9 - Concorrenza, Mutua Esclusione e Sincronizzazione OK

Terminologia

Leggere attentamente! Utile per capire diverse nozioni qui:

- **Sezione critica**: insieme di istruzioni attraverso le quali si richiede l'accesso e si gestisce la risorsa condivisa
- **Deadlock** (semplice): situazione in cui **almeno** due o più processi sono impossibilitati dal procedere poiché sono in attesa l'uno dell'altro
- **Livelock** (stallo attivo): situazione in cui **almeno** due o più processi cambiano continuamente il proprio stato a causa del cambiamento di stato degli altri (senza fare alcun lavoro utile)
- **Race condition**: situazione dove i thread o processi leggono e scrivono un dato condiviso e il risultato dipende dalla loro velocità, il più veloce vince rispetto a quello più lento
- **Starvation**: situazione nella quale un processo non riceve mai l'utilizzo di una risorsa e viene costantemente scavalcato da altri processi
- **Variabile a guardia di una risorsa**: Variabile che stabilisce l'occupazione di una risorsa, per convenzione quando tale variabile ha valore **true** la risorsa è occupata.

Concorrenza

La concorrenza è la competizione tra processi (o threads) per ottenere (e condividere) le risorse come CPU, memoria, I/O, files etc.
Il problema della concorrenza nasce dall'implementazione e l'utilizzo di:

- Multiprogrammazione (gestire più processi su un singolo processore)
- Multiprocessing (gestire più processi su più processori)
- Multi-Threading (a livello di pipeline)

Problemi derivati dalla concorrenza

Sistemi a singolo processore

- La condivisione diventa pericolosa, poiché l'ordine delle operazioni di lettura e scrittura su aree di memoria condivise potrebbe portare a perdite di dati
- Esiste una difficoltà ad assegnare le risorse ai processi in maniera ottimale
- Diventa difficoltosa la rilevazione di errori nel codice e dei conflitti di interlacciamento (race-conditions)

Esempio:

```
void echo()
{
    char in,out;
    scanf("%c", &in);
    out = in;
    printf("%c", out);
```

Condivisione della procedura echo():
- Risparmio dello spazio di memoria
- Due processi concorrenti.
- P1 viene interrotto dopo la scanf,
- P2 esegue tutto echo,
- P1 viene riattivato da scanf in poi e ha perso il dato che aveva letto

Sistemi SMP

- Stessi problemi di un calcolatore a singolo processore
- Interlacciamento di esecuzione di processi paralleli:
  Quando processi paralleli operano contemporaneamente, possono verificarsi situazioni di **perdita di aggiornamenti**, in cui i dati letti o scritti vengono sovrascritti o perduti a causa della mancanza di coordinamento tra i processori.
  Esempio:

Processo P1 – Processore1	Processo P2 – Processore2
……	……
scanf("%c", &in);	……
……	scanf("%c", &in);
out = in;	out = in;
printf("&c", out);	……
……	printf("&c", out);

Il carattere letto da P1 è perso prima di poter essere stampato (perdita di aggiornamento)

Il risultato finale dipende dalla velocità degli n (in questo caso 2) processi, quello più veloce avrà la prevalsa

Mutua esclusione

La soluzione ai problemi della concorrenza quindi è la mutua esclusione, un requisito (che noi programmatori richiediamo) per il quale, se un processo è nella propria sezione critica, nessun'altro processo può entrare nella propria sezione critica se questa fa riferimento a risorse condivise con il primo processo

Requisiti per la mutua esclusione

Ci sono dei requisiti fondamentali per poter implementare e garantire la mutua esclusione:

1. Un solo processo alla volta deve accedere alla sezione (o risorsa) critica
2. Un processo fuori dalla sezione critica non deve interferire con il processo nella sezione critica (come la richiesta continua della sezione critica)
3. Ogni processo deve poter accedere dopo un tempo finito di attesa in coda alla risorsa critica (senza creare situazioni di stallo o starvation)

