Sistemi20241108

Andrea Comar

8 November 2024

1 Ripasso

La sezione critica va protetta, evitando che vengano eseguite contemporaneamente. Tra i criteri ne troviamo quindi 3:

- attesa limitata (no starvation)
- mutua esclusione (criterio principe)
- progresso (non bloccato da processi non in sezione critica

Abbiamo visto alcuni procedimenti (algoritmo di Peterson, codice atomico implementato in assembler

2 Semafori

2.1 Busy Wait

un processo che attende di accedere in sez. critica attende di in maniera attiva controllando continuamente se è possibile entrare in sez critica (magari controllando un parametro). Attese attive possono portare a problema di inversione di priorità, ovvero un processo ad alta priorità continuerà a far richiesta di sez. cricitica, e lo scheduler continuerà a dargli controllo cpu (loop)
soluzioni possibili:

- sleep(): system call processo in cui si autosospende
- wakeup(pid): system call per risveglio

Tuttavia c'è un problema
(slide 220)
problema del produttore e del consumatore con soluzione di primitive sleep e wakeup. Utilizzate per gestire sincronizzazione dei due processi, in particolare nei casi critici buffer pieno e beffer vuoto.
buffer vuoto: risveglio produttore
buffer pieno: risveglio consumatore
(... descrizione dei due cicli produttore e consumatore)

istruzione su variabile count andrebbero protette
problema: segnale di sveglia con wakeup, se consumer è andato in sleep, ma se consumatore non è in stato di sleep, andrà perduto e non sortirà effetto.

ipotizziamo che produttore riempe buffer, trovando counter == n;
se scheduler passa al consumatore cpu prima che produttore vada in sleep, questo inizia a leggere e manda segnali wake al produttore, va avanti finchè non svuota buffer, manda wake e va a ninna. Appena scheduler ripassa cpu al produttore, questo riprende da dove era stato interrotto, quindi entra in sleep perché pensa che counter sia ancora uguale a n. Pertanto soluzione non va bene.

2.2 concetto di semaforo

semaforo s è una struttura dati che consiste di una variabile intera più due operazioni **up**, **down**che mantiene il numero di segnali inviati ma non ancora ricevuti.
up(S): incrementa s
down(S): se S = 0, il processo va in sleep, se invece S è maggiore di 0 allora ne consuma uno (... controllare meglio)

NOTA: ovviamente up e down sono atomici e non interrompibili, eseguiti in mutua esclusione.
slide (223)
come utilizzo i semafori per risolvere?
più processi accedono a struttura dati condivisa

codice P_i while(TRUE)
down(mutex)
sezione critica
up(mutex)
sezione non critica

semaforo binario (0 o 1) MUTEX, quando MUTEX==1, sezione libera, MUTEX==0 sezione occupata.
se trovo mutex a 0, processo va in stato di waiting
se trova mutex a 1, down porta mutex a 0, processo entra. Alla fine della sez.crit. up riporta semaforo a 1.

ovviamente funziona perché up, down sono atomiche.
Garantisce mutua esclusione, il primo che effettua la down blocca l'altro.
attesa limitata viene garantita se semaforo viene implementato correttamente, i processi quando vanno in sleep si mettono in coda, in modo ordinato.
progresso verificato, perché processi non interessati non cambiano valore del semaforo e non interferiscono.

I semafori possono essere utilizzati per sincronizzare ordine tra più processi,

esempio nella slide 224

2.3 Produttore-consumatore con semaforo

Abbiamo più problematiche da gestire:

- la sezione critica: condivisione buffer, inserzione produttore e remove, vanno protette (???) con MUTEX
- sincronizzazione tra produttore e consumatore

Utilizzo di tre semafori, MUTEX, EMPTY E FULL.
EMPTY = quante celle sono ancora disponibili nel buffer, inizialmente vale n, a 0 produttore si blocca
FULL = quante celle son piene, inizialmente vale 0, il consumatore se trova FULL vuoto si blocca

il produttore produce item, controlla che il buffer sia non pieno, quindi va a fare down su empty. Se empty va a 0, produttore si blocca per down. Altrimenti down decrementa e produttore testa MUTEX, con down su MUTEX, se mutex è libera procede e inserisce item. Se mutex non è libera va in stato di waiting. Inserito item il prod. fa up su MUTEX e poi di seguito fa up su FULL.

Il consumatore vuole estrarre.
fa una down sul semaforo FULL
Se full maggiore di 0, decremento FULL e procedo
se full == 0, allora down blocca consumatore. Verrà risvegliato da up del produttore su FULL
Procedo, sto per entrare in sez. critica, quindi down su MUTEX
se libero, entro, estraggo, rimodifico MUTEX
e faccio up su EMPTY (potrebbe risvegliare produttore)

il codice è corretto, ci risparmiamo la dimostrazione della correttezza.

