Sleep e Wakeup

Sotto forma di chiamata di sistema si ha una primitiva che riesca ad addormentare, su richiesta, un thread. Esiste anche la controparte in grado di risvegliare il thread dormiente.

Produttore-Consumatore

Il problema produttore-consumatore è un classico problema di sincronizzazione dei processi nei sistemi operativi, che si presenta quando un insieme di processi produttori generano dati e li inseriscono in una risorsa condivisa, mentre un insieme di processi consumatori prelevano i dati dalla stessa risorsa. Il problema richiede la sincronizzazione tra i processi produttori e consumatori, in modo che i produttori producano i dati solo se la risorsa ha posti vuoti per ospitarli, e i consumatori consumino i dati solo se la risorsa contiene dati disponibili per essere prelevati. La sincronizzazione può essere implementata mediante semafori, mutex o altre primitive di sincronizzazione offerte dal sistema operativo.

PRELAZIONE

```
function producer()
    while (true) do
        item = produce_item()
        if (count = N) sleep()
        insert_item(item)
        count = count + 1
        if (count = 1)
            wakeup(consumer)
```

```
function consumer()
    while (true) do
        if (count = 0) sleep()
        item = remove_item()
        count = count - 1
        if (count = N - 1)
            wakeup(producer)
        consume_item(item)
```

Count è una variabile condivisa, essa rappresenta il buffer condiviso di capienza massima N a cui si garantisce la mutua esclusione. Sleep è una funzione bloccante nei confronti del chiamante e solo la funzione wakeup potrà rimandare in attività il processo chiamante. Anche in questo modello che utilizza sleep e wakeup presenta un problema : supponiamo che il consumer abbia accesso alla risorsa quando il buffer è vuoto. A seguito dell'istruzione if esso va ad addormentarsi con la sleep ma nel mezzo di queste due istruzioni avviene la prelazione che gli sottrae CPU (consumer non dormiente quindi) e passa il controllo al producer , esso inserisce un elemento rendendo il buffer non vuoto e dato che si ha un elemento si invoca la wakeup al consumer, che non stava nemmeno dormendo. Il consumer riceve il controllo ed esegue la sleep, che non era riuscito ad eseguire prima e si addormenta. Esso non si sveglierà più e ad un certo punto , a seguito della saturazione del buffer, anche il producer si addormenterà ed entrambi i processi dormiranno causando quindi uno stallo. Lo stesso problema può verificarsi anche in senso opposto bloccando prima il producer e poi il consumatore.

Una sveglia lanciata su un processo sveglio va persa. Un bit di attesa wakeup si attiva quando viene lanciata una wakeup su un processo che non sta dormendo : se invoco una sveglia sul processo sveglio il bit passa a 0 "consumando" la sveglia . Si può di nuovo manifestare il problema nel caso in cui ci siano più produttori che lanciano wakeup multipli . Il problema di base risiede nel fatto che un bit può memorizzare al massimo una sveglia. Quest'idea ha portato all'implementazione dei semafori di Dijkstra.

Semafori

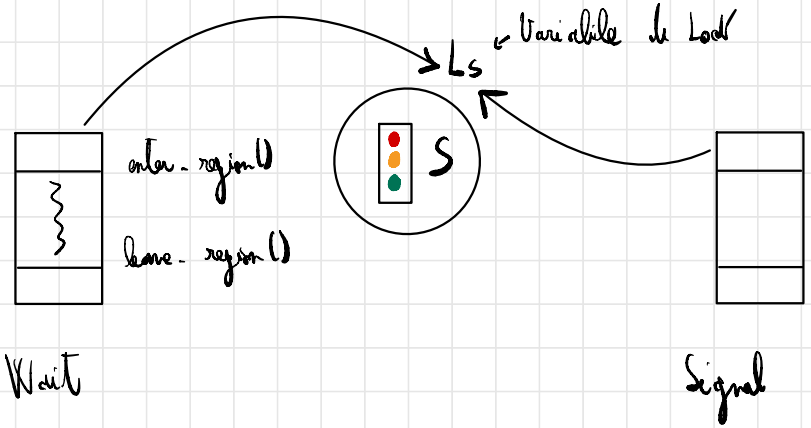
Generalizzano il concetto di sleep-wakeup, dando la possibilità di contare il numero di sveglie perse al posto di poterne memorizzare solo una. Un semaforo può essere visto come una variabile intera con delle proprietà precise :

- Il semaforo non può essere mai negativo. Non è importante il valore di inizializzazione del semaforo.
- Le uniche operazioni consentite sono l'incremento e il decremento Down = Wait = Sleep / Up = Signal = wakeup
  Eseguire una wait su un semaforo non nullo non è problematica ma decrementare un semaforo nullo è una chiamata bloccante. Un processo per il quale il semaforo è nullo dorme e si può svegliare solo a seguito di una Signal.

