## Implementazione di un semaforo

- Deve garantire che due processi **non possano** eseguire wait () / signal () sullo stesso semaforo contemporaneamente.
- L'implementazione diviene il problema della sezione critica (i.e., il codice delle operazioni wait() e signal() è posto nella sezione critica)
- Monoprocessori – *disabilitazione degli interrupt*
- Possono essere usate le soluzioni software ( e.g., Peterson e generalizzazioni…)
- Possono essere usate le soluzioni che si basano su istruzioni atomiche - se disponibili - (e.g., Swap, TestAndSet,…)

## Implementazione di un semaforo

- C'è ancora attesa attiva nell'implementazione ma è limitata al codice delle operazioni wait() e signal()
  - Codice da proteggere è breve
  - Al contrario, le applicazioni possono spendere molto tempo nelle sezioni critiche.

## Deadlock e Starvation

- Deadlock (stallo) – due o più processi aspettano un evento che può essere causato solo da uno dei processi in attesa.
- Siano S e Q 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 indefinito) – Un processo può non essere mai rimosso dalla cosa del semaforo in cui è sospeso.

# Problemi classici di sincronizzazione

- Problema del buffer limitato.
- Problema dei lettori e degli scrittori.
- Problema dei filosofi a pranzo.

# Problema del buffer limitato

- *N* buffer, ciascuno contiene un item
- Semaforo mutex inizializzato a 1
- Semaforo full inizializzato a 0
- Semaforo empty inizializzato a N.

# Problema del buffer limitato

- La struttura del processo produttore

```
while (true)  {

  //   produce an item

    wait (empty);
    wait (mutex);

  //  add the item to the  buffer

    signal (mutex);
    signal (full);
}
```

- La struttura del processo consumatore

```
while (true) {

    wait (full);
    wait (mutex);

    //  remove an item from  buffer

    signal (mutex);
    signal (empty);

    //  consume the removed item
}
```

# Problema dei lettori e degli scrittori

- Un insieme di dati è condiviso tra un numero di processi concorrenti
  - Lettori – leggono soltanto i dati;
  - Scrittori – possono leggere e scrivere.
- Problema – permettere a più lettori di leggere contemporaneamente. Solo uno scrittore (e nessun altro) può accedere ai dati condivisi.
- Dati condivisi
  - Insieme di dati
  - Semaforo mutex inizializzato a 1.
  - Semaforo wrt inizializzato a 1.
  - Variabile intera readcount inizializzata a 0.

# Problema dei lettori e degli scrittori

- Struttura del processo scrittore

```
while (true) {
    wait (wrt) ;

    //    writing is performed

    signal (wrt) ;
}
```

- Struttura del processo lettore

```
while (true) {
    wait (mutex) ;
    readcount ++ ;
    if (readercount == 1)  wait (wrt) ;
    signal (mutex)

    // reading is performed

    wait (mutex) ;
    readcount  - - ;
    if (readcount  == 0)  signal (wrt) ;
    signal (mutex) ;
 }
```

# Problema dei filosofi a pranzo


- Dati condivisi
  - Ciotola di riso (insieme di dati)
  - Vettore di semafori chopstick [5] inizializzati a 1

# Problema dei filosofi a pranzo

- Struttura del filosofo *i-esimo*:

```
while (true)  {
        wait (chopstick[i]);
        wait (chopStick[ (i + 1) % 5]);

            //  eat

        signal (chopstick[i]);
        signal (chopstick[ (i + 1) % 5]);

            //  think
}
```

… può essere generato un deadlock!

# Problemi con i Semafori

- Uso corretto delle operazioni sul semaforo:
  - signal (mutex)  ….  wait (mutex)
  - wait (mutex)  …  wait (mutex)
  - Omissione di wait (mutex) o signal (mutex) o entrambe

# Monitor

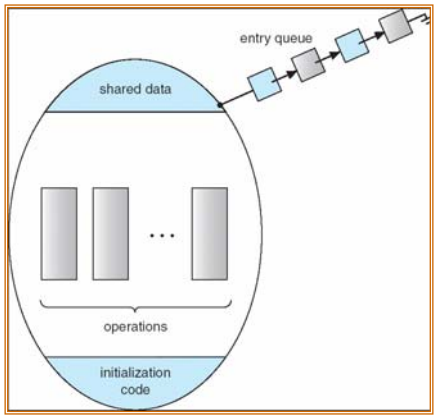
- **Un'astrazione ad alto livello** che fornisce un conveniente ed efficace meccanismo per la sincronizzazione tra processi
- Solo un processo alla volta può essere attivo all'interno di un monitor

