

Comunicazione tra processi

Esistono varie tecniche

Più venire (i messaggi) ne c'è un collegamento

I/O tra processi (uso di Pipe)

• La comunicazione è detta Inter Process Communication (IPC)

Di solito ci sono 3 canali separati nella comunicazione: Target - Origin - Elenco

Pipe: strumento di comunicazione minimale, fa da buffer, è obbligatorio limita poiché non costituisce una comunicazione come flusso di dati

Problemi della comunicazione:

- come combinare dati
- accavallamento di operazioni
- sincronizzare le operazioni

Tecniche di memoria concorrente

Due giochi (o giri) contribuiscono alla struttura dati, non possono essere come problemi che le CORSE CRITICHE (legate al parallelismo)

Esempio: rendimento sottocorrente

Bisogna realizzare (bene)

le operazioni dei processi
non fatti combinarsi.

A	$c=0$	B
c++	\downarrow	c++
c=2		illegalmente
	Però aperti i problemi potrebbero sopravvivere a	
	$c=1$	

Le corse critiche possono verificarsi anche nel codice del Kernel, fatto che anche qui ci sono flussi paralleli

Soluzione: mettere esclusione nell'accesso dei dati combinarsi

Dobbiamo individuare le SEZIONI CRITICHE.

- Ecco cosa restituisce alle strutture dati in combinarsi.
- In un processo poniamo identificare pezzi di codice che agiscono su dati combinarsi con un altro processo.

All'inizio e alla fine della sezione critica ci sono dei meccanismi che entrano che un altro processo "può entrare", ciò viene ritardato finché il 1° processo non esce dalla sezione critica.

Quindi se A è in sezione critica, B può proseguire a fatto che non entra in una zona critica

Così l'importante è che non entrambi siano in zone critiche (contemporaneamente, e nello stesso istante) 

Buona soluzione ha le caratteristiche

- mutua esclusione nell'accesso alle zone critiche
 - Non fanno assunzioni sulla velocità di esecuzione e/o numero di CPU
 - Un processo che è fuori da una zona critica non può bloccare un altro
 - Nessun processo può aspettare all'infinito il permesso di entrare in zona critica

NB: Se A e B condividono la stessa struttura dati
la nostra escluzione non è altro se non, la
nozione critica è relativa a una certa
struttura dati.

REALIZZARE LA MUTUA ESCLUSIONE

fa illo: Dizdilte di interno (gli come fatti in k-mole)

E' molto semplice già che la memoria diizzazione
fornita CPU.

Svantaggio: se ti ridilitiono gli interventi, li ridilitiono per TUTTI

A volte basta il Kernel quando vuole essere sicuro che nessuno lo interrompa

Distribuzione degli interrupt: mom funzione come
tutrizione nei sistemi multiprocessore.

Condizione: non è una vera e propria soluzione

2^a: Variabile di lock (soluzione software)

Il processo controlla una variabile comune con uno (o più) processi, la variabile è detta lock.

Se è 0 il processo lo setta a 1 e entra nella sezione critica (quando esce lo riporta a 0).

Se è 1 aspetta nello stato a 0.

Esiste fenomeno busy waiting.

Problema: la variabile è comune, il processo fetch-store può dare problemi di corre critiche.

3^a: Alternanza turn (soluzione software)

```
[int N=2 int turn=0]
```

```
function enter-region (int process)
```

```
while (turn != process) do nothing
```

Si ripete se turn corrisponde al process, se non lo è, lo blocca

```
[function leave-region (int process)]
```

```
turn = 1 - process
```

Questi sono i due meccanismi degli estremi della sezione critica.

Con queste due procedure facciamo fare ai processi una alternanza turn.

E' una soluzione che fa busy waiting
(un lock che non guadagna meccanismo e' detto
spin lock)

La soluzione è generalizzabile a N processi

Sono impostati turni rigidi, e si viola la 3^a
combinazione in quanto i processi potrebbero
bloccarne altri anche se non sono in
zona critica.

Un'ottimizzazione è la **Soluzione di Peterson**
per ridurre le trimoslock dell'alternanza stretta

[int N=2

int turn

int interested[N] → inizializza le celle a false

function enter-region (int process)

other = 1 - process

interested[process] = true

turn = process

while (interested[other] = true AND turn = process) do
nothing

function leave-region (int process)

interested[process] = false

Analisi della soluzione di Peterson nei 3 possibili casi:

1) Processo A va "fa bb", interessed [other] è falso quindi non vi blocca nel loop

2) Siamo dentro A, dovrebbe scattare la regola di B ma vi blocca nel loop perché
• interessed [other] è vero
• tutti_m = process è vero

Quando A finisce interessed [other] diventa falso
quindi B può proseguire

3) A e B cercano di entrare nella zona critica contemporaneamente

C'è una contesta, entrambi cercano di impostare tutti_m

Per entrambi interessed [other] sarà vero, però tutti_m avrà uno dei due valori.

