8° lezione S.O.

Due Date	@April 1, 2025
Multi-select	Futex Monitor Produttore-Consumatore
Status	Done

Futex

In Linux, i **futex** (**fast user space mutex**) sono una **soluzione ibrida** che **combina** i vantaggi delle
implementazioni viste precedentemente. Ad esempio, la libreria Pthreads li utilizza per implementare i mutex.
L'idea chiave dei futex è combinare la gestione della **contesa a livello user-space** tramite una variabile di lock, con un **meccanismo di blocco gestito dal kerne**l, che entra in gioco solo quando strettamente necessario. Questo approccio:

- **Riduce significativamente l'overhead delle chiamate** di sistema rispetto ai normali semafori e mutex, in cui sia la gestione del lock che il bloccaggio dei processi era gestito dal kernel attraverso singole procedure
- Garantisce che il kernel venga richiamato solamente quando **strettamente necessario.**

Un futex ha due componenti

- **Componente user-space**: il processo controlla autonomamente la variabile di lock e tenta di
  acquisirla senza coinvolgere il kernel, garantendo la mutua eclusione
  - Ciò avviene tramite istruzioni di `TSL` o `XCHG`
- **Componente kernel-space:** se il lock non può essere acquisito, il kernel sospende il
  processo e lo risveglia quando il lock diventa disponibile

- L’overhead derivante dal passaggio al kernel è minimizzato poiché il kernel viene coinvolto solo per le operazioni di `sleep` e `wakeup` , e non per manipolare direttamente la variabile di lock. In ogni caso il processo sarebbe dovuto essere sospeso, quindi l’intervento del kernel era necessario a prescindere

Garantiscono quindi una implementazione efficace dei mutex per processi.

Monitor

Il costrutto monitor è una **struttura astratta fornita da alcuni linguaggi di programmazione** ed è un meccanismo ad **alto livello** rispetto ai semafori. Esso è simile a un **oggetto**, poiché possiede proprietà e metodi, il cui obiettivo principale è garantire **mutua esclusione** e **sincronizzazione tra thread**. Per i processi, in quanto questi potrebbero essere scritti in linguaggi diversi, si utilizzano i semafori e mutex. Ad esempio, in Java la keyword `synchronized` utilizza un monitor per
garantire la mutua esclusione

I monitor **non sono offerti direttamente dal sistema operativo**, ma sfruttano le primitive offerte dal sistema operativo per **mascherare al programmatore i meccanismi sottostanti**. Ad esempio, su Linux, un monitor potrebbe essere implementato tramite **futex**.

Tutte le proprietà del monitor (dati e strutture dati) sono interne e private e l’accesso avviene esclusivamente attraverso interfacce fornite dal monitor stesso: quest’ultime permettono l’accesso alle strutture dati interne ma **non possono accedere a strutture esterne**. Un’importante caratteristica è che **tutti i metodi definiti internamente al monitor sono eseguiti in mutua esclusione**: se un thread ha invocato un metodo del monitor (si dice che il thread è dentro il monitor), nessun altro thread può farlo. La mutua esclusione, quindi, è garantita di default.

Vantaggi dei Monitor

Uno dei principali vantaggi dei monitor è che il programmatore **non deve preoccuparsi direttamente dei dettagli della mutua esclusione**: non deve gestire variabili di lock o ricordarsi di eseguire operazioni di `up` e `down` , **riducendo così la possibilità di errori** (ad esempio, dimenticare di rilasciare un lock tramite una `down`). Il funzionamento è smile alla libreria Pthreads; possiamo dire che sono dei wrapper.

Tuttavia, il solo meccanismo di mutua esclusione non è sufficiente a garantire una corretta
sincronizzazione tra thread, che necessita di un meccanismo per gestire i casi in cui un thread deve aspettare che una certa condizione si verifichi (es. **problema del produttore-consumatore**).