4. Se nessun processo è nella sezione critica, un processo deve poter entrare nella sezione critica senza tempi di attesa
5. Non ci devono essere ipotesi di velocità di esecuzione dei processi
6. Il tempo di permanenza nella sezione critica è finito e definito
   Un processo quando entra nella sezione critica, che abbia finito o no le sue operazioni importanti, deve rilasciare la risorsa allo scadere del tempo

Meccanismi di Mutua esclusione

A questo soluzione è possibile applicare più algoritmi:

- **Approccio software**: soluzione totalmente a carico del programmatore che scrive un pezzo di codice per la coordinazione dei processi (Algoritmo di Dekker)
  Questo approccio ha come svantaggi:
  - Aumento dei tempi di esecuzione
  - Errori frequenti dati da un codice pensato o scritto male
- **Approccio hardware**: utilizzo di istruzioni macchina, tipicamente test-set o swap
- **Supporto del sistema operativo**: il sistema operativo mette a disposizione dei costrutti specifici (monitor o semafori) che sono in grado di gestire la mutua esclusione

=Quali delle 3 uso? Sicuramente non quella software, superata da oramai tanto tempo (a meno di casi molto particolari) =

Approccio software: Algoritmo di Dekker

L'algoritmo di Dekker è un metodo progettato per garantire la mutua esclusione tra due processi che devono accedere a una risorsa condivisa, è basato sull'uso di due variabili principali:

- Una per indicare l'intenzione di ogni processo di accedere alla risorsa
- Un'altra per stabilire quale processo ha la priorità in caso di conflitto.

Il funzionamento si basa su una combinazione di comunicazione e attesa;
Quando un processo decide di accedere alla risorsa, dichiara questa intenzione impostando una variabile condivisa, se l'altro processo non sta cercando di accedere, può entrare direttamente nella sezione critica.
Tuttavia, se entrambi i processi vogliono accedere contemporaneamente, l'algoritmo utilizza una variabile aggiuntiva, chiamata turno, per stabilire chi può procedere per primo, il processo che non ha il turno aspetta fino a quando l'altro ha completato l'accesso alla risorsa.

Questo approccio garantisce che solo un processo alla volta possa entrare nella sezione critica, evitando conflitti e garantendo che entrambi i processi abbiano la possibilità di accedere alla risorsa e inoltre non richiede dipendenze esterne come meccanismi hardware o supporto del sistema operativo.

Tuttavia, presenta alcuni limiti, uno dei principali è l'attesa attiva, che implica che un processo in attesa consuma cicli di CPU controllando continuamente lo stato dell'altro e inoltre, se un processo si blocca all'interno della sezione critica, l'altro rimane in attesa indefinitamente. Questi aspetti rendono l'algoritmo meno efficiente rispetto a soluzioni più moderne, anche se il suo design rimane un esempio storico di come risolvere il problema della concorrenza in maniera indipendente.

Esempio:

```
boolean flag[2];
int turno;
void main()
{     flag[0]=false;          flag[1]=false;              turno=1; //turno=0;
      …
      processo P0;
      processo P1;
      …
}

//processo P0
{
      …
            flag[0]=true;
            while(flag[1])
            {
                  if (turno==1)
                  {
                        flag[0]=false;
                        while(turno==1)
                              {<nulla…ATTESA ATTIVA>}
                        flag[0]=true;
                  }
             }
            <sezione critica>
            flag[0]=false;
      …
}
```

Approccio Hardware

Esiste un supporto da parte del costruttore del calcolatore per la mutua esclusione, un modo per poter ottenere la mutua esclusione nelle macchine monoprocessore è prevenire l'interruzione di un processo attivando e disattivando gli interrupt in modo seguente:

```
<disattiva le interruzioni>
<sezione critica>
<attiva le interruzioni>
```

Il problema della mutua esclusione viene risolto a discapito di un inefficienza, il processore non può liberamente alternare i processi, oltre a non funzionare su macchine SMP (se nel core 1 viene avviata una sezione critica la stessa potrebbe essere avviata su altri core, la soluzione sarebbe disabilitare gli interrupt a tutti, praticamente impossibile).