Quindi, per un processo tutti_m = process sarà vero per l'altro processo sarà falso.

Quando la "vincente" farà la leave-region, per l'altro processo (che era bloccato) la interessed [other] sarà falso e quindi potrà proseguire.

- C'è ancora busy waiting

- Si può generalizzare a N processi

- Può dare problemi nei sistemi multi-procenzione per il Notching degli accessi alla memoria centrale.

Alcune architetture implementano delle "barriere" che generano al programmatore di evitare il Nondio di scrive in certe zone.

Soluzione con TSL e XCHG (avendo busy waiting)

Molti architetture mettono a disposizione la TSL (test and set lock)

Sintesi: TSL registro, lock

Era è un'operazione atomica, blocca il bus di interno.

L'effetto della TSL è
bloccare il bus
garantisce atomicità.

1) MOVE registro, lock
2) MOVE lock, 1

Funzioni:

[enter_region]:

TSL registro, lock
CMP registro, #0
JNE enter_region
RET

[leave_region]:

MOVE lock, #0
RET

La lock viene a cogliere se già c'era un blocco
e non c'era il blocco lo fa il processo stesso.

In ogni caso lock sarà 1 alla fine, controllando
il register vediamo se era bloccato (lo stavolta sarà
primo, se lo era (quindi era 1) ritorna alla
etichetta enter_region con lo JNE (è un loop)

Se non c'era il blocco fa RET e prosegue il processo
(che ha bloccato la funzione mettendo lock a 1)

La XCHG è molto simile alla TSL

Le soluzioni viste sono basate sullo SPIN lock
Lo spin lock causa:

- Prezzo di risorsa
- Problemi di inversione
- Si muore

Sono H e L due processi
riguardanti ad altri e hanno priorità e via
della stessa volta contraria.
H è bloccato per un I/O (non finito o una
sezione critica)

L entra in una sezione critica, finisce l'I/O e H
è tornata pronta.

L viene interrotto perché H ha maggiore priorità ma H sta
pronta ad entrare in sezione critica e
non può far colpa della variabile di lock e (=1)
detta da L.

Si è creato uno stesso

Soluzione

Si fa al processo la possibilità di bloccarsi
(in modo passivo, viene rimesso al proc. pronto)
e più di rivedere gli altri al momento abatto.

Strumenti: sleep, wakeup Sono due syscall
Se un processo fa sleep si addormenta
Se fa wakeup in un altro processo lo risveglia

Problema del Produzione - Consumazione

Abbioro due processi produzione
processi consumo \Rightarrow & informazione

variabile count comune, iniz. a 0

function producer()

while(true) do

item = produce_item()

if (count=N) sleep() \leftarrow inserisci_items(item)

Count = Count + 1

if (Count=1) wakeUp(consumer) \leftarrow buffer ~~vuoto~~
con un item, megliano cons.

function consumer()

while(true) do

if (Count=0) sleep() \leftarrow buffer vuoto, banane

item = remove_item()

Count = Count - 1

if (Count=N-1) wakeUp(producer) \leftarrow C'è una
coda libera,
meglio Prog.

Facciamo questa sol. perché non aggiungere
quanto velocemente i processi.

Si monterebbe in parte perché ci sono dei limiti e
faccendo forzare e negare i processi

Problema: I due processi potrebbero bloccarsi

Il SO blocca consumer poco prima di fare sleep.
Il producer fa wakeUp ma consumer ha già
stato forzando

Il producer continua a ciclare e il buffer
è sempre vuoto e la CPU è in sleep.

~~nel frattempo il SO ha non cessa la CPU al
Consumer che esegue sleep (dopo che producer
ha fatto wakeup)~~

Sono entrate in sleep. Sono in stallo per sleep
di un'area critica

Situazione inversa: il SO blocca producer joco
già dalla sleep. Consumer fa wakeup sul
producer (non dormiente)

Consumer cicla e il buffer è vuoto, la CPU è in sleep.
Nel frattempo il SO ha non cessa la CPU al
producer che esegue sleep.

Sono entrate in sleep.

Soluzione: bit-flag ~~se si pone a impostare a 1 se si pone a~~
~~impostare a 0 meglio un processo~~
~~NON dormiente~~

Come funziona il bit?

