La mutua escluzione, Mutex e Pthreads

Un mutex è una **versione semplificata dei semafori** che può essere usata per gestire l'accesso a risorse o codice condiviso.

> **Nota**
>
> Conviene usarlo se non bisogna tenere di conto i numeri degli accessi alla risorsa o altre cose più complesse.

Può trovarsi in due stati:

- **Locked**
- **Unlocked**

> **Nota**
>
> Basta un solo bit per rappresentarlo.

Le procedure principali sono:

- **mutex_lock**
- **mutex_unlock**

Quindi se un thread vuole accedere ad una regione critica, chiama mutex_lock se il mutex è:

- locked --> thread attende.
- unlocked --> thread entra nella regione critica.

Al termine dell'accesso il thread chiama mutex_unlock per liberare la risorsa.

> **Nota**
>
> Se il thread non può acquisire un lock, chiama thread_yield per cedere la CPU ad un altro thread.

> **Considerazioni aggiuntive**

- I mutex possono essere implementati nello spazio utente con istruzioni come TSL o XCHG.
- Alcuni pacchetti di thread offrono mutex_trylock, cioè che si tenta di acquisire il lock o restituisce un errore, senza bloccare.
- I mutex sono efficaci quando i thread operano in uno spazio degli indirizzi comune.
- La condivisione di memoria tra processi può essere gestita tramite il kernel o con l'aiuto di sistemi operativi che permettono la condivisione di parti dello spazio degli indirizzi.

All'aumentare del parallelismo diventa cruciale migiorare anche l'efficienza nella sincronizzazione tra processi.

Per il momento abbiamo visto sipnlock e mutex per la gestione, ma questi possono tornare utili solo per attese brevi.
Inoltre se le contese sono poche si va a **sprecare molto tempo della CPU** per passare al Kernel.

Futex (Fast User Space Mutex)

Per risolvere i problemi sopra elencati possiamo utilizzare i **futex**.
Servono per implemetare **lock elementari evitando il kernel finché non è necessario.**

> ⚠ **Vantaggi e Svantaggi**
>
> I futex migliorano le prestazioni riducendo il passaggio al kernel, ma non sono uno standard (`#include <inux/futex.h>`).

Sono composti da due parti:

- Servizio kernel.
- Libreria utente.

Le operazioni eseguibili con il futex sono:

- **Setting di una variabile di lock**, che dice al thread se può accdedere alla risorsa.
- **Passaggio al kernel**, se un thread è bloccato da un altro.

Quando un lock è rilasciato, il kernel può essere chiamato per svegliare altri processi in attesa.

> ⓘ **Pthread**

Libreria che ci permette di utilizzare le funzioni di sincronizzazione dei thread

Monitor

Un *monitor* è un costrutto che incapsula sia i dati condivisi che le operazioni (o funzioni) che li manipolano. Fornisce un modo per controllare l'accesso esclusivo a una risorsa condivisa, assicurando che solo un processo alla volta possa eseguire operazioni su di essa.

Solo un processo può essere attivo in un monitor in un dato momento, garantedo la mutua esclusione, che è gestita dal compilatore.

Scambio di messaggi

E' un metodo di comunicazione tra due processi che si basa sulle funzioni atomiche:

- **send**
- **recieve**

E' spesso utilizzato, nonostante i alcuni problemi che lo circondano come:

- Possibili **messaggi persi nella rete.**
- Necessità di **ACK**.
- Necessità di autenticazione e denominazione dei processi.

Per evitare questi errori utilizziamo le **Barriere**.

> **Barriere**
>
> Utilizzate per sincronizzare i processi in fase diverse. Quando un processo ragguinge una barriera, attende fino a quando tutti gli altri processi la raggiungono.

Inversione delle priorità

Il problema dell'inversione delle prirità si verifica quando:

- Un thread ad alta priorità aspetta una risorsa bloccata da un thread di bassa priorità.
- Un thread di priorità media, che non ha nulla a che fare con la risorsa, impedisce al thread di bassa priorità di completare il suo lavoro.

Tutto questo comporta che il thread ad alta priorità non riesce a lavorare anche se dovrebbe avere la precedenza su tutti.

La soluzione sta nella tecnica chiamata: **Priority Inheritance Protocol**.

