Lez sistemi

Andrea Comar

6 November 2024

Part I
PROCESSI COOPERANTI

1 introduzione

1.1 IPC

meccanismi che permettono ai processi di comunicare tra loo e si fa riferimento a tutte le tecniche utilizzate dai rocessi cooperanti per scambiarsi e condividere nformazioni.

problematiche: processi cooperanti devono accedere a strutture dati condivise, ma bisogna stare attenti che non portino a inconsistenze. Regolare accesso di conseguenza. il lavoro di diversi processi dovrà essere sincronizzato

problema del produttore-consumatore abbiamo due processi, uno **produttore**, uno **consumatore**. Il produttore produce dei dati, inserisce i dati nel **buffer** e l consumatore li estrae dal buffer. Il buffer può ssere implementato come array circolare, con due dati, un puntatore IN dove produttore estrae dati, mentre avremo puntatore OUT per consumatore.

IN punta alla locazione successiva

OUT punta alla prossima locazone da cui costruttore estrarrà il prossimo dato.

ovviamente i processi andranno sincronizzati

caso buffer PIENO: poichè il produttore li inserisce in modo circolare, il produttore finito array ricomincierà dall'inizio. Nota: importante che IN non scavalchi mai OUT. Dato N dimensione buffer, se counter = N il buffer è pieno. Se buffer è pieno, il produttore non può scrivere nuovi dati finché il consumatore non ha letto almeno una cella (la prima)

caso buffer VUOTO(counter = 0, counter rappresenta dati utili da leggere):

il consumatore ha letto tutti i dati utili, consumatore deve aspettare la scrittura da parte del produttore.

rappresenta un caso paradigmatico di molti problemi, per esempio caso stampante, oppure compilatore in assembler

da slide possiamo vedere due esempi di linguaggio ad alto livello per gestire il problema. Il codice nasconde diverse difficoltà: problema dell'esecuzione parallela è l'accesso alle variabili condivise, ovvero counter il buffer. Counter è condivisa tra i due processi. L'esecuzione parallela può portare a inconsistenze su counter.

In codice assembler:
processo produttore:
reg_1 := counter;
$reg_1 = reg_1 + 1$;
$counter = reg1$;
codice consumatore
$reg_2 = counter$;
$reg_2 = reg_2 + 1$;
$counter = reg_2$;

possiamo considerare valore iniziale counter = 5
immaginiamo che processo p aumenti di 1 il variabile counter, mentre in parallelo processo c consumatore va in letture, quindi decrementa counter. In teoria alla fine delle due parti di codice dovrei avere di nuovo 5 (+1,-1) ma nel caso di linguaggio assembler potrei trovarmi della condizione di avere o 4 o 6. Perché counter = counter + 1 non è atomica, cioè sono più istruzione assembler. Stesso discorso per counter = counter - 1.
Il rischio è esecuzione passata da p a c dopo solo un'istruzione assembler, in modo NON ordinato, magari prima del cambio del valore di counter.
Problema appunto dal linguaggio ad alto livello. Dunque bisogna trovare delle soluzioni a questo problema, race conditions sono MOLTO pericolose, rischio di danni di grosso livello. RACE CONDITIONS sono difficili da individuare perché dipendono dallo scheduler e dalla politica, difficili da individuare perché non dipendono dalla parte di codice ad alto livello.

sezione critica: parte di codice in cui un processo accede a dati condivisi. Per evitare race condition dobbiamo fare in modo che la sezione critica siano in maniera atomica, cioè fare in modo che le sezioni critiche di più processi NON si sovrappongano. I processi possono sovrapporsi finché eseguono sezioni NON critiche.

Per risolvere si può utilizzare del codice di controllo che viene eseguito prima dell'accesso alla sezione critica **entry section**. Se viene dato accesso alla sezione critica, altri processi non possono interferire. POi codice di controllo in fase di

uscita **exit section**. Accesso ovviamente dato solo se non ci sono sezioni critiche in corso. La slide 207 mostra proprio questo.
pertanto è necessario scrivere: **protocollo di entrata e uscita**. Il nome di questo metodo è detto **mutua esclusione.**

Tuttavia una buona soluzione richiedere altri due criteri importanti:
PROGRESSO: nessun processo che sta eseguendo codice NON critico può bloccare processi che desiderano entrare in sezione critica
ATTESA LIMITATA: se un processo intende entrare in sezione critica, deve esserci un limite al numero di possibili scavalcamenti da parte di altri processi. Se p entra nella sezione critica, sa che ha atteso al massimo x accessi critici da parte di altri processi. No Starvation.

Si suppone che il processo venga eseguito a una velocità non nulla, ma non si suppone nulla sulla velocità relativa.