Se il bit è 1 la sleep non farà dormire
il processo

Il problema sta nel fatto che

`if (count=0) sleep()`

NON fa nulla

`if (count=N) sleep()`

Il bit di attesa ha un problema: conserva
tutte le sue regole
(e ciò può già consentire che mandano wakeup, non
è più possibile)

Semaforo

Generalizzazione del modello sleep e wake up

- E' una variabile comune S
- Si definiscono le operazioni down, up (o wait, signal)

Ese decremente e incremento S

Down per non più decrementare S se è 0,
diventa bloccante.

(Un processo se prova a decrementare S se è 0
è bloccato)

La down non può fare il decremento se S è 0
ma non finisce se non ne ha a fondo.

La up non è mai bloccante

L'incremento potrebbe svegliare un certo processo

Gli esponenti di up e down sono i processi
ma nemmeno.

Da un punto di vista logico c'è la cosa dei
processi bloccati dal ~~processo~~ S. semaforo S.

Le operazioni down e up sono atomiche

Non ci sono corde critiche sul semaforo

La garanzia è che la disponibilità degli interrupt
e la memoria è della CPU o meno la
TSL / XCHG. ← si usa un lock che "protegge" S

C'è sempre un po' di spin lock ma è meno
presente rispetto allo spin lock tra processi

L'implementazione avviene senza busy waiting,
mentre la lista dei processi bloccati

Esercizio

- A prova a fermentazione S che è + 0, va in stasi nella coda
 - B stessa cosa
 - C fa la uj, B si blocca
- $B \xrightarrow{++} 1 \xrightarrow{} B \text{ si blocca} \rightarrow S \xrightarrow{-} 0$

Il semaforo ha non utilizzati

Semaforo Mutex : per la mutua esclusione

Ricorda che i processi di non entrare tutti e due in sezione critica.

La S si trova tra 0 e 1

All'inizio della sezione critica: $down(S)$

Allo fine: $up(S)$

In questo modo garantisce la mutua esclusione

Semaforo contatore per le risorse per l'incronizzazione

Conteggio: una coda tipologia di risorse

Esempio: numero di item nel buffer

Quando il buffer si riempie si potrebbe bloccare un altro processo

Risolvere Prodotto-Consumo con semafori

Dati controlli

$$\text{int } N = 100$$

semaphone mutex = 1

semaphone empty = N

semaphone full = 0 \rightarrow Conteggio delle risorse

function producer()

while(true) do

item = produce_item()

down (empty)

down (mutex)

insert_item(item)

up (mutex)

up (empty)

bloccante se
empty è 0

Ci significano
di lavorare su
nella critica

function consumer()

while(true) do

down (full)

down (mutex)

item = remove_item()

up (mutex)

up (empty)

consume_item(item)

bloccante se full = 0

Se $N=1$
mutex è
iniziale.

Ci significano mutex esclusione nelle zone
critiche: insert_item() e remove_item()

Blocciamo Producer se le celle vuote (empty) sono 0,
bloccano Consumer se le full sono 0

L'ordine tra down e up è fondamentale.

down e up sul mutex devono coincidere solo
cioè de è regione critica

Il procedimento funziona anche con più di due
processi

Nei thread stente (1 o molti) che fanno riferimenti
a un solo processo N implementano mutex con TSL

mutex_lock:

```
TSL REGISTRA, MUTEX  
CMP REGISTRA, #0  
JZ F OK  
CALL thread_yield  
JMP mutex_lock
```

OK: RET

mutex_unlock:

```
MOVE MUTEX, #0  
RET
```

E' tutto implementato in modifica user

Non ha nemmeno fatto busy waiting (unisci la
yield con lavoriamo in altri thread)

E' molto efficiente dato che non lo lock né lo
unlock richiedono il Kernel

Queste due procedure sono ricavate dagli estremi
della zona critica nei riferimenti a mutex.

FUTEX

I mutex nella user-space sono inefficienti per
lo spin lock già come hanno dimostrato.

Ci sono situazioni in cui è meglio fare spin lock
classico altrice in cui è meglio usare in Kernel

Per questo nome Futex = fast user space mutex
che entra in syscall because se è direttamente
necessario

Il Futex ha due parti

- A livello Kernel
- Libreria utente

che formano coda di thread bloccati

che usa un variabile lock

la catena per la sezione critica è
in mutuale con lo TSL (utile)

L'unico motivo per cui richiamiamo il Kernel è
per ridere bloccare il processo / thread
L'unico caso in cui si richiede il Kernel è
la contesta di azione critica dei thread.