Nelle macchine SMP allora si è deciso di creare delle istruzioni macchina speciali per l'accesso a locazioni di memoria in modo atomico (o non interrompibile) come:

- **Test&Set**
- **Scambio (swap)**
  L'accesso sequenziale ad una locazione di memoria è garantita dall'hardware, infatti se vengono eseguite due Test&Set contemporaneamente, esse vengono serializzate, cioè eseguite in sequenza d'ordine

I **vantaggi** che portano le implementazioni hardware di questo genere sono:

- Applicazione a un qualsiasi numero di processi, anche su multiprocessori a memoria condivisa (Nella memoria centrale risiede il valore delle variabili a guardia della sezione critica, viene passato alle funzioni tramite puntatori con indirizzo della)
- Si può usare per gestire più di una sezione critica, ciascuna con una propria variabile;

Alcuni **svantaggi** di queste implementazioni hardware sono:

- Attesa attiva (i processi consumano tempo di CPU, i while controllano continuamente)
- Starvation (la scelta di quale processo andrà nella sezione critica è arbitraria, il processo che vince la sezione critica è totalmente casuale)
- Stallo su singolo processore
  - P1 entra nella sezione critica
  - P2, con priorità più alta di P1
  - P2 tenterà di accedere (tramite test&set o swap) alla risorsa bloccata da P1 le nella sua attesa attiva non lascerà mai il posto a P1 che ha priorità più bassa

Test&Set

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

La funzione Test&Set accetta un puntatore a una variabile booleana target, che restituisce il valore corrente di target e imposta target a TRUE.

```
boolean lock = FALSE; // La risorsa è inizialmente libera

while (true) {
    while (TestAndSet(&lock))
        ; // Attesa attiva
    // Sezione critica
    lock = FALSE; // Rilascio della risorsa
```

```
    ...
}
```

In una realizzazione si utilizza una variabile a guardia di una risorsa chiamata `lock`, inizializzata a `FALSE`;
Quando un processo tenta di entrare nella sezione critica chiamando `TestAndSet(&lock)` controlla il valore di ritorno, se quest'ultimo metodo restituisce FALSE il processo può entrare, altrimenti deve attendere.

Swap

```
void swap (boolean *a, boolean *b) {
    boolean temp = *a; // Salva il valore di a in temp
    *a = *b;           // Copia il valore di b in a
    *b = temp;         // Copia il valore di temp (originale di a) in b
}
```

La funzione swap accetta due puntatori a variabili booleane `a` e `b`, scambiando i loro valori.

```
boolean lock = FALSE; // Variabile condivisa inizializzata a FALSE (risorsa
libera)
boolean key;          // Variabile locale per ogni processo

while (true) {
    key = TRUE;  // Imposta key a TRUE per tentare di acquisire il lock
    while (key == TRUE) {
        swap(&lock, &key); // Scambia lock e key
        // Se key diventa FALSE, il processo ha acquisito il lock
    }
    // Sezione critica: codice che accede alla risorsa condivisa
    lock = FALSE; // Rilascia la risorsa impostando lock a FALSE
    ...
}
```

Nell'esempio di mutua esclusione si utilizza una variabile `lock` inizializzata a `FALSE`, un processo usa `swap` per tentare di entrare nella sezione critica.
Se `key` diventa `FALSE`, il processo può entrare

Approccio del SO e linguaggi di programmazione: I semafori

Esiste un costrutto (o una classe) fornito dal sistema operativo che gestisce l'accesso della risorsa condivisa (o alla sezione critica).
Questo costrutto viene chiamato **semaforo**, una variabile condivisa su cui sono possibili 3 operazioni:

- Inizializzazione ad un valore non negativo
- Operazione di **wait()**, decremento di una unità del valore della variabile. Se questo valore diventa negativo, il processo che ha eseguito la wait viene bloccato
- Operazione di **signal()**, incremento di una unità del valore della variabile. Se questo valore è negativo, uno dei processi bloccati sull'operazione di wait() viene sbloccato.

Un processo normalmente quindi attiva la funzione di wait() per poter utilizzare la risorsa e la rilascia con la funzione di signal(), i processi non possono accedere direttamente alla variabile condivisa ma solo tramite le funzioni definite.
La variabile numerica indica il numero di istanze specifiche per quella risorsa, nel caso sia un valore negativo rappresenta il numero di processi in attesa;
Tutte le modifiche al valore del semaforo contenute nelle operazioni wait() e signal() si devono eseguire in modo **atomico**: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore.

