**Parte II - Capitolo 6 - Sincronizzazione Dei Processi**

Un processo cooperante è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l'influenza. I processi cooperanti possono condividere direttamente uno spazio logico di indirizzi oppure condividere dati soltanto attraverso i file. L'accesso concorrente a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati. Per evitare queste situazioni occorre assicurare una forma di ***sincronizzazione*** dei processi, che permette a un solo processo per volta la modifica di una variabile.

Si consideri un sistema composto da n processi ciascuno avente un segmento di codice chiamato ***sezione critica*** o *regione critica* in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere un file e così via. Quando un processo è in esecuzione nella propria sezione critica, non si deve consentire a nessun altro processo di essere in esecuzione nella propria sezione critica. L'esecuzione delle sezioni critiche da parte dei processi è mutuamente esclusiva nel tempo. Il problema della sezione critica viene affrontato progettando un protocollo che i processi possono usare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica attraverso la sezione di codice detta ***sezione d'ingresso***. La sezione critica può essere seguita da una ***sezione d'uscita*** e la restante parte del codice di un processo viene detta ***sezione non critica***. Una soluzione al problema della sezione critica deve soddisfare i seguenti requisiti: mutua esclusione, progresso (nessun processo deve essere rallentato da un processo che non vuole entrare nella sezione critica) e attesa limitata (esiste un limite di numero di volte che un processo può entrare nella propria sezione critica). Due o più processi che dovessero aprire un file, ognuno per proprio conto e simultaneamente, potrebbero generare nel sistema una cosiddetta *race condition* legata ai necessari aggiornamenti della lista dei file aperti. Gli accessi al file sono simultanei e la modifica viene fatta solo dal primo processo che accede.

Illustriamo una soluzione software al problema della sezione critica. La ***soluzione di Peterson*** si applica a due processi che eseguono alternativamente la propria sezione critica e la sezione rimanente. I due processi devono condividere un intero che indica di chi sia il turno di accesso alla sezione critica e un booleano che indica se un processo dell'array è pronto per entrare nella sezione critica. Se i due processi tentano di accedere nello stesso momento solo un permane, mentre l'altro viene immediatamente sovrascritto. La soluzione risulta corretta in quanto sono preservate la mutua esclusione (un processo accede alla sua sezione critica solo se l'intero indica il suo turno), il requisito del progresso e dell'attesa limitata (un processo viene bloccato se non è pronto per entrare nella sua sezione critica attraverso il valore booleano).

In generale qualunque soluzione al problema richiede l'uso di un semplice strumento detto ***lock*** (*lucchetto*): per accedere alla propria sezione critica un processo deve acquisire il possesso di un lock, che restituirà al momento della sua uscita. Le funzionalità hardware possono rendere più facile il compito del programmatore e migliorare l'efficienza del sistema. In un sistema dotato di una singola CPU tale problema si potrebbe risolvere semplicemente se si potessero interdire le interruzioni mentre si modificano le variabili condivise; in questo modo si assicura un'esecuzione ordinata e senza possibilità di prelazione della corrente sequenza di istruzioni. Sfortunatamente questa soluzione non è sempre praticabile perchè la disabilitazione delle interruzioni nei sistemi multiprocessore può comportare sprechi di tempo con annessa diminuzione dell'efficienza. Per questo motivo molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola in memoria, oppure di scambiare il contenuto di due parole in memoria in modo *atomico* (cioè come un'unità non interrompibile).

