Fondamenti dei Sistemi Operativi

Sincronizzazione

Sommario

- Inconsistenza di dati condivisi: i rischi dell'*interleaving* nei processi concorrenti
- La *race condition*
- Il problema della "sezione critica" dei processi concorrenti e la soluzione
- Algoritmi tentativi per la soluzione del problema della "sezione critica"
- Istruzioni di macchina per la sincronizzazione: *Test&Set* oppure *Swap*
- Il semaforo binario ed il semaforo contatore: variabile semaforica e operatori
- Rischi della mutua esclusione: *deadlock* e *starvation*
- Monitor: astrazione, procedure pubbliche, variabili condizioni e sincronizzazione

Shared data inconsistency

- 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 (o thread) cooperanti che condividano uno spazio di indirizzi (codice e dati).
- Es.: in un sistema multithread, più thread cooperano mentre sono eseguiti in modo asincrono e condividono dati.
  - Possiamo esemplificare un paradigma di cooperazione come nel modello del produttore/consumatore.

Shared data concurrent access risks 1/2

Shared data

```
#define BUFFER_SIZE 10
typedef struct { . . . } item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

processo ***Producer***

```
item nextProduced;
{
   while (counter == BUFFER_SIZE); /* do nothing: none free buffer */
   counter++;                      /* add the next product to the buffer */
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
}
```

processo ***Consumer***