Implementazione dei semafori con contatore

```
// Struttura dati per un semaforo contatore
typedef struct {
  int istanze;               // Numero di risorse disponibili
  struct processo *P;        // Lista dei processi in attesa, puntatore al
PCB di ogni processo
} semaforo;

// Operazione di attesa (wait)
void wait(semaforo s) {
  s.istanze--;                // Decrementa il contatore delle risorse
  if (s.istanze < 0) {        // Se non ci sono risorse disponibili
    <poni processo in coda>;
    <blocca questo processo: running->blocked>;
  }
}

// Operazione di segnalazione (signal)
void signal(semaforo s) {
  s.istanze++;                // Incrementa il contatore delle risorse
  if (s.istanze <= 0) {       // Se ci sono processi in attesa
    <rimuovi un processo in coda>;
    <sveglia il processo: blocked->ready>;
  }
}
```

Implementazione dei semafori binari

I semafori binari (o booleani) hanno una variabile booleana con unici valori di 0 e 1, si utilizza in situazioni in cui è necessario garantire **mutua esclusione** per l'accesso a una

singola risorsa condivisa o sincronizzare due processi.

```
// Struttura dati per un semaforo binario
typedef struct {
  boolean val;              // Stato del semaforo: 1 (risorsa libera) o 0
(risorsa occupata)
  struct processo *P;       // Lista dei processi in attesa, puntatore al
PCB di ogni processo
} semaforo_bin;

// Operazione di attesa (wait)
void wait(semaforo_bin s) {
  if (s.val == 1)           // Se la risorsa è libera
    s.val = 0;              // Occupa la risorsa
  else {
    <poni processo in coda a P>;
    <blocca questo processo: running->blocked>;
  }
}

// Operazione di segnalazione (signal)
void signal(semaforo_bin s) {
  if (*P == NULL)           // Se la coda di attesa è vuota
    s.val = 1;              // Libera la risorsa
  else {
    <rimuovi un processo in coda>;
    <sveglia il processo: blocked->ready>;
}
```

Implementazioni atomiche

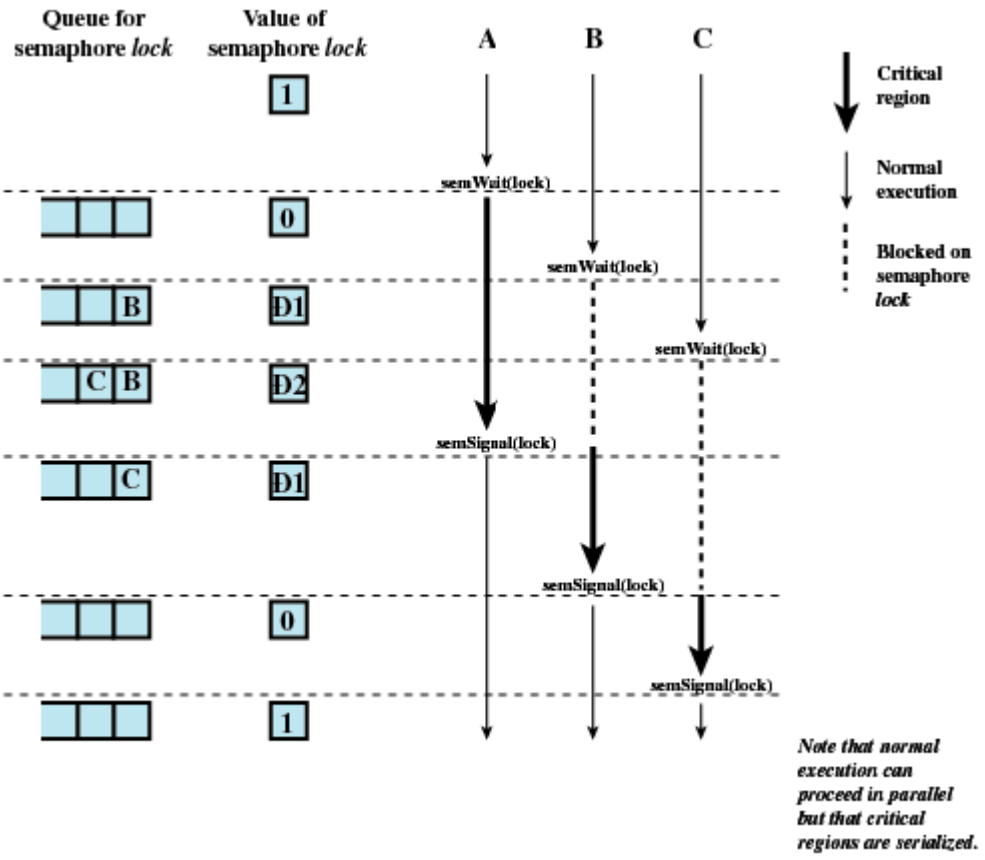
Come detto precedentemente, la modifica della variabile del semaforo deve avvenire in modo atomico, ad oggi si conoscono diverse modalità:

