Processi concorrenti : indipendenti e interagenti

Con il termine di processi concorrenti viene indicato **un insieme di processi la cui esecuzione si sovrappone nel tempo o quando svolgono operazioni su una medesima risorsa**. Nel caso in cui ogni processo possiede la sua unità di esecuzione si ha un **overlapping** degli stessi; nel caso in cui condividano la CPU si ha il caso dell'**interleaving**.
La concorrenza stabilisce che due processi si dicono tali quando la prima operazione di uno inizia prima dell'ultima dell'altro. I processi concorrenti possono comportarsi come indipendenti o come interagenti. Un insieme di processi si dice indipendente **quando nessuno di essi può influenzare l'esecuzione degli altri**. Processi che non condividono dati e che non si scambiano informazioni sono processi indipendenti. Caratteristica fondamentale di un processo indipendente è che il suo **comportamento** è **riproducibile**, ovvero produce sempre lo stesso risultato in ogni sua istanza.
Caratteristica, invece, dei processi interagenti è la possibilità di **influenzarsi vicendevolmente durante l'esecuzione**. Questo può avvenire in modo esplicito mediante scambio di messaggi o segnali temporali (**cooperazione**), oppure in modo implicito tramite **competizione** per la stessa risorsa. Poiché gli effetti delle interazioni tra processi dipendono dalla velocità degli stessi, ed essendo queste ultime non riproducibili, il **comportamento** di suddetti processi **non è riproducibile.**

Cooperazione tra processi

La cooperazione tra processi prevede tra gli stessi uno **scambio di informazioni**. Il caso più semplice è quello in cui l'unica informazione scambiata è costituita da un segnale temporale senza trasferimento di dati. Esiste pertanto un **vincolo di precedenza** tra l'operazione con la quale viene inviato il segnale da parte del processo gestore e la prima operazione di un processo ricevente. Il rispetto di questi vincoli impone una **sincronizzazione** dei due processi, nel senso che il processo che esegue una specifica attività non può iniziare la sua esecuzione prima dell'arrivo del segnale da parte del processo gestore. Nel caso in cui sia previsto anche uno scambio di dati, oltre che una sincronizzazione è necessaria anche una **comunicazione**.

Competizione tra processi

La competizione tra processi si ha quando questi richiedono l'uso di **risorse comuni che non possono essere usate contemporaneamente**. Questo tipo di interazione non è insito nella

logica dei processi ma imposto da vincoli di reale disponibilità delle risorse stesse. Anche in questo caso esiste un vincolo di precedenza tra le operazioni con le quali i processi possono accedere alla risorsa comune, ma mentre nel caso precedente il vincolo richiedeva un ordinamento tra le operazioni, in questo caso l'ordine di accesso alla risorsa è indifferente purché le operazioni siano mutuamente esclusive nel tempo.

Immagine del processo

Un processo è costituito da un insieme di **locazioni** per i dati, le **variabili globali e locali**, dallo **stack** e dal suo **descrittore**. L'insieme di queste informazioni prende il nome di immagine del processo. Al processo sono inoltre assegnate alcune risorse, come file aperti, processi figli, dispositivi I/O ecc..

Spazio di indirizzamento

Ogni processo ha uno spazio di indirizzamento distinto da quello di altri processi. Esso contiene **l'immagine del processo e le risorse** ad esso assegnate. La locazione dello spazio di indirizzamento dipende dalla tecnica di gestione della memoria adottata.

Modello ad ambiente globale

In un modello ad ambiente globale ogni tipo d'interazione tra i processi avviene tramite la **memoria comune**. Ogni applicazione viene strutturata come un insieme di processi e risorse. In un modello ad ambiente globale entrambe le forme di interazione (cooperazione e competizione) avvengono tramite l'utilizzo di **risorse globali**; i processi competono per l'utilizzo di risorse comuni e le utilizzano per lo scambio di informazioni. Viene utilizzato il modello del semaforo per la sincronizzazione.

Modello ad ambiente locale

Nel modello ad ambiente locale ogni processo opera esclusivamente su **proprie variabili** alle quali non possono accedere direttamente altri processi. Esiste anche in questo modello il concetto di risorsa comune ma essa è, tuttavia, locale ad un processo che ne risulta il suo **gestore**. Quando un processo intende operare sulla risorsa deve comunicare al processo gestore questa esigenza inviando un opportuno messaggio; il processo gestore svolgerà l'operazione richiesta agendo sulla risorsa e comunicherà eventualmente l'esito al richiedente.