```
Item nextConsumed;
{
   while (counter == 0); /* do nothing: none available buffer */
   counter--;
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
}
```

Shared data concurrent access risks 2/2

Gli statement **counter++ e counter--** devono essere eseguiti *atomicamente*.

- Un'operazione è eseguita atomicamente se viene completata interamente senza interruzioni.

Lo statement "**counter ++**" è eseguito, in linguaggio di macchina di un processore, tramite le istruzioni:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

Lo statement "**counter --**" è eseguito, in linguaggio di macchina di un processore, tramite le istruzioni:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Se sia il "produttore" che il "consumatore" cercano contemporaneamente di aggiornare il buffer, le istruzioni di macchina possono essere intercalate.

L'intercalare dipende da come vengono schedulati il "produttore" e il "consumatore".

Interleaving effect

Si consideri il seguente ordine di esecuzione e si supponga che sia counter = 5:

producer :	`register1 = counter`	(register1 = 5)
producer :	`register1 = register1 + 1`	(register1 = 6)
consumer :	`register2 = counter`	(register2 = 5)
consumer :	`register2 = register2 - 1`	(register2 = 4)
producer :	`counter = register1`	(counter = 6)
consumer :	`counter = register2`	(counter = 4)

Il risultato dell'esecuzione non è deterministico, ma dipende dall'ordine con cui si è dato accesso ai dati (*race condition*).

Infatti il valore di **counter** potrà essere 4 oppure 6, laddove il risultato corretto dovrebbe essere 5.

The "race" condition

Race condition: è questa la situazione che si determina quando più processi (o thread) **accedono e manipolano concorrentemente dati condivisi (*shared data*)**.

Il valore finale dei dati condivisi dipende da quale processo (o thread) termina per ultimo.

- Frequente nei sistemi operativi multitasking, sia per dati in *user space* sia per strutture del kernel.
- Estremamente pericolosa: porta al malfunzionamento dei processi (o thread) cooperanti, o anche (nel caso delle strutture in *kernel space*) dell'intero sistema.
- Difficile da individuare e riprodurre: dipende da informazioni astratte dai processi o thread (decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, . . .)

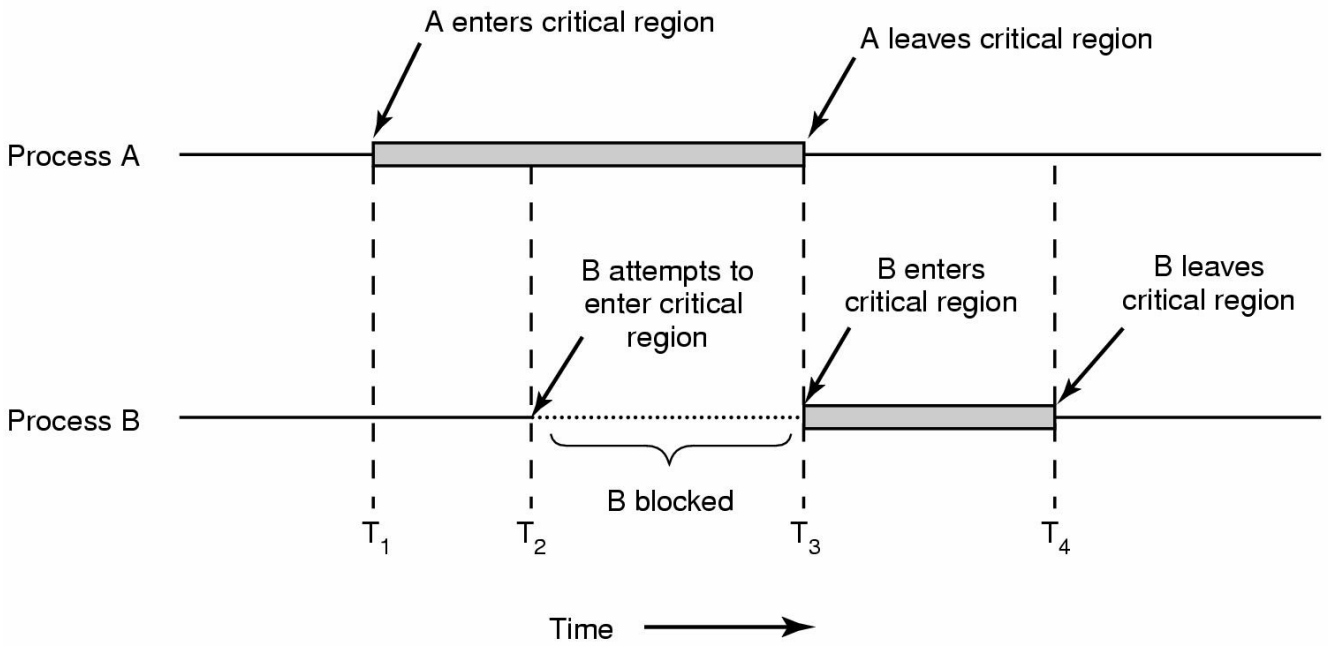
- Per prevenire le *race condition*, i processi (o thread) concorrenti devono essere **sincronizzati**.

Nel seguito si assumerà che quanto asserito per i processi valga identicamente se si considerano i thread.

The Critical-Section problem

- Se *n* processi **competono tutti per l'uso** di un qualche dato condiviso e . . . .
- . . . . se ogni processo ha un segmento di codice segment, detto ***critical section*** (sezione critica), in cui si fa accesso al dato condiviso:
- **Problema** – per evitare una race condition, è necessario garantire:

1) ***Mutua esclusione*** e cioè che: quando un processo è in esecuzione nella sua sezione critica, a nessun altro processo concorrente sia consentito di accedere ad eseguire la propria sezione critica;



2) ***Progresso*** e cioè che: se nessun processo sta eseguendo la sua sezione critica e alcuni processi vogliono entrare nella loro, allora solo i processi che sono in attesa di far uso della loro sezione critica possono partecipare alla decisione su chi sarà il prossimo a entrare nella propria sezione critica, e questa scelta non può essere ritardata indefinitamente;

3) ***Attesa limitata*** *(bounded waiting)* e cioè che: deve esistere 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.

- Questa condizione garantisce la prevenzione della starvation di un singolo processo.
- Si assume che ciascun processo sia in esecuzione a velocità non nulla.
- Non si può fare alcuna ipotesi sulle velocità *relative* degli *n* processi.

Attempts to solve the Critical-Section problem

Per semplicità si assume che:

- vi siano solo 2 processi, P_i and P_j

- la struttura generale del processo P_i (e dell'altro processo P_j) siano analoghe

- i processi possono condividere qualche variabile comune per sincronizzare le proprie azioni.

Algorithm 1

Shared variables:

`int turn;` inizialmente `turn = 0`

`turn = i` ⇒ P_i può entrare nella sua sezione critica

Processo P_i