2.4 implementazione dei semafori

I semafori sono la struttura dati costituita da variabile e dalle due operazioni che permettono l'accesso alla variabile, rigorosamente atomiche. Come si possono implementare le operazioni?
Le due operazioni up e down sono a loro volta sezioni critiche che vanno eseguite in mutua esclusioni, protette con meccanismi spiegati nelle lezioni precedenti. (peterson, assembler, controllo interrupt).
Semaforo visualizzabile anche come variabile e rispettiva coda dei processi in attesa.
per implementare, assumiamo presenza di sleep e wakeup.

slide 227 c'è lo pseudocodice di down e up.

DOWN
inizialmente decremento S. Se S ¡ 0 vado negativo ed entro sleep.Se S non negativo, down termina.
UP(S)
Incremento S. Se S è minore o uguale a 0, tolgo un processo dalla coda e sveglio un processo.Altrimenti nulla

questo codice va protetto, perché sezione critica. come?

- disabilitazione interrupt (occhio)
- test and set
- Peterson

Abbiamo introdotto semaforo per evitare spin lock, ma anche loro sono sezione critica, torniamo al problema precedente con soluzioni precedenti.
Qual è la differenza?
Up e down sono codici BEN DEFINITI e BREVI, quindi l'attesa attiva è veramente limitata. In un certo senso abbiamo spostato il problema, ma l'attesa attiva avviene solo su codice breve.

(slide 229)
MUTEX con codice assembler, down = MUTEX-lOCK, up = MUTEX-UNLOCK
nota uso di 0 e 1 è invertito rispetto al normale utilizzo nel contesto dei semafori.
Esempio utilizzato per gestire thread a livello utente, si capisce dalla call del thread-yield.

2.5 memoria condivisa?

Necessità di condividere spazi di memoria per implementare queste strutture.
tra processi diversi serve far condividere segmenti di memoria, esempio in UNIX tramite shared memory.

2.6 Deadlock con semafori

Se non implementati correttamente possono facilmente creare problemi, tra cui deadlock. Esempio nella slide 231.
P_0 procede soltanto se entrambi semafori a 1; analogo per P_1. Il problema è che non essendo eseguiti in ordine, se P_0 blocca S e P_1 blocca Q, entrambi i processi si fermano.
Programmare con i semafori è difficile

3 Monitor

Il monitor è un tipo di costrutto alternativo ai semafori. Fornisce funzione di mutua esclusione. Il monitor è un costrutto molto più ad alto livello, gestito a livello di compilazione (linguaggio), senza dover obbligare a gestire la mutua esclusione su sez critica.
Il monitor è un costrutto molto più flessibile.

Dentro un monitor raccolgo tutta una seria di procedure che vengono eseguita in mutua esclusione. Se un programma chiama una procedura del monitor, siamo sicuri che non verrà eseguita un'altra in concomitanza.
Questo è garantito dal compilatore che inserisce automaticamente un protocollo di ingresso e uscita sez. critica.
Fornisce la possibilità di usare **variabili condition**, utili per la sincroonizzazione tra processi. SUlle variabili condition posso usare le primitive wait e signal. UN programma ch esegue wait(c) si blocca sulla variabile c. Un signal(c) risveglia uno dei processi che era in sleep su un processo in wait su c.
Quale dei due processi deve però continuare a eseguire il codice di monitor? (il signal risveglia un altro programma che vuole agire in monitor)
La risposta dipende dalla tipologia di monitor, implementazioni diverse portano a soluzioni diverse, come in slide 233.

3.1 Produttore-consumatore con monitor

Il codice si trova nella slide 234.

Il produttore vuole inserire item nel buffer condiviso. Tuttavia questa volta il buffer è implementato con struttura dati con monitor, insert e altre opzioni di accesso sono sezione critica.
Essendo monitor il compilatore ci garantisce che l'insert viene eseguito in mutua esclusione.
Ritorno delle variabili FULL e EMPTY, stavolta condition. Stessa soluzione rispetto sleep e wait. Anche se viene bloccato il processo durante insert, un altro processo non può eseguire remove perché garantito dal monitor.

Essendo costrutti e non funzione di libreria, bisogna modificare i compilatori in caso
problema. necessità di memoria condivisa, quindi in caso di sistemi distribuiti non va bene.

4 Passaggio di messaggi

Tecnica utilizzata nei sistemi distribuiti, tecnica più generale, ottima per lo scambio di informazioni. Può essere utilizzata anche a livello locale su una sin-

gola macchina.
si basa su due primitive, send e receive.
SEND: spedisce, non bloccante
RECEIVE: riceve, opzione bloccante

PROBLEMATICHE:
affidabilità dei canali (esempio le reti), pertanto bisogna ovviare alla perdita di dati (acknowledgmet e timestamping, protocollo abbastanza standard)
autenticazione, garanzia del mittente
sicurezza e rischio intercettazione
efficienza, più lenta rispetto a memoria condivisa e semafori.