Se tutto si limitasse a questa descrizione i requisiti sarebbero a livello utente mentre il meccanismo dovrebbe svolgere tutto in kernel mode. Inoltre in uno scenario problematico, essendo il semaforo stesso una variabile condivisa, si può vare una race condition sul semaforo stesso.

Soluzione : le operazioni di wait e signal devono essere atomiche , per non poter essere interrotte da eventi che coinvolgono il semaforo stesso. Devo garantire la mutua esclusione per evitare la race condition sulle operazioni implementate sul semaforo. Qui è accettabile che tutto venga bloccato in quanto la up e la down sono operazioni brevi e che vengono eseguite a livello kernel.

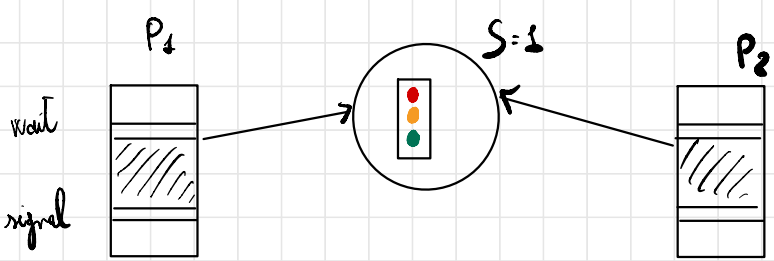
Disabilitare gli interrupt potrebbe risolvere il problema , garantendo l'atomicità in sistemi monoprocessore a seguito di una prelazione. La soluzione non è accettabile su sistemi multicore, quindi si ricorre ai meccanismi di TSL/XCHG.



Essendo implementati dal sistema operativo , questi meccanismi (semafori) non soffrono della busy waiting. Ogni semaforo implementa una coda di processi bloccati sul semaforo stesso : quando un processo viene bloccato (il semaforo viene posto a 0) il suo PCB viene inserito nella coda dei processi bloccati sul semaforo . Quando viene svegliato un processo, qualora ci fosse più di un processo addormentato, si deve avere un modo per scegliere quale processo verrà svegliato dal "parcheggio" rappresentato dalla coda .

Mutex (Mutual-Exclusion)

Il semaforo è uno strumento abbastanza generale che ci permette di risolvere problemi diversi. Per garantire la mutua esclusione nell'accesso ad una struttura dati condivisa si può usare un semaforo. Un mutex è un meccanismo di mutua esclusione implementabile con un semaforo generale . Prima di entrare nella sezione critica, un processo invoca la wait() che bloccherà l'ingresso ad un altro processo che troverà il semaforo a 0 (bloccato). Dopo l'esecuzione della sezione critica il processo invoca la signal() ed il semaforo torna ad 1 (significa che può essere utilizzato di nuovo.

