Sistemi20241113

Andrea Comar

November 2024

1 Introduction

1.1 passaggio di messaggi

i nostri processi girano su un sistema distribuito in cui non è possibile condividere strutture dati. Allora si usa il **passaggi di messaggi**.
SI basa sull'uso di due primitive, SEND e RECEIVE che sono chiamate di libreria o sistema.

SEND non è una chiamata bloccante
RECEIVE è bloccante così si mette in attesa del messaggio.
Ci sono delle problematiche: autenticazioni, efficienza, sicurezza. Per quanto riguarda l'efficienza ovviamente la memoria condivisa è più rapido, in quanto prevede minori system call.
La comunicazione può essere di due tipi:
ASINCRONA: send non è bloccante, i messaggi spediti ma non ancora ricevuti vengono registrati in un buffer MAILBOX kernel o librerie spazio utente, mentre ricevente preleva dalla MAILBOX. si blocca con mailbox vuota, problema analogo produttore consumatore
SINCRONA: SEND e RECEIVE sono entrambe bloccanti, finchè controparte non fa duale.
(slode 239) codice visto con problema del produttore e consumatore.
Consumatore esegue ciclo for con messsaggi vuoti in modo tale che il produttore possa basarsi su item vuoti per dimensionare gli item che invierà.
Successivamente produttore invia mano a mano i messaggi e fa SEND al consumatore il quale fa RECEIVE aspettando messaggio dal produttore.

1.2 barriere

Altro metodo per sincronizzare, soprattutto calcolo parallelo memoria condivisa. Il compito di ciascun processo è diviso in vari step. Tutti i processi devono aver eseguito lo step p_i per passare allo step successivo p_{i+1}. Ogni processo che raggiunge fine computazione step chiama la BARRIER va in sleep. ultimo processo che chiama barrier sveglia tutti.

2 Problemi classici del mondo S.O

2.1 problema dei filosofi a cena

Serve per rappresentare in maniera astratta la competezioni di un certo numero di processi (filosofi) per un numero di risorse limitate(piatti). I filosofi sono seduti in tondo con un piatto di fronte. Le forchette dividono i piatti e quindi sono condivise.
Il filoso pensa, quando ha fame prende due forchette per poter mangiare
pensare non richiede risorse, mangiare invece sì perché richiede forchette.
Vogliamo evitare:

- starvation: prima o poi un filosofo mangia (no scavalcamenti infiniti)
- deadlock: non si verificano blocchi (esempio: tutti i filosofi prendono una forchetta con la mano destra, non ci sono più forchette per le mani sinsitre e nessuno vuole rilasciarla).

slide 242 : codice NON soluzione
TAKE FORK (i), forchetta sx, TAKE FORK (i+1) forchetta dx
nota: indice sono in modulo N, l'ultimo filosofo deve prendere la forchetta 0 a dx!
filosofo prende sinistra, prende la destra, mangia, posa la sinistra, posa la destra.

TENTATIVO di correzione: controllo disponiilità forchetta dx, possibilità di starvation, no deadlock.

Per evitare starvation, posso utilizzare un ritardo casuale prima di ripetizione del tentativo. Tuttavia riduce ma non limita.

2.2 filosofi a cena, risoluzione

SEMAFORO MUTEX
Per proteggere la risoluzione critica è l'utilizzo di **semafori** per proteggere la sezione critica. Primo pensiero è considerare le due TAKE_FORK come sezione critica. Oppure sezione critica tutta la parte di codice da prima TAKE FORK a ultima PUT_FORK.
Evita starvation ma riduce al minimo parallelismo. Nessun altro filoso può eseguire parte di codice critica, solo un filosofo alla volta mangia. In realtà parte intera inferiore n mezzi.
TRE STATI (THINKING, EATING, HUNGRY)
Utilizzo di tre stati: filosofo può entrare in eating solo se lui era in hungry e i vicini NON sono in hungry. Introduzione del semaforo MUTEX oer difendere il vettore STATE, che è condiviso da tra filosofi. In più array di semafori, uno per ogni filosofo. Ogni filosofo ha un ciclo in cui. (..)
codice commentato. Blocca deadlock, starvation e garantisce massimo parallelismo. MUTEX protegge variabil state impedendo che più utenti siano attivi contemporanemante.

3 Lettori-scrittori

un insieme di dati condivisi tra due tipologie di processi ovvero lettori e scrittori. lettori possono accedere contemporanemante alla struttura dati ma invece gli scrittori devono accedere in modom esclusivo. Lettori non hanno necessità di mutua esclusione perchè non andando a modificare la struttura dati non creano problemi di interferenza.
Cambia il discorso se nella struttura dati sta operando uno scrittore, oerché possono esserci fenomeni di interferenza.
esempi di processi lettori e scrittori nelle slide.
abbiamo bisogno di variabile RC condivisa, semaforo MUTEX (a valori 0,1) per variabile RC e semaforo db per proteggere accesso al database. Se base dati utilizzata da lettore, un lettore può accedere.

codice lettore:
ciclo infito in cui tenta di accedere in lettura alla base dati
down su mutex
accedo rc e pongo rc
variabile reader counter (RC), condivisa e quindi protetta
se rc è uguale a 1 è unico lettore, allora rilascia mutex,
accede ai dati in letture
fa down su mutex
modifica rc, se rc = 0 allora fa UP su db (rilascia base dati)
NOTA: solo ultimo lettore rilascia la base di dati.

PROCESSO SCRITTORE
ciclo infito in cui scrittore vuole accedere in scrittura.
prepara i dati (sezione privata)
down su db, in modo da verificare che non ci sia accesso da parte di lettori e scrittori
in caso accede, scrive, rilascia, ecc
Il lettore se trova altri lettori non fa down su db, questa differenza
Solo primo lettore fa down su base di dati.
Presenta una problematica indesiderata: starvation da parte di un processo scrittore perché continuamente scavalcato da parte di processi lettori. soluzione NON FAIR.
Come posso modificarla? Il processo scrittore se vuole accedere allora nessun altro processo lettore può accedere scavalcandolo

3.1 problema del barbiere che dorme

Un barbiere, una sedia da barbiere e n sedie per l'attesa. Metafora che rappresenta una situazione con un server e una serie di client. Se non ci sono clienti il barbiere (server) dorme. Se arriva un cliente il barbiere si sveglia. Se la sedia da barbiere e libera il cliente viene fatto sedere, altrimenti messo in sedia di attesa. Se sedie di attesa piene, cliente se ne va.

Uso di 3 semafori: CUSTOMERS (numero di clienti in attesa, assieme a una variabile WAITING), BARBERS (barbiere in attesa di clienti, in questo caso o 0 o 1), MUTEX (usato per proteggere variabile WAITING).
Ogni barbiere usa una procedure che lo blocca se non ci sono clienti.
codice del processo barber:
down su CUSTOMERS, se non ce ne sono allora si blocca
se ci sono CUSTOMERS, la down abbassa il valore, down su mutex, waiting viene decrementata e viene fatta up su barber.
rilascia WAITING con up su mutex.
Codice del CUSTOMERS
controlla waiting, quindi prima down su mutex per agire su waiting. Se waiting $>$ CHAIRS ...