Semafori - Sincronizzazione a23 - Dijkstra

Un semaforo è una semplice variabile non-negativa condivisa tra threads. Questa variabile è usata per risolvere il problema della sezione critica e per ottenere la sincronizzazione tra processi in ambienti con più processori.

Un semaforo S è un valore intero (variabile) che, a parte per la sua inizializzazione, è accessibile solo tramite due operazioni atomiche: wait() e signal()

wait() -> P "Testare"
signal() -> V "Incrementare"

Definizione della wait()
Questa operazione controlla se la variabile S (il semaforo) è minore uguale a 0. Finché S è <=0 non vengono eseguite operazioni. Non appena S diventa >0, S viene decrementata.

Se il valore di S è <=0 vol dire che un altro processo è nella sua sezione critica, e quindi sta facendo uso delle ~~sue~~ risorse condivise. In questo periodo nessun altro processo dovrebbe entrare nella sua sezione critica, ed è per questo motivo che un processo che vuole entrare nella sezione critica ATTENDE che essa sia vuota tramite il loop "while".

Siccome nella wait() viene TESTATO il valore di S, la wait è anche detta "TEST"

Definizione della Signal()
La signal() non fa altro che incrementare il valore del semoforo S. La funzione viene chiamata quando il processo che stava facendo uso del semoforo ha la sua sezione critica completa la sua operazione e quindi esce dalla SC.
Incrementando S il processo denota di aver rilasciato il semoforo.

Importante:
Tutte le modifiche al valore intero del semoforo effettuate nelle operazioni wait() e signal() debbono essere eseguite in modo INDIVISIBILE. Questo vuol dire che quando un processo modifica il valore del semoforo, nessun altro processo può modificarne il valore.

Diversi tipi di semaforo:
Semaforo Binario
Il valore del semoforo Binario ha range tra 0 e 1. Su alcuni sistemi questo tipo di semoforo è noto come Mutex Locks, visto che esso fornisce la mutua esclusione.
Mutua esclusione: con questo termine si invoca un processo di sincronizzazione tra processi o thread concorrenti con il quale si impedisce che più task parallelli accedano contemporaneamente ai dati in memoria.
Tornando al semoforo binario, quando S vale 0, vuol dire che già un processo sta eseguendo la sua sezione critica, e quindi deve attendere. Quando invece S=1, il processo è libero di eseguire la sua sezione critica.

Esempio

Siamo nel caso in cui un processo è nella sua sezione critica, acceduta grazie al semaforo.
Facciamo finta che un processo voglia a sua volta entrare nella sua sezione critica; eseguirà la wait(), ma siccome il processo precedente è ancora nella sua S.C., allora S≠1 e di conseguenza il processo resta fermo nel while loop ad attendere.

Dopo un certo lasso di tempo, il primo processo avrà terminato l'esecuzione della sua sezione critica, e di conseguenza chiamerà la signal(), che ha come unico compito quello di incrementare S.

Nel frattempo è presente il secondo processo che sta attendendo all'interno del loop while. Appena S diventa >0, esce dal loop e decrementa il valore di S; può finalmente entrare nella sua S.C.

Vantaggi e svantaggi dei semafori

Il maggiore svantaggio dei semafori è sicuramente il fatto che essi richiedono il busy waiting.
Per busy waiting intendiamo che mentre un processo è nella sua sezione critica, qualsiasi altro processo che prova ad entrare nella propria sezione critica deve restare in un "loop" per tutta la durata dell'attesa.

Il busy waiting è un problema principalmente perché spreca cicli di CPU che altri programmi potrebbero usare in modo produttivo. Questo tipo di semaforo è