```
do      {
        while (turn != i);  no-op  /* entry section */
        sezione critica
        turn = j;                  /* exit section */
        sezione rimanente (non critica)
        } while (1);
```

- L'algoritmo soddisfa la mutua esclusione, ma non il requisito di "progresso", poichè richiede una stretta alternanza dei 2 processi nell'uso della sezione critica. Infatti, se `turn == 0` e P_j è pronto ad entrare nella sua sezione critica, non può farlo, anche se P_i è nella sua sezione non critica.

  → **inadatto per processi con differenze di velocità**

- È un esempio di **busy wait**: attesa attiva di un evento.
  - semplice da implementare
  - può portare a consumi inaccettabili di cpu
  - in genere, da evitare, ma a volte è preferibile (es. in caso di attese molto brevi)

Algorithm 2

Shared variables:

`boolean flag[2];` inizialmente **flag [0] = flag [1] = false.**

flag [i] = true $\Rightarrow$ P_i pronto per entrare nella sua sezione critica

Process P_i

```
do      {
        flag[i] := true;        /* entry section */
        while (flag[j]);
        sezione critica
        flag [i] = false;       /* exit section */
        sezione rimanente (non critica)
        } while (1);
```

- L'algoritmo è sbagliato.
  Infatti, l'intercalare delle *entry section* porta ad un loop infinito (*deadlock*) per entrambi i processi. Invertire le istruzioni della entry section porterebbe solo alla violazione della mutua esclusione.

Bakery algorithm

Critical section for n processes

- Quello che segue è l'unico algoritmo che risolve via software il problema di evitare una race condition.
  - Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
  - If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
  - The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- L'algoritmo "del fornaio" è, però, complesso da gestire.

Synchronization by hardware

- Molti sistemi ricorrono all'hardware per la risoluzione del problema della sezione critica:
  - Nei monoprocessori: si possono disabilitare gli interrupt durante l'accesso alle sezioni critiche in maniera tale che la sequenza di istruzioni corrente venga eseguita in ordine senza interruzioni.
  - Nei sistemi multiprocessore: di solito la disabilitazione degli interrupt è troppo inefficiente, perchè i comandi per la disabilitazione devono essere passati a tutti i processori.

- I calcolatori moderni risolvono efficacemente il problema fornendo **almeno una delle seguenti istruzioni hardware speciali che funzionano in modo atomico**.

1. Controllare e modificare il contenuto di una parola di memoria → `TestAndSet(boolean &target)`
2. Scambiare il contenuto di due parole. → `Swap(boolean &a, boolean &b)`

Synchronization instruction: Test&Set

- Controlla e modifica atomicamente il contenuto di una variabile booleana (o generica).

```
boolean TestAndSet(boolean &target) {
            boolean rv = target;
            target = true;
            return rv;
                                    }
```

Shared data:

```
boolean lock = false;
```

Process P_i

```
        do      {
                while (TestAndSet(lock)) ;
                        critical section
                lock = false;
                        remainder section
                }
```

Synchronization instruction: Swap

- Scambia atomicamente il valore di due variabili booleane (o generiche).

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

Shared data (initialized to false):