Le varie soluzioni hardware basate sulle istruzioni atomiche complicano l'attività dei programmatori di applicazioni. Questa difficoltà può essere superata attraverso l'uso di uno strumento di sincronizzazione detto semaforo. Un ***semaforo*** (*semaphore*) è una variabile intera in cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: wait() e signal(). Tutte le modifiche al valore del semaforo contenute nelle operazioni atomiche si devono eseguire in modo invisibile. Si usa distinguere tra i ***semafori contatori***, il cui valore numerico è illimitato, e i ***semafori binari***, il cui valore è 0 o 1. In relazione a certi sistemi i semafori binari sono detti anche ***lock mutex*** (*mutex locks*) perchè fungono da lock che garantisce la mutua esclusione (*mut*ual *ex*clusion). I semafori sono utilizzabili per risolvere il problema della sezione critica con n processi, i quali condividono un semaforo mutex comune inizializzato a 1. I semafori contatore trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di esemplari disponibili. I processi che desiderano utilizzare un esemplare della risorsa invocano wait() sul semaforo, decrementando così il valore; i processi che restituiscono un esemplare della risorsa invocano signal() sul semaforo, incrementando il valore. Quando il semaforo vale 0 vengono allocati tutti gli esemplari della risorsa, e i processi che ne richiedono l'uso dovranno bloccarsi fino a quando il semaforo non ritorna positivo. Il principale svantaggio della definizione di semaforo è che richiede una condizione di ***attesa attiva*** (*busy waiting*). Mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrarvi si trova sempre nel ciclo del codice della sezione d'ingresso. Questo tipo di semaforo viene anche detto ***spinlock*** perchè i processi girano mentre altri attendono al semaforo, e comporta uno spreco dei cicli di CPU. I semafori spinlock hanno però il vantaggio di non richiedere un cambio di contesto nel caso in cui un processo sia fermo in attesa (sono molto utili quando i lock sono applicati a brevi intervalli di tempo). Per superare la necessità dell'attesa attiva, si possono modificare le definizioni delle operazioni wait() e signal(): quando un processo invoca il primo e trova che il valore del semaforo non è positivo, deve attendere, ma anzichè restare in attesa attiva può *bloccare* se stesso tramite l'operazione block(). L'operazione di bloccaggio pone il processo in una coda d'attesa associata al semaforo e cambia lo stato del processo nello stato d'attesa. Il controllo viene quindi trasferito allo scheduler della CPU che sceglie un altro processo pronto per l'esecuzione. Il processo sarà riavviato in seguito all'esecuzione di un'operazione di signal() da parte di un altro processo. Il processo si riavvia tramite un'operazione di wakeup(), che modifica lo stato del processo da attesa a pronto, quindi il processo viene inserito nella coda dei processi pronti. La realizzazione di un semaforo con coda d'attesa può condurre a situazioni in cui ciascun processo di un insieme di processi attende indefinitamente un evento. Quando si verifica una situazione di questo tipo si dice che i processi sono in ***stallo*** (*deadlocked*). Un insieme di processi è in stallo se ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme. Un'altra questione connessa alle situazioni di stallo è quella dell'***attesa indefinita*** (*starvation*).

- Problemi tipici di sincronizzazione (pag. 229)

Produttori e consumatori con memoria limitata

Problema dei lettori-scrittori

Problema dei cinque filosofi

Benchè i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare; inoltre l'uso dei semafori non esclude la possibilità che si verifichi un qualche errore di sincronizzazione. Per rimediare a queste mancanze, i ricercatori hanno sviluppato un costrutto fondamentale di sincronizzazione ad alto livello: il ***monitor***. Il tipo monitor è un tipo di dato astratto che presenta un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono cotraddistinte dalla mutua esclusione. Il tipo monitor contiene anche la dichiarazione delle variabili i cui valori definiscono lo stato di un'istanza del tipo, oltre ai corpi delle procedure o delle funzioni che operano su tali variabili. Una procedura definita all'interno del monitor ha accesso unicamente alle variabili dichiarate localmente e ai relativi parametri formali; in modo analogo, alle variabili locali di un monitor possono accedere solo le procedure locali. Il costrutto monitor assicura che all'interno del monitor possa essere attivo un solo processo alla volta, sicchè non si deve codificare esplicitamente il vincolo di mutua esclusione. Questa definizione di monito non è abbastanza potente per esprimere alcuni schemi di sincronizzazione. Abbiamo quindi bisogno di una o più variabili condizionali (condition x, y;) che possono eseguire solo due operazioni: x.wait() implica che il processo che la invoca rimanga sospeso finchè un altro processo non invochi l'operazione x.signal() che risveglia il processo sospeso. Se non esistono processi sospesi questa operazione non viene eseguita.

Soluzione del problema dei cinque filosofi per mezzo di monitor (pag. 237)

Realizzazione di un monitor per mezzo di semafori (pag. 239)

Ripresa dei processi all'interno di un monitor (pag. 240)

- Esempi di sincronizzazione (pag. 241)