Mutua esclusione

La competizione tra processi su risorse comuni che possono essere utilizzate da un solo processo alla volta impone il vincolo della mutua esclusione delle operazioni con cui i processi accedono alle risorse stesse, ovvero che **l'accesso alle risorse da parte delle operazioni dei processi non si sovrappongano le une con le altre**. Nessun vincolo è invece imposto sull'ordine con il quale le operazioni sulle variabili comuni vengono eseguite. Tali operazioni prendono il nome di **sezioni critiche**. Con tale termine si intende quindi una porzione di codice che accede a una **risorsa condivisa** tra più flussi di esecuzione (processi) di un sistema concorrente.

Prologo e Epilogo

Ogni processo, prima di entrare in una sezione critica dovrà richiedere l'**autorizzazione** eseguendo una serie di operazioni che gli garantiscano l'**accesso esclusivo** alla risorsa, se questa è libera, oppure ne impediscano l'accesso se questa è occupata. Questa serie di istruzioni prende il nome di **prologo**, mentre con **epilogo** si intende l'insieme di istruzioni eseguite dal processo per dichiarare libera la sezione critica al completamento della sua azione.
Il modo più semplice per definire il prologo e l'epilogo è mediante l'utilizzo di una **variabile condivisa**, *occupato*, che può assumere valore 1 (risorsa occupata), o 0 (risorsa libera).

Lock e Unlock

Molte macchine posseggono particolari istruzioni che permettono di esaminare e modificare il contenuto di una parola o di scambiare il contenuto di due parole in un ciclo di memoria. Un esempio tipico di queste istruzioni è rappresentato dall'istruzione **TSL** (Test and Set Lock) in asm. L'esecuzione di **TSL R , x** funziona nel seguente modo : il valore contenuto nella locazione x viene copiato nel registro R del processore e viene scritto in x un valore diverso da 0. Le operazioni di lettura e scritture sono eseguite in modo **indivisibile** perché la CPU che esegue l'istruzione TSL **blocca il bus di memoria** per impedire ad altre CPU di accedere alla memoria finché non ha completato la TSL.
La mutua esclusione si ottiene introducendo due funzioni : **lock(x)** e **unlock(x)**.

```
lock(x):
    TSL registro, x
    CMP registro, 0
    JNE lock
    RET
```

```
unlock(x):
MOVE x,0
RET
```

Questa soluzione è caratterizzata da condizioni di **attesa attiva** dei processi che **non** possono entrare nella sezione critica richiesta (busy form of waiting). Tali processi infatti **ripetono la sequenza di istruzioni** con la quale richiedono l'accesso alla sezione critica, tenendo occupato il processore sul quale sono in esecuzione.

Semafori

Un semaforo *s* è una struttura dati alla quale sono applicabili solo due operazioni primitive e indivisibili : **wait(s)** e **signal(s)**. La struttura dati è costituita da una variabile intera non negativa, s.value, con valore iniziale =0, e da una coda di processi (s.queue) sospesi.
L'operazione di wait viene utilizzata da un processo per **verificare lo stato di un semaforo**, se il valore è positivo questo viene decrementato di un'unità e il processo prosegue l'esecuzione; se il valore è nullo lo stato del processo passa da attivo a bloccato e la wait inserisce il suo PCB nella coda associata al semaforo. L'esecuzione del processo è quindi sospesa e la CPU viene passata ad un altro processo.

```
void wait(s){
    if(s.value != 0){ //se non c'è nessuno in esecuzione
        s.value--; //il valore viene decrementato (blocca gli altri processi dall'accedere)
        //inizia l'esecuzione della sezione critica
    }
    else
    //il processo viene sospeso perché il semaforo è occupato
    //il descrittore viene inserito nella coda
}
```

L'operazione di signal viene utilizzata **per risvegliare eventuali processi sospesi sul semaforo**, se non esistono il valore viene incrementato di uno; diversamente viene attivato il primo processo in coda. A differenza della wait, l**a signal non prevede la sospensione del processo** che la esegue, salvo politiche di scheduling (priorità, round robin, ecc...), quindi l'esecuzione potrebbe, logicamente, continuare.

```
void signal(s){
    if(s.queue != null){ // se c'è almeno un PCB nella coda
        //il primo PCB viene estratto dalla coda e il suo stato passa a pronto
    }
    else s.value++;
}
```

Soluzione al problema della mutua esclusione

Si ottiene associando alla risorsa condivisa, il cui utilizzo deve essere mutuamente esclusivo da parte di un insieme di processi, un **semaforo *mutex*** (binario) **inizializzato ad 1**.

```
wait(mutex);
//sezione critica
signal(mutex);
```

La soluzione presentata evita condizioni di attesa attiva in quanto l'impossibilità di accedere alla propria sezione critica comporta la sospensione del processo.