> ⓘ **Priority Inheritance Protocol**
>
> Eleviamo il thread con priorità bassa a livello del thread bloccante, così il thread di bassa priorità può finire velocemente il suo lavoro, lasciare spazio al thread di alta priorità, per poi tornare al suo livello originario.

ready-copy-update

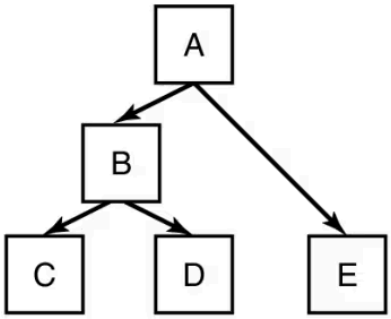
Dobbiamo trovare un modo per evitare i lock così da non rischiare l'inversione delle priorità.

Possiamo permettere accessi in lettura e scrittura su strutture dati condivise senza imporre lock? NO.

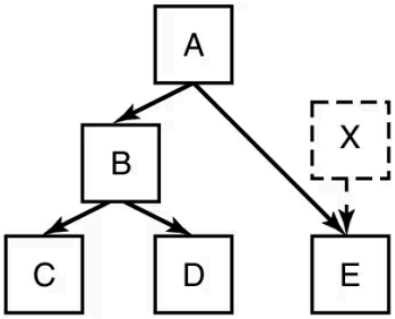
> ⚠ **Esempio di problema**
>
> Immaginate di avere due thread che accedono ad un array, uno calcolca la media, l'altro lo ordina.
> Il primo thread avrà difficoltà a calcolare la media esatta se l'altro continua a spostargli gli elementi.

Ma esiste una maniera che consente ad un'entità scrittore di aggiornare la struttura dati anche se questa è in uso da parte di altri processi.
Praticamente ogni entità lettore deve leggere una versione obsoleta, oppure solo quella nuova, della struttura dati.

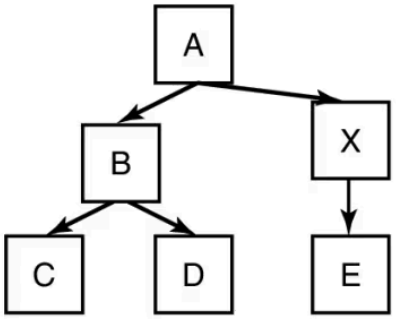
Aggiunta di un nodo:



(a) Albero originale.

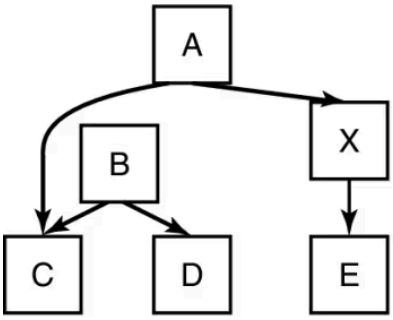


(b) Inizializzate il nodo X e connettete E a X. I thread in lettura in A e E non sono influenzati.

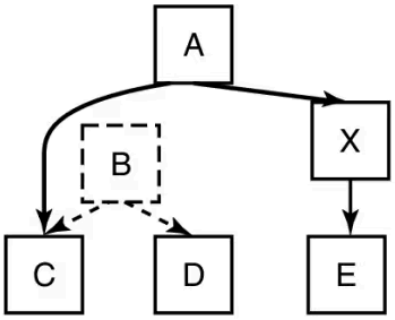


(c) Quando X è completamente inizializzato, connettete X ad A. I thread in lettura attualmente in E avranno letto la vecchia versione, quelli in A leggeranno la nuova.

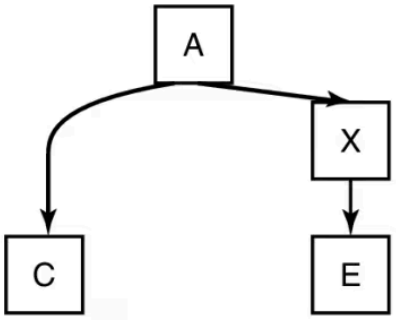
Rimozione di nodi:



(d) Scollegate B da A. Notate che possono esserci altri thread in lettura in B. Tutti i thread in lettura in B vedranno la vecchia versione dell'albero, quelli in A vedranno la nuova.



(e) Attendete finché non siete certi che tutti i thread in lettura abbiano lasciato B e C. Non è più possibile accedere a questi nodi.



(f) Ora si possono eliminare B e D in tutta sicurezza.