```
boolean lock;
boolean waiting[n];
```

Process P_i

```
        do      {
                key = true;
                while (key == true)
                        Swap(lock,key);
                        critical section
                        lock = false;
                        remainder section
                }
```

Binary Semaphore

Synchronization tool that does not require busy waiting.

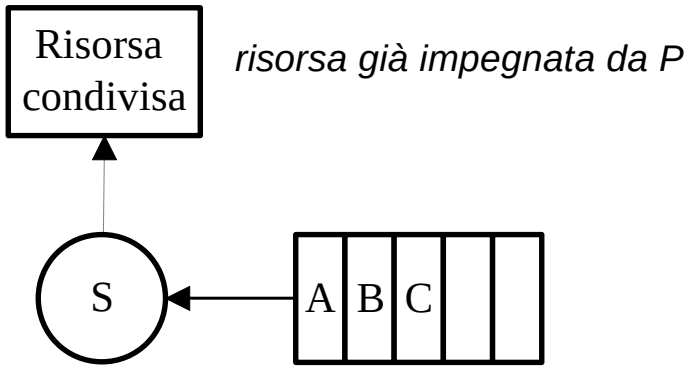
- Variabile semaforica *S* – variabile intera → può variare solo tra 0 e 1
- può essere utilizzata solo attraverso due operazioni indivisibili (atomiche)

wait (S): down(S): attendi finchè S è maggiore di 0; quindi decrementa S