Tuttavia rimane il problema dell'ottenere l'**indivisibilità delle operazioni wait e signal**. Nell'ipotesi che tutti i processi coinvolti nel problema della mutua esclusione operino sullo **stesso processore**, l'indivisibilità delle primitive è garantita dalla **disabilitazione delle interruzioni** del processore durante la loro esecuzione. Qualora i processi siano eseguiti su **processori diversi** bisogna ricorrere alle funzioni **lock(x)** e **unlock(x)** :

```
lock(x);
    wait(mutex);
unlock(x);
//sezione critica
lock(x);
    signal(mutex);
unlock(x);
```

L'esecuzione della lock(x) **garantisce che la wait(mutex) possa essere seguita solo da un processo alla volta**. Durante l'esecuzione della wait(mutex) da parte di un processo, gli altri processi che chiamano la wait(mutex) **si chiudono in un ciclo** fino all'esecuzione della unlock(x). Lo stesso ragionamento vale per la signal(mutex).
Il **semaforo** mutex (con le primitive wait e signal) **assicura la mutua esclusione delle sezione critiche** su una risorsa R, mentre **la variabile x (con le primitive lock e unlock) assicura la mutua esclusione delle primitive wait e signal** sul semaforo mutex.

Produttore-Consumatore

Un ben noto problema nel campo della comunicazione tra processi è quello conosciuto come **produttore-consumatore**.
Un processo (**il produttore**) **genera ciclicamente un messaggio** e lo deposita in un'area di memoria (un **buffer**), capace di contenere **un solo messaggio** alla volta; un secondo processo (**il consumatore**) **preleva**

dall'area di memoria il messaggio e lo utilizza. I vincolo imposti nell'esecuzione delle due operazioni sono i seguenti :

1. il produttore non può inserire nel buffer un nuovo messaggio prima che il consumatore abbia prelevato il precedente;
2. il consumatore non può prelevare dal buffer un nuovo messaggio prima che il produttore lo abbia depositato.

Se il buffer può invece contenere **più messaggi** valgono le seguenti condizioni :

1. il produttore non può inserire un messaggio nel buffer se questo è già pieno;
2. il consumatore non può prelevare un messaggio dal buffer se questo è vuoto.

La soluzione a questo problema impone un ordinamento nelle operazioni dei due processi e per ottenere ciò è necessario che i due processi si scambino dei segnali per indicare l'avvenuto deposito e prelievo dei messaggi dal buffer : tutto questo può essere implementato con 2 semafori e le primitive wait e signal.

```
Processo Produttore{
    do{
        //produzione del messaggio;
        wait(spazio_disponibile);
        //il messaggio viene depositato nel buffer;
        signal(messaggio_disponibile);
    }while(!fine)
}

Processo Consumatore{
    do{
        wait(messaggio_disponibile);
        //il messaggio viene prelevato dal buffer;
        signal(spazio_disponibile);
        //il messaggio viene usato;
    }while(!fine)
}
```

Nel caso di un buffer con capacità di **un singolo messaggio**, esso sarà inizialmente vuoto e i due semafori, indicati con *messaggio_disponibile* e *spazio_disponibile*, avranno valori, rispettivamente, 0 e 1.

Nel caso in cui il buffer possa contenere **N messaggi**, il valore iniziale del semaforo *spazio_disponibile* sarà N, mentre il valore dell'altro semaforo sarà comunque 0.
Il produttore verifica la disponibilità di spazio nel buffer tramite la primitiva wait(*spazio_disponibile*), se è pieno (valore=0) egli viene bloccato. Ogni qualvolta che il consumatore rende disponibile uno spazio, tramite la primitiva signal(*spazio_disponibile*), egli risveglierà il produttore, se sospeso, oppure incrementerà il valore del contatore *spazio_disponibile*.
Analogamente, il consumatore verifica la disponibilità di un messaggio tramite la primitiva wait(*messaggio_disponibile*). Se il buffer è vuoto il consumatore viene bloccato. Ogni qualvolta che il produttore rende disponibile un messaggio, tramite la primitiva signal(*messaggio_disponibile*), egli risveglierà il consumatore, se sospeso, oppure incrementerà il valore del contatore *messaggio_disponibile*.
E' tuttavia anche necessario che che il produttore e il consumatore **non accedano mai alla stessa porzione di buffer contemporaneamente**.