Sincronizzazione in Solaris

Sincronizzazione in Windows XP

Sincronizzazione dei processi Linux

Sincronizzazione in Pthreads

La mutua esclusione nelle sezioni critiche assicura che siano eseguite in modo atomico: se due sezioni critiche sono eseguite in modo concorrente, il risultato che si ottiene coincide esattamente con quello che si otterrebbe dall'esecuzione sequenziale delle due sezioni critiche in qualsiasi ordine. Con particolare attenzione alle basi di dati, i sistemi operativi si possono considerare sistemi per la manipolazione di dati e, quindi, possono trarre vantaggio da tecniche e modelli ottenuti nella ricerca sulle basi di dati. Un insieme di istruzioni che esegue una singola funzione logica prende il nome di ***transazione***. L'operazione commit indica che la transazione è terminata con successo, mentre l'operazione abort significa che la transazione è fallita a causa di qualche errore logico o di un guasto del sistema. Al fine di conservare l'atomicità, la terminazione anomala di una transazione non deve produrre alcun effetto sullo stato dei dati che questa ha già modificato. Quindi, è necessario ripristinare lo stato dei dati adoperati dalla transazione fallita, riportandolo a quello che li caratterizzava appena prima dell'inizio della transazione (***roll back***). Un modo per assicurare l'atomicità è registrare in memorie stabili le informazioni che descrivono tutte le modifiche che la transazione ha apportato ai dati a cui ha avuto accesso. Il metodo più largamente usato è quello della ***registrazione con scrittura anticipata*** (*write-ahead-logging*): il sistema mantiene, nella memoria stabile, una struttura dati chiamata ***log***, in cui ciascun elemento descrive una singola operazione write eseguita dalla transazione ed è composto dai seguenti campi: nome della transazione, nome del dato modificato, valore precedente, nuovo valore. Mediante l'uso dei log il sistema può gestire qualsiasi malfunzionamento, purchè non sia una perdita delle informazioni contenute nella memoria non volatile. L'algoritmo di ripristino impiega le procedure di undo(T) per ripristinare il precedente valore di tutti i dati modificati dalla transazione T, e di redo(T) per assegnare il nuovo valore a tutti i dati modificati dalla transazione T. Se si verifica un guasto nel sistema, il ripristino di uno stato corretto comporta la consultazione del log per determinare quali transazioni annullare e quali ripetere: una transazione deve essere annullata se il log contiene l'elemento start ma non l'elemento commit; una transazione deve essere ripetuta se il log contiene sia l'elemento start che l'elemento commit. Quando si verifica un guasto nel sistema bisogna consultare il log per il ripristino del sistema. Questo procedimento ha due inconvenienti: la ricerca può richiedere un tempo piuttosto lungo e la maggior parte delle transazioni deve essere ripetuta, quindi il ripristino richiedere ancora più tempo. Per ridurre questo genere di sprechi di introduce il concetto di ***punto di verifica*** (*checkpoint*). Durante l'esecuzione il sistema esegue la registrazione con scrittura anticipata e registrazioni periodiche secondo le seguenti azioni: registrazione nella memoria stabile di tutti gli elementi del log, registrazione nella memoria stabile di tutti i dati modificati, registrazione dell'elemento checkpoint nel log residente nella memoria stabile. Il checkpoint servirà come punto di partenza per la verifica delle transazioni da ripetere o da annullare. Abbiamo finora considerato un ambiente in cui sia eseguibile solo una transazione per volta. Passiamo ora al caso in cui transazioni multiple siano attive concorrentemente. Poichè le transazioni sono atomiche, il risultato dell'esecuzione concorrente deve essere equivalente a quello che si otterrebbe eseguendo le transazioni in una sequenza arbitraria. Questa caratteristica di ***serializzabilità*** si può rispettare semplicemente eseguendo ciascuna transazione all'interno di una sezione critica. Una sequenza di esecuzione in cui ciascuna transazione è eseguita in modo atomico è detta ***sequenza d'esecuzione seriale***. Essa è composta da una sequenza di istruzioni appartenenti a transazioni diverse, caratterizzata dal fatto che tutte le istruzioni appartenenti a una transazione sono raggruppate. Se si consente a due transazioni di sovrapporre le proprie esecuzioni, il risultato non è più seriale e quindi scorretto (***operazioni conflittuali***). Se una sequenza d'esecuzione si può trasformare in una sequenza d'esecuzione seriale con una serie di scambi tra operazioni non conflittuali, si dice che essa è in ***conflitto serializzabile***. Uno dei metodi che si usano per garantire la serializzabilità consiste nell'associare un lock a ciascun dato, e richiedere che ogni transazione rispetti il ***protocollo per la gestione dei lock*** (*locking protocol*), che governa l'acquisizione e il rilascio dei lock. Si può applicare un lock a un dato in due modi: condiviso, cioè la transazione può leggere, ma non scrivere nell'elemento, e esclusivo, cioè la transazione può leggere e scrivere nell'elemento. Un lock potrebbe non essere immediatamente rilasciato dopo il suo ultimo accesso, quindi la serializzabilità non è sempre garantita. Un protocollo che assicura la serializzabilità è il ***protocollo per la gestione dei lock a due fasi***, che esige che ogni transazione richieda l'esecuzione delle operazioni di lock e di rilascio (*unlock*) in due fasi distinte: fase di crescita (una transazione può ottenere nuovi lock sui dati, ma non rilasciarne alcuno in suo possesso) e fase di riduzione (una transazione può rilasciare lock sui dati di cui è in possesso, ma non ottenerne di nuovi). Questo protocollo garantisce la serializzabilità, ma non elimina la possibilità di situazioni di stallo. Un altro metodo per determinare l'ordine di serializzabilità consiste nella scelta anticipata di un ordinamento delle transazioni. Il metodo più comunemente adottato consiste nell'usare uno schema con ***ordinamento a marche temporali*** (*timestamp ordering*). A ogni transazione viene associata una marca temporale (*timestamp*) unica. Per assegnare le marche temporali esistono due metodi: adoperare come marca temporale il valore dell'orologio di sistema oppure adoperare come marca temporale un contatore logico. In questo modo le operazioni vengono serializzate in ordine di marca temporale (Vedere sul libro e sulle slide).