- **Hardware**: nell'architettura **Pentium** (e più in generale nei processori **x86**), esistono istruzioni specifiche che permettono di rendere atomiche le operazioni di wait() e signal(), esistono alcuni prefissi come
  - **LOCK** : può essere usato con alcune istruzioni di memoria per garantire che l'operazione venga eseguita in modo atomico, segnala alla CPU di bloccare il bus della memoria fino a quando l'operazione non è completata, impedendo l'accesso concorrente ad altre CPU o core.
  - **XCHG (Exchange)**: scambia il contenuto di un registro con quello di una variabile in memoria, è atomica **di default** sugli x86, senza bisogno di aggiungere il prefisso **LOCK**
  - **CMPXCHG (Compare and Exchange)**: confronta il valore di un'area di memoria con un registro e, se i due valori coincidono, aggiorna l'area di memoria con un nuovo valore. Comunemente usata per implementare algoritmi come **Compare-And-Swap**.
  Altrimenti si potrebbero usare istruzioni come Test&Set.

- **Software**: Dekker o Peterson
- Disabilitare gli **interrupt** nei sistemi monoprocessori

Accesso a dato condiviso tramite uso di semafori

Piccolo schema per la gestione di risorse condivise



Deadlock e Starvation

Il deadlock si può ritrovare nell'approccio del sistema operativo:

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);

- *P1 non rilascia S fino a quandonon ottiene Q*
- *P0 non rilascia Q fino a quando non ottiene S*