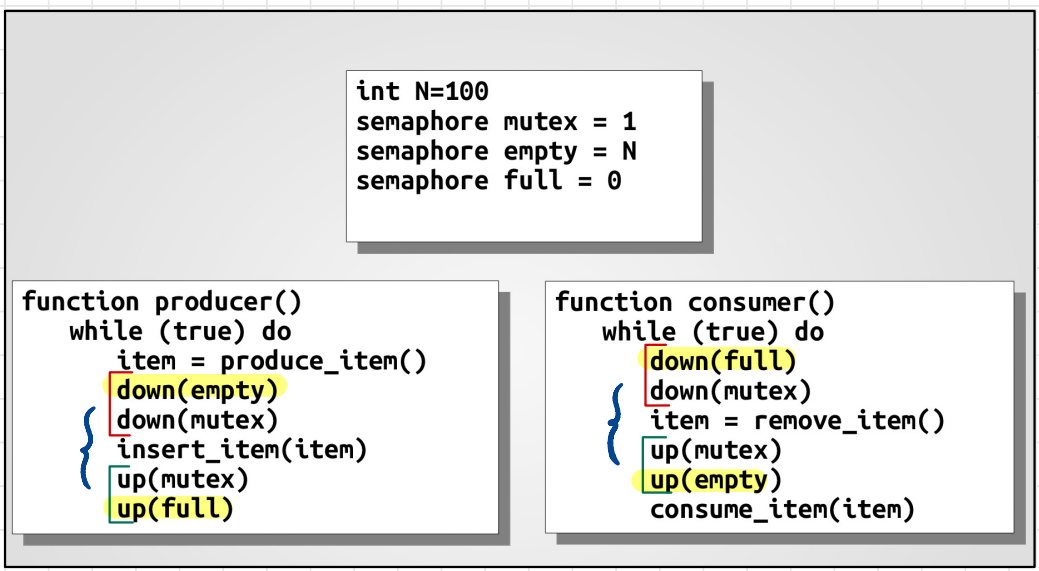


Conteggio di risorse (sincronizzazione)

Un valore del semaforo che passa solo da 0 ad 1 risolve la mutua esclusione. Altri impieghi riescono a risolvere il conteggio di risorse. Sono i semafori contatori implementati come empty e full. Un semaforo può essere utilizzato per controllare il conteggio di risorse disponibili e garantire che solo un numero specifico di processi o thread accedano a tali risorse contemporaneamente.

Abbiamo semafori di tre tipi :

- mutex (mutua esclusione): garantisce il corretto accesso ad alcune strutture dati condivise (buffer).
- empty (sincronizzazione) : conteggia il numero di slot vuoti, ovvero quanti item ci sono nel buffer.
- full (sincronizzazione) : conteggia il numero di slot pieni nel buffer.



Producer

La logica che c'è dietro la "down(empty)" è quella di prenotare uno slot vuoto . Se non sono presenti slot vuoti l'esecuzione si blocca qui. Proseguo mettendo il mutex a 0 con "down(mutex)" per garantire che non ci saranno altri accessi e rimettendolo ad 1 con "up(mutex)" alla fine della sezione critica. Dopo viene effettuata una chiamata ad "up(full)" per incrementare il numero di slot pieni.

Consumer

Esegue la down(full) per prenotare uno slot pieno. Se non ci sono slot pieni nel buffer l'esecuzione si blocca su quest'istruzione. Il comportamento delle down(mutex) e up(mutex) delimitano la zona critica come nel caso del producer.

Questa soluzione è scalabile anche al caso in cui ci sono più produttori e più consumatori
Una sezione critica non deve mai contenere chiamate bloccanti e deve essere il più breve possibile. Invertire le istruzioni evidenziate in rosso genera uno stallo in quanto inseriamo una chiamata bloccante nella zona critica.
Invertire le istruzioni evidenziate in verde invece non causa blocchi ma allunga inutilmente la zona critica che dovrebbe rimanere il più breve possibile.

Mutex e Thread utente

Se utilizzassi un semaforo nel modello di thread a livello utente, non avrei supporto da parte del SO sull'utilizzo dei thread ma potrei rendere efficiente il sistema attraverso l'uso di TSL (nonostante faccia spin lock).
L'implementazione di un mutex con i thread a livello utente utilizzando l'istruzione TSL non soffre di busy waiting perché l'istruzione TSL è un'operazione a livello di hardware che garantisce l'atomicità dell'operazione stessa. Ciò significa che l'istruzione TSL atomica esegue l'operazione di acquisizione del lock del mutex in modo atomico, senza la possibilità di interruzioni o interferenze da parte di altri thread.

Quando un thread tenta di acquisire il lock del mutex e l'istruzione TSL atomica restituisce il valore "locked", il thread non entra in un loop di busy waiting, ma viene invece sospeso dal kernel e messo in attesa di una notifica che il lock del mutex è stato rilasciato da un altro thread. In questo modo, il thread non spreca cicli di CPU inutilmente e il sistema operativo può utilizzare le risorse del processore in modo più efficiente.

Quindi l'implementazione di un mutex con i thread a livello utente non soffre di busy waiting grazie alla garanzia di atomicità dell'operazione di acquisizione del lock e alla possibilità di sospendere il thread in attesa del lock senza consumare risorse di CPU inutilmente.

Il modello è funzionante in quanto con i thread a livello utente gira tutto in modalità utente.

```
mutex_lock:
    TSL REGISTER,MUTEX
    CMP REGISTER,#0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
ok:RET
```

```
mutex_unlock:
    MOVE MUTEX,#0
    RET
```

Producer-Consumer :

- Sleep e Wkeup
- Mutex e Semaforo per le zone critiche di producer e consumer