Variabili di Condizione

Per gestire la sincronizzazione tra thread, i monitor utilizzano delle **variabili condizione**, che sono delle **etichette**. Si trovano sempre all'interno del monitor e permettono ai thread di **sospendersi** e **attendere** che **determinate condizioni siano soddisfatte** prima di proseguire l'esecuzione. Un thread può eseguire due operazioni sulle variabili di condizione:

- `wait` : se una condizione è **vera**, il thread viene **sospeso** e **rilascia il lock**, permettendo ad altri
  thread di entrare nel monitor
- `signal` : se una condizione è **falsa**, **risveglia un thread sospeso** sulla variabile condizione
  corrispondente

È importante notare che queste procedure si occupano di prendere e rilasciare il lock in maniera automatica. Le variabili condizione non hanno:

- Uno **stato numerico** come i semafori, ma rappresentano **etichette** su cui i thread possono
  sospendersi e risvegliarsi in base a delle condizioni
- Una memoria, in quanto il monitor garantisce la mutua esclusione, quindi non si verificano
  race condition nell'uso di queste variabili (es. scenario in cui il processo viene prelazionato
  prima di una
  `sleep` , con un altro processo che provoca una `wakeup`

Semantiche della Signal

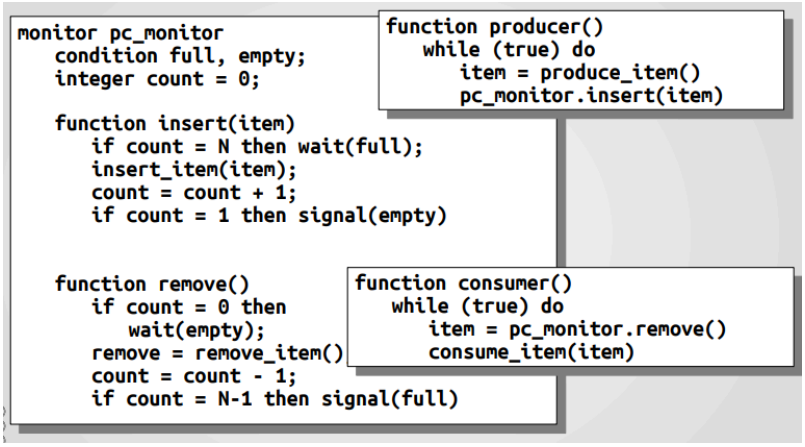
L'operazione `signal` nei monitor può seguire diverse semantiche, ognuna con implicazioni sulla
mutua esclusione e sulla gestione dei thread:

1. **Signal & Wait** (**Hoare, teorico**): il thread che esegue `signal` **si sospende immediatamente**, cedendo l'esecuzione all'altro thread in sleep, che viene risvegliato. Non appena quest'ultimo termina l'esecuzione, il thread originale si **risveglia**
    - Può causare interlacciamento indesiderato dei metodi all'interno del monitor non garantendo correttamente la mutua esclusione
2. **Signal & Continue** (**Mesa, adottata da Java**): il thread che esegue `signal` **continua la sua esecuzione** ed il **sistema prende nota** della richiesta. Non appena la sua esecuzione termina, il thread che ha ricevuto la `signal` si risveglia. In poche parole, deve attendere che il monitor venga liberato
    - Come unico problema potrebbe accadere che la situazione precedente che ha prodotto una `signal` sia **cambiata** quando il nuovo thread inizia la sua esecuzione, richiedendo eventuali verifiche. L'approccio quindi può andare bene solo se le istruzioni che il primo thread esegue dopo la signal non invalidano il motivo per cui è stata chiamata.
3. **Signal & Return** (**Brinch Hansen, Concurrent Pascal**): la `signal` deve essere **l'ultima istruzione** che esegue il primo thread, evitando che quest'ultimo possa alterare lo stato del monitor dopo aver svegliato un altro thread
    - Riduce l'interlacciamento dei processi.