```
while S≤ 0 do no-op;
S--;
```

signal (S): up(S): incrementa S

```
S++;
```



coda dei processi in attesa

- Normalmente, l'attesa è implementata spostando il processo in stato di wait, mentre la up(S) mette uno dei processi eventualmente in attesa nello stato di ready.
- I nomi originali delle operazioni erano **P** (proberen, testare), **V** (verhogen, incrementare)

Counting Semaphore

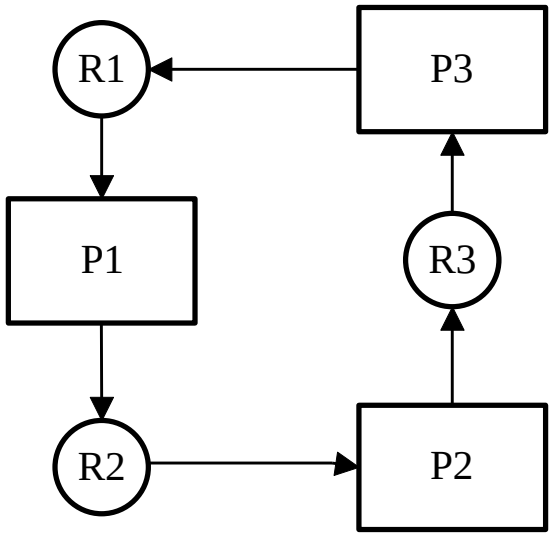
- **Semaforo contatore** (o generalizzato) – la variabile semaforica può variare in un dominio senza restrizioni, p.e. tra 0 e 10. Si può adoperare per controllare l'accesso ad una risorsa con un numero finito di istanze

- Può essere realizzato con caratteristiche simili a quelle di un semaforo binario.

  - S viene inizializzato al numero di istanze della risorsa disponibili.
    - Un processo che desidera accedere alla risorsa esegue una wait () su S (decrementando il contatore).
    - Un processo che rilascia la risorsa esegue una signal () (incrementando il counter).
    - Quando S=0, tutte le istanze sono allocate e i processi che richiedano la risorsa resteranno in coda in attesa del rilascio di un esemplare di essa.
  - Il semaforo può assumere anche valori negativi. Il valore assoluto rappresenta il numero di processi in coda di attesa.

Mutual exclusion risks: Deadlock and Starvation

- **Deadlock** – due o più processi attendono indefinitamente un evento che può essere causato solo da uno dei processi in attesa.



- **Starvation** (*indefinite blocking*) – un processo non può mai essere rimosso dalla coda di un semaforo in cui è sospeso in attesa.
- In informatica, per starvation (termine inglese che tradotto letteralmente significa **inedia**) si intende l'impossibilità, da parte di un processo pronto all'esecuzione, di ottenere le risorse hardware di processamento di cui necessita per essere eseguito. Un esempio è il non ottenere il controllo della CPU da parte di processi con priorità molto bassa, qualora vengano usati algoritmi di scheduling a priorità. Può capitare, infatti, che venga continuamente sottomesso al sistema un processo con priorità più alta. È la storia del processo con bassa priorità, scoperto quando fu necessario fermare il sistema sull'IBM 7094 al MIT nel 1973: era stato sottomesso nel 1967 e fino ad allora non era stato eseguito.

Monitor

Benché i semafori forniscano un meccanismo conveniente ed efficace per la sincronizzazione, il loro utilizzo non corretto comporta errori di temporizzazione difficilmente individuabili.

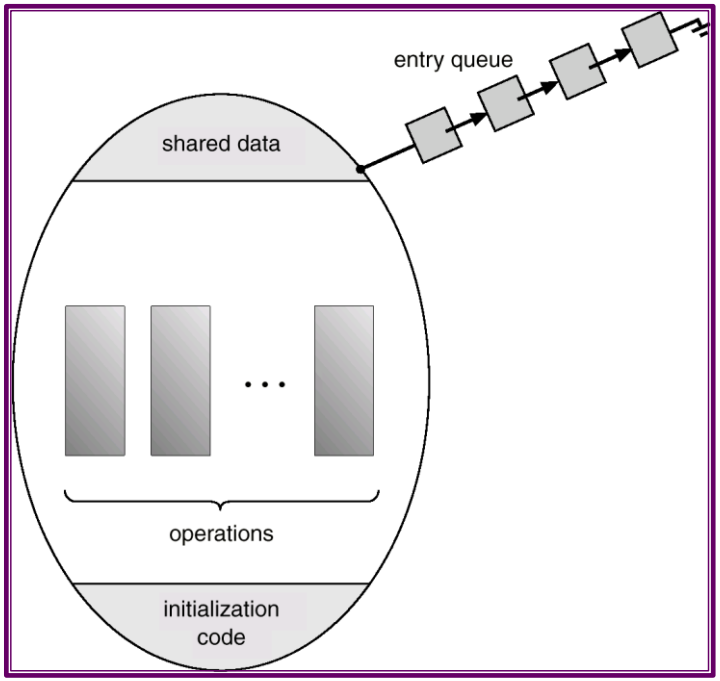
Un costrutto di sincronizzazione di alto livello che garantisce completamente la condivisione di ***strutture di dati astratti*** da parte di processi concorrenti è, invece, il **monitor**.

- Un monitor è un tipo di dato astratto che fornisce funzionalità di mutua esclusione; è, cioè, una collezione di dati privati e funzioni/ procedure per accedervi
  - i processi non possono accedere direttamente alla struttura interna di un monitor (variabili locali o condivise), ma possono chiamare le procedure pubbliche, che sono le sole che vi possono accedere.
  - un solo processo alla volta può eseguire codice di un monitor
  - il programmatore raccoglie quindi i dati condivisi e tutte le sezioni critiche relative in un monitor; non deve perciò codificare la sincronizzazione esplicitamente
- I monitor sono implementati dal compilatore con dei costrutti per mutua esclusione

```
monitor monitor-name
{shared variable declarations
        procedure body P1 (…)
        {       . . .
                }
        procedure body P2 (…)
        {       . . .
                }
        procedure body Pn (…)
        {       . . .
                }

                {
                initialization code
                }
}
```

Monitor



Per consentire ad un processo di attendere nel monitor fintanto che si verifichi uno specifico evento, è necessario dichiarare una specifica **variabile condizione**

```
condition x, y;
```

Monitor

Una variabile condizione è solo usata con gli operatori di sincronizzazione **`wait`** e **`signal`**.

- L'operazione **`x.wait();`** chiede che il processo che la lancia venga sospeso finchè un altro processo effettua un'operazione **`x.signal;`**
- l'operazione **`x.signal();`** chiede che venga ripresa l'esecuzione di un processo sospeso in attesa della condizione x. Se non c'è alcun processo sospeso su x, l'operazione **`signal`** non ha alcun effetto.