1.2 Soluzione HW: controllo Interrput

Nel momento in cui un processo chiede di entrare nella sezione critica, questo sospende tutti gli interrupt hardware. Questo assicura assenza di context switch (non ci sono interrupt).
Soluzione semplice ma pericolosa, il processo potrebbe non riabilitare più interrupt, non abbiamo garanzie sul corretto comportamento del programma, potrebbe allungare di molto i tempi di latenza, e non si estende a macchine multiprocessore.

ottimo per eseguire codice molto breve e assolutamente affidabile (es.livello kernel), quindi utilizzabile ma soltanto in casi molto particolari e limitati

1.3 soluzioni software - errore

per semplicità ipotizziamo di avere solo due processip_0 e P_1
Struttura del processo P_i

(slide per codice)
tentativo sbagliato: immaginiamo di far condividere una variabile, occupato. Se occupato vale a 0 nessuno sta eseguendo sezione critica, vale 1 se sezione critica è in esecuzione.

Il programma P_i verifica la condizione di occupato, se vale 1 continua a ciclare in un while senza body (continuo a verificare).
Appena while è a 0, riporto occupato a 1 ed eseguo la sezione critica, poi riporto con 0.

è una soluzione al problema della soluzione critica? mi garantisce la mutua

esclusione?

Per capirlo deve immaginare di eseguire i processi in parallelo e devo verificare che entrambi non si trovino contemporaneamente nella propria sezione critica. Questo non viene verificato, perché P_0 entra nel while trova occupato = 0; ma in questo momento potrebbe essere bloccato il processo e scheduler potrebbe passarlo a P_1 che troverebbe sempre occupato = 0, per poi ripassare a P_0 che ripartirebbe da sezione critica. NON garantisce mutua esclusione.

1.4 Alternanza Stretta

Facciamo sì che i due processi condividano una variabile TURN che serve a gestire i turni.
se TURN = 0, il turno è P_0
se TURN = 1, il turno è di P_1
Nella entry valuto se TURN corrisponde al mio indice.
Se TURN dice che è il mio turno, allora entro in sezione critica e poi cambio il valore di TURN.
Pertanto garantisce la tua esclusione
garantisce anche ATTESA LIMITATA, visto che cambia indice e quindi so quanti processi prima di me
manca però condizione di PROGRESSO, perché se impongo $i \to j$ alla fine della mia sezione critica, se il processo j non vuole entrare nella sua sezione critica, allora il processo $j + 1$ (oppure i) non può accedere nella sezione critica perché nessuno cambia più il valore di turn.
è un processo di busy wait, il processo testa ripetutamente che TURN indichi il valore corretto (attesa attiva o building waiting), consumo di CPU inutile, perché continua a testare. Sarebbe meglio metterlo in stato di WAIT e risvegliarlo una volta che TURN assuma il suo valore.
metodo utile per situazioni in cui lo SPIN LOCk (busy witing) è limitato, quindi conosco bene le condizioni temporali

1.5 Algoritmo di Peterson (1981)

Algoritmo molto famoso e importante, utilizzato fino ai nostri giorni. SI basa sull'idea che ci siano dei turni, quindi TURN c'è, ma è accompagnata da una variabile interna ai programmi che segnali la voglia di entrare in sezione critica. Esiste pertanto un vettore INTERESTED che raccoglie tutti i valori di voglis. Se P_1 vuole entrare in sezione critica cambia INTERESTED[1] in 1. Se ci sono due programmi interessati a entrare vince in base alla variabile TURN, se invece solo uno è interessato allora può entrare lo stesso pur se TURN indica altro visto che è l'unico interessato.
Codice nella slide 214

Inizialmente nessuno dei due vuole entrare nella sezione critica

P_i while(TRUE){
entry-region();
sezione critica;
leave-region(
sezione-non-critica
}

Nota, in questo codice se entrambi i processi sono interessati la variabile turn stabilisce quali dei due entra. Nel codice slide entra chi ha turn NON settato nella proprio valore (se TURN == PROCESS rimango nel while). Nella leave region pongo a FALSE il mio interesse INTERESTED[i] = FALSE.

Entrambi fanno INTERESTED in true, ma turn avrà soltanto 1 valore, per cui mutua esclusione verificata.
Attesa limitata garantita: chi vince entra, ma l'altro processo non verrà scavalcato una seconda volta (più il prossimo processo porrà interested a false, e nel caso dovesse mostrare interesse nel pre while porrà TURN == PROCESS, permettendo all'altro di scavalcare).
Da analizzare bene il codice
Nota: è ancora basato su SPIN LOCK, quindi c'è attesa attivs.

1.6 Algoritmo del fornaio

generalizza algoritmo di Peterson per n processi.
Ogni processo riceve un numero prima di entrare nella sezione critica. Numero più basso vince
assegnazione è sezione critica
un eventuale conflitto con stessi numeri può essere risolto mediante PID (PID più basso vince)

1.7 Istruzioni di Test& Set

istruzione speciale presente a livello di linguaggio assembler (Test-and-Set-Lock). Permette di testare e settare il contenuto di una parola(word) in un'unica istruzione atomica, NON INTERROMPIBILE, ovviamente implementata in assembler. (es BTC, BTR, BTS su intel).

TSL RX,LOCK

Così risolvo problema sezioni critiche. Codice slide 218, Peterson con aggiunta delle Istruzioni TSL.

loop