Le signal non saranno mai eseguite: STALLO

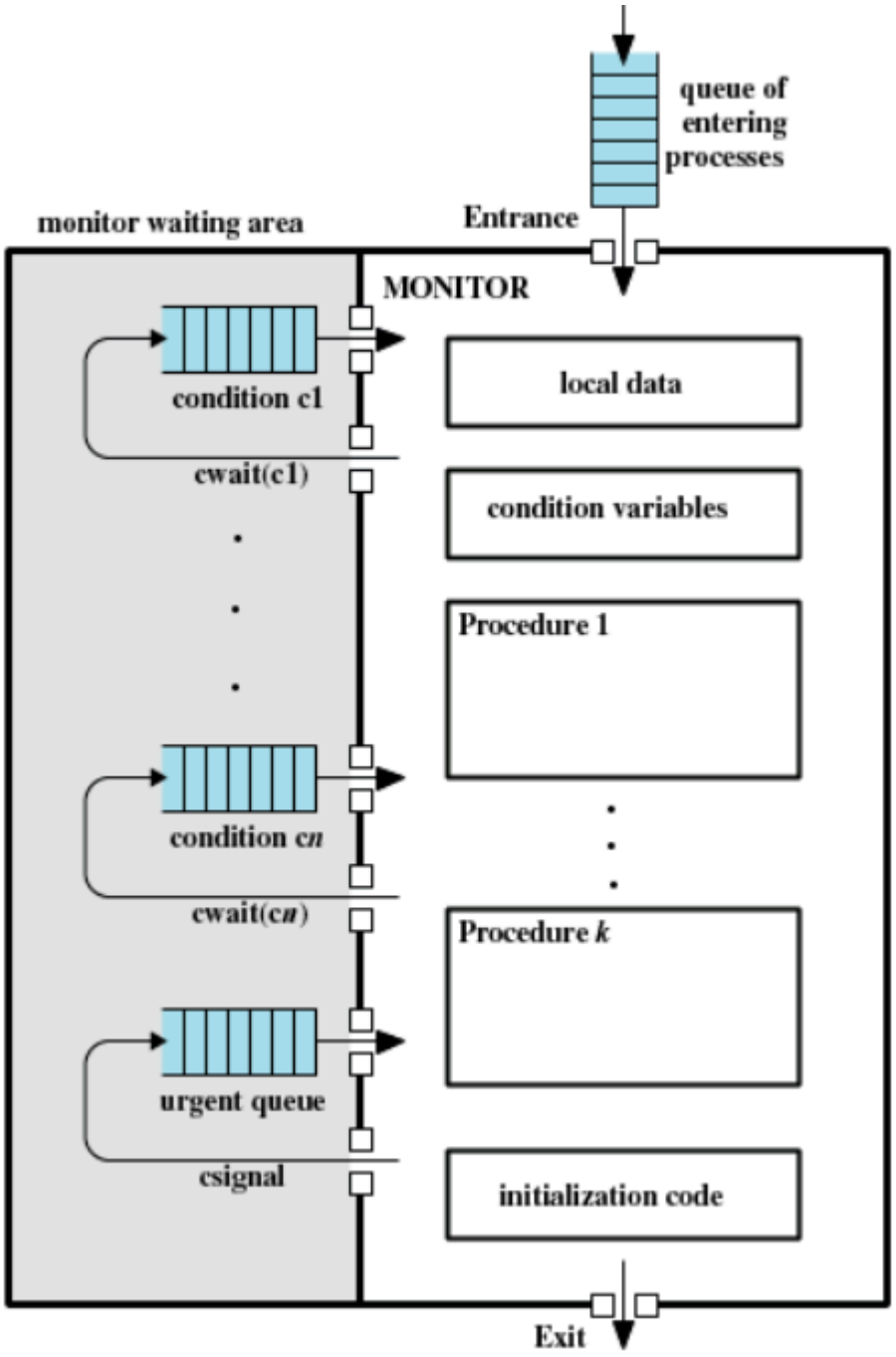
Nel caso della starvation invece il processo non viene mai rimosso dalla coda del semaforo (con una gestione LIFO)

Monitor

I monitor sono un costrutto di sincronizzazione di alto livello utilizzato nella programmazione concorrente, rappresentano una soluzione modulare per la gestione della **mutua esclusione** e la protezione delle strutture dati condivise, semplificando il coordinamento tra processi.

I monitor presentano caratteristiche come:

1. **Modularità**: Un monitor è un modulo software contenente **variabili locali** accessibili solo dalle procedure definite al suo interno, non sono visibili né modificabili dall'esterno, garantendo un controllo rigoroso sull'accesso ai dati condivisi.
2. **Mutua esclusione**: Solo un processo può eseguire il codice all'interno del monitor in un dato momento, questo evita condizioni di corsa e garantisce che l'accesso alle risorse condivise avvenga in modo controllato.
3. **Procedure (metodi)**: I processi accedono al monitor invocando le sue procedure, la logica di gestione delle risorse condivise è quindi centralizzata all'interno del monitor stesso.
4. **Sincronizzazione tramite variabili di condizione**: I monitor utilizzano **variabili di condizione** per gestire situazioni in cui i processi devono attendere determinati eventi. Le due operazioni principali su queste variabili sono:
   - `cwait(c)` : Sospende l'esecuzione del processo chiamante fino a quando una determinata condizione (c) non diventa vera.
   - `csignal(c)` : Risveglia un processo sospeso sulla condizione c, se presente. In assenza di processi in attesa, il segnale viene perso.


queue of
entering
processes
monitor waiting area
Entrance
MONITOR
condition c1
cwait(c1)
local data
condition variables
Procedure 1
condition cn
cwait(cn)
Procedure k
urgent queue
csignal
initialization code
Exit