Primitive send e receive

In un ambiente a memoria locale qualunque forma di interazione tra processi avviene attraverso la **comunicazione, cioè lo scambio di informazioni tra i processi sotto forma di messaggi**. Con il termine IPC (Inter-Process-Communication) si intende un meccanismo offerto dal nucleo del sistema operativo mediante il quale avviene la comunicazione tra processi. Lo strumento IPC più basilare è quello basato sulle primitive send(destinazione, messaggio) e receive(origine, messaggio).
La primitiva send() spedisce un messaggio ad una determinata destinazione e la receive() riceve un messaggio da una determinata origine (o da una qualunque origine).
Il concetto che sta alla base del loro utilizzo è la separazione degli spazi di indirizzamento dei singoli processi, tipico di un modello a memoria locale. Per comunicare i processi hanno bisogno di un canale di comunicazione; nel caso di sistemi privi di memoria comune il canale è realizzato tramite una qualche forma di collegamento fisico tra i processori sui quali operano i processi; nel caso di sistemi con memoria comune, il canale è realizzato da una porzione di memoria, gestita dal SO, dove vengono depositati e prelevati i messaggi.
Il formato tipico di un messaggio è diviso in 2 parti : **intestazione e corpo**. Nell'intestazione sono presenti l'identificazione del mittente e del destinatario, oltre che ad eventuali informazioni relative al tipo di messaggio, alla sua lunghezza ecc.

Soluzione al problema di comunicazione tra processi

In un sistema di scambio di messaggi i processi possono comunicare tra loro **direttamente o indirettamente**. Nel caso di comunicazione **diretta** è necessario **indicare esplicitamente nelle primitive send e receive i nomi dei processi** cui si vuole inviare/ricevere un messaggio :
send(P2,messaggio); | receive(P1, messaggio);
Questa comunicazione è di tipo **simmetrico**.
Se invece **non è possibile per il processo ricevente definire un nome** di mittente, in quanto più processi possono inviargli messaggi, la comunicazione è di tipo **asimmetrico** :
send(P2,messaggio); | receive(id, messaggio);

La comunicazione **indiretta** prevede che i messaggi **non siano inviati direttamente ai processi**, ma depositati e prelevati da una struttura dati detta **porta**, denominata anche mailbox.

Sincronizzazione tra processi comunicanti

Send :

- **asincrona** : consente al processo di proseguire la sua esecuzione immediatamente dopo l'esecuzione della send stessa e quindi l'invio del messaggio;
- **sincrona** : blocca il processo mittente fino al ricevimento del messaggio da parte del destinatario;
- **chiamata di procedura remota** : viene utilizzata quando il processo mittente chiede l'esecuzione di un servizio (procedura remota) al destinatario e i due processi operano su due processori collegati. Il processo mittente rimane in attesa fino a quando il servizio richiesto non è terminato e il risultato ricevuto.

Receive :

- **bloccante** : provoca la sospensione del processo che la esegue nell'ipotesi che non ci siano messaggi in attesa di essere serviti; all'arrivo del primo messaggio il processo viene risvegliato;
- **non bloccante** : consente la prosecuzione dell'esecuzione del processo anche in assenza di messaggi.

Nel caso di un modello client-server, sia send che receive sono bloccanti.

Risorse

Si identificano come ogni componente riusabile o meno, sia software che hardware, necessario al processo o al sistema.
Le risorse possono essere **suddivise in classi** e le risorse appartenenti alla stessa classe sono equivalenti, per questo le risorse di una classe vengono detta **istanze** di quest'ultima; **il numero di risorse di una classe viene detto molteplicità** del tipo di risorsa : ovvero il numero massimo di processi che la possono utilizzare contemporaneamente.
Quando un processo necessita di una risorsa richiede una generica istanza di quella specifica classe di risorse.

Deadlock

La situazione di blocco critico, o stallo (deadlock), si può verificare tra un gruppo di due o più processi **quando ciascuno processo possiede almeno una risorsa e fa richiesta per un'altra**. La richiesta non può essere soddisfatta perché la risorsa richiesta è trattenuta da un altro processo che è a sua volta bloccato in attesa della risorsa posseduta dal primo. Questa situazione di deadlock è dipendente dalla velocità relativa di esecuzione dei processi.
Affinché ci sia un deadlock è necessaria la presenza di quattro condizioni di **Coffman** :

1. **mutua esclusione** : le risorse coinvolte devono essere seriali, cioè ogni risorsa o è assegnata ad un solo processo o è libera;
2. **assenza di preemption** : le risorse coinvolte non possono essere prerilasciate, ovvero devono essere rilasciate volontariamente dai processi che le controllano solo quando hanno terminato di usarle;
3. **richieste bloccanti**;
4. **attesa circolare** : devono essere presenti almeno due processi e ciascuno di essi è in attesa di una risorsa occupata dall'altro.

I teorema sul grafo di Holt

Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili lo stato **è di deadlock** se e solo se il grafo di Holt contiene un **ciclo**.

II teorema sul grafo di Holt

Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili lo stato **NON è di deadlock** se e solo se il grafo di Holt è **completamente riducibile**, cioè se esiste una sequenza di passi di riduzione che elimina tutti gli archi del grafo.