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
            …

   procedure Pn (…) {……}

    Initialization code ( ….) { … }
            …
   }
}
```

# Visione schematica di un Monitor



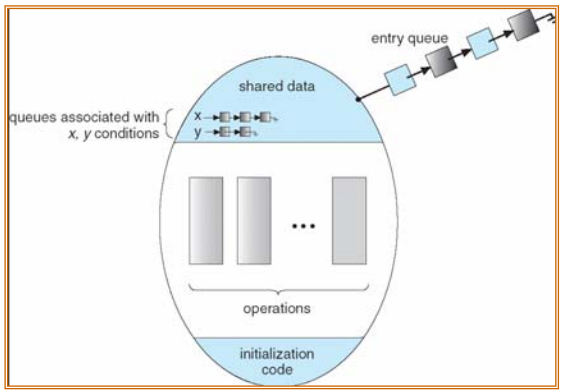

# Variabili condizione

- condition x, y;

- Due operazioni su una variabile condizione:
  - x.wait () – un processo che invoca l'operazione viene sospeso.
  - x.signal () – risveglia uno dei processi (se c'è) che ha precedentemente invocato x.wait ()

- Possibili implementazioni:
  - x.signal() e aspetta: P aspetta fino a che Q lasci il monitor
  - x.signal() e continua: Q aspetta che P lasci il monitor

# Monitor con variabili condizione



## Soluzione al problema dei filosofi a pranzo

```
monitor dp
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
          state[i] = THINKING;
               // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
      }
```

## Soluzione al problema dei filosofi a pranzo

```
        void test (int i) {
              if ( (state[(i + 4) % 5] != EATING) &&
              (state[i] == HUNGRY) &&
              (state[(i + 1) % 5] != EATING) ) {
                   state[i] = EATING ;
                    self[i].signal () ;
               }
         }

         initialization_code() {
              for (int i = 0; i < 5; i++)
              state[i] = THINKING;
         }
    }
```

## Soluzione al problema dei filosofi a pranzo

- Ciascun filosofo invoca le operazioni pickup() e putdown() come segue:

```
dp.pickup (i)

    EAT

dp.putdown (i)
```

## Implementazione di un monitor usando semafori

- Variabili

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
int next-count = 0;
```

- Ciascuna procedura ***F*** verrà sostituita da

```
wait(mutex);
   …
        corpo di F;

   …
if (next-count > 0)
   signal(next)
else
   signal(mutex);
```

- La mutua esclusione all'interno del monitor è garantita.

## Implementazione di un monitor usando semafori

- Per ciascuna variabile condizione ***x***, abbiamo:

```
semaphore x-sem; // (inizialmente  = 0)
int x-count = 0;
```

- L'operazione x.wait può essere implementata come segue:

```
x-count++;
if (next-count > 0)
     signal(next);
else
     signal(mutex);
wait(x-sem);
x-count--;
```

## Implementazione di un monitor usando semafori

- L'operazione x.signal può essere implementata come segue:

```
if (x-count > 0) {
     next-count++;
     signal(x-sem);
     wait(next);
     next-count--;
}
```

## Linguaggi per la programmazione concorrente

Diversi linguaggi di programmazione concorrente hanno incorporato il concetto di monitor (e.g., Concurrent Pascal, Mesa, C#, NeWs …)

Java fornisce un meccanismo per il controllo della concorrenza molto simile al concetto di monitor.

Ad ogni oggetto in Java è associato un **lock**. Quando un metodo è dichiarato **synchronized**, la chiamata richiede il possesso del lock.

```
public class SimpleClass{
     …
     public synchronized void safeMethod{
               /* implementazione del metodo */
          }
}
```

## Monitor in Java

Creiamo un'istanza dell'oggetto

```
SimpleClass sc = new SimpleClass();
```

Se un thread invoca il metodo sc.safeMethod() e il lock è già posseduto da un altro thread, il thread invocante si blocca e viene messo nell'**entry set** del lock dell'oggetto.

Il lock viene rilasciato da un thread quando l'esecuzione del metodo termina.

Java offre anche le operazioni **wait()** e **notify()** che sono molto simili alle operazioni wait() e signal() definite per le variabili condizione in un monitor

## Esempi di sincronizzazione

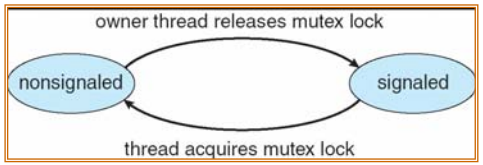
- Solaris.
- Windows XP.
- Linux.
- API Pthread.