Produttore-Consumatore con i Monitor

L'uso dei monitor semplifica la gestione della mutua esclusione e della sincronizzazione nel problema **produttore-consumatore**, eliminando la necessità di gestire esplicitamente i lock.
Definiamo il monitor
`pc_monitor` con:

- **Variabili di condizione**: `full` e `empty`, che regolano l'accesso al buffer
- Variabile `count` : tiene traccia del numero di elementi nel buffer
- **Metodi** `insert(item)` e `remove()` che gestiscono rispettivamente l'inserimento e la rimozione di
  elementi



Il produttore

1. Genera un nuovo item e tenta di inserirlo nel buffer
2. Se il buffer è pieno, si blocca su `wait(full)`
3. Dopo aver inserito l'elemento, incrementa `count`
4. Se il buffer era vuoto, segnala `signal(empty)`, risvegliando un consumatore in attesa

Il consumatore:

1. Tenta di prelevare un elemento dal buffer
2. Se il buffer è vuoto, si blocca su `wait(empty)`
3. Dopo aver prelevato l'elemento, decrementa `count`.
4. Se il buffer era pieno, segnala `signal(full)` , risvegliando un produttore in attesa

Scambio messaggi tra processi

Lo scambio di messaggi è un meccanismo di comunicazione tra processi che **non richiede**
memoria condivis
a. I processi scambiano dati attraverso l'invio e la ricezione di messaggi **gestiti dal sistema operativo**
; l'idea è simile alla comunicazione tra processi di macchine diverse nelle reti. Le chiamate di sistema relative sono:

- `send(destinazione, messaggio)`: il processo mittente invia un messaggio a un altro processo o a una mailbox

- `receive(sorgente, messaggio)` : il processo destinatario attende e riceve un messaggio

Buffer di messaggi

Il **sistema operativo** usa un buffer per immagazzinare i messaggi quando il destinatario non è pronto a riceverli, similmente al problema **produttore-consumatore**. Per questo motivo, le syscall di comunicazione sono **bloccanti**:

- `send()` può bloccarsi se il buffer è pieno.
- `receive()` può bloccarsi se non ci sono messaggi disponibili.

N.B.: il buffer è fornito dal sistema operativo ed esposto tramite le syscall, per questo si dice che il meccanismo non richiede memoria condivisa, non se ne occupano i processi.

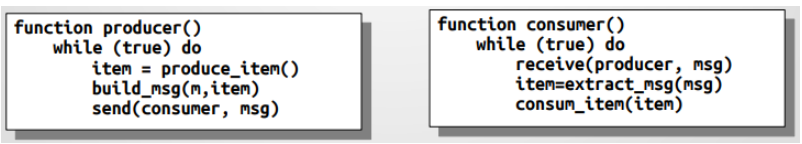
Mailbox

Per gestire la comunicazione tra **più mittenti e destinatari** si usano le mailbox, che funzionano
come caselle postali intermedie in cui un sender invia un messaggio alla mailbox e un receiver lo
preleva

Problema del Produttore-Consumatore con i Messaggi

Il problema può essere risolto anche utilizzando lo scambio di messaggi. In questo scenario:

- Il **produttore** genera elementi e li invia al **consumatore** tramite messaggi.
- I **consumatore** riceve i messaggi ed elabora i dati.



Se il **produttore è più veloce**, riempie il buffer e si blocca in attesa che il consumatore ne elabori uno. Se il **consumatore è più veloce**, consuma tutti i messaggi e si blocca in attesa di uno nuovo

Vantaggi e Svantaggi

Vantaggi

- Funziona su **sistemi distribuiti** (i processi possono trovarsi su macchine diverse)
- **Non richiede memoria condivisa**

Svantaggi

- **Inefficienza dovuta alle chiamate di sistema frequenti**.
    - Le due primitive implicano una chiamata di sistema, e se i processi che comunicano sono
      tanti con una frequenza elevata, si ha un overhead non trascurabile
- Le due primitive implicano una chiamata di sistema, e se i processi che comunicano sono tanti con una frequenza elevata, si ha un overhead non trascurabile
- **Maggiore latenza rispetto alla memoria condivisa**