## La sincronizzazione nel kernel di Solaris

- Implementa adaptive mutex, variabili condizione, semafori, lock di lettura e scrittura, e turnstile
- Adaptive mutex (semaforo binario adattivo) per proteggere dati acceduti da sezioni di codice brevi. Si comportano come spinlock su architetture SMP
- Le variabili ed i lock di lettura e scrittura sono usati quando ci sono lunghe sezioni di codice e molti lettori.
- Usa i turnstile per ordinare la lista dei thread in attesa di ottenere un adaptive mutex o un lock di lettura-scrittura.

## La sincronizzazione nel kernel di Windows XP

- Disabilita temporaneamente gli interrupt per proteggere l'accesso alle risorse globali sui sistemi a singolo processore
- Usa gli spinlock sui sistemi multiprocessore
- Fornisce dispatcher object: mutex, semafori, eventi e timer
- I dispatcher object possono essere in due stati: signaled e nonsignaled
- Gli eventi sono simili alle variabili condizione




## La sincronizzazione nel kernel Linux

- Linux singolo processore:
  - disabilita la possibilità di prelazione
  - preempt_disable(), preempt_enable, preempt_count
- Linux SMP :
  - spinlock
  - semafori

| single processor | multiple processors |
|---|---|
| Disable kernel preemption. | Acquire spin lock. |
| Enable kernel preemption. | Release spin lock. |

## L' API Pthread

- L' API Pthread è indipendente dal sistema operativo.
- Fornisce:
  - mutex lock
  - condizioni variabili
- Le estensioni non portabili comprendono:
  - lock di lettura-scrittura
  - semafori

Si veda la descrizione dell'API sul sito del corso – link *letture consigliate*

## Transazioni atomiche

- Singole unità logiche di lavoro eseguite **completamente** o **per niente**
- Relazione con le basi di dati
- Difficoltà: assicurare l'atomicità anche in presenza di guasti hardware
- Transazione – collezione di istruzioni o operazioni che realizza una singola funzionalità logica
  - Serie di read e write
  - Terminata da commit (transaction successful) o abort (transaction failed)
  - Una transazione abortita richiede il ripristino dell' informazione pre-esistente

## Dispositivi di memorizzazione

- Memoria volatile
  - Esempio: memoria centrale, cache
- Memoria non volatile
  - Esempio: disco e nastro
- Memoria stabile – L' Informazione non viene persa mai
  - Nella realtà non esiste, ma viene approssimata attraverso organizzazioni dei dischi RAID, aventi modalità di guasto indipendenti

L'obiettivo è assicurare l'atomicità delle transazioni laddove guasti causano perdita di informazione

## Recupero basato su log

- Memorizza in memoria stabile informazione circa tutte le modifiche effettuate da una transazione
- Molto comune è lo schema *write-ahead logging*
  - Ogni record del file di log in memoria stabile descrive la singola write e include
    - Nome Transazione
    - Nome del dato
    - Vecchio valore
    - Nuovo valore
  - <$T_i$ starts> scritto nel log quando la transazione $T_i$ inizia
  - <$T_i$ commits> scritto quando $T_i$ è completa (nel file di log)
- I record del file di log devono risiede su memoria stabile prima che le operazioni reali della transazione abbiano luogo

## Algoritmo basato su log

- Usando il log, il sistema può gestire ogni errore nella memoria volatile
  - Undo($T_i$) ripristina i valori modificati da $T_i$
  - Redo($T_i$) aggiorna i valori in accordo alla transazione $T_i$
- Undo($T_i$) e redo($T_i$) devono essere idempotenti
  - Esecuzioni multiple hanno lostesso effetto di una singola esecuzione
- Se il sistema fallisce, procede come segue
  - Se il log contiene <$T_i$ starts> senza <$T_i$ commits>, undo($T_i$)
  - Se contiene <$T_i$ starts> e <$T_i$ commits>, redo($T_i$)

## Transazioni concorrenti

- L'esecuzione deve essere equivalente ad un'esecuzione seriale – **serializzabilità**
- Tutte le transazioni potrebbero essere effettuate in sezioni critiche
  - inefficiente, troppo restrittivo
- Esistono algoritmi efficienti per il controllo della concorrenza che implementano la serializzabilità

## Protocollo con variabile turn

$P_i$ e $P_j$ condividono la variabile ***turn*** (inizializzata a i o j).

Codice per $P_i$

```
do{
while (turn != i)  no-op;
    sezione critica
turn = j;
   sezione non critica
}while(TRUE);
```

## Protocollo con variabili flag

$P_i$ e $P_j$ condividono due variabili booleane ***flag***[i] e ***flag***[j] inizializzate a FALSE.

Codice per $P_i$

```
do{
flag[i]=TRUE;
while (flag[j] ==TRUE)  no-op;
    sezione critica
flag[i]=FALSE;
   sezione non critica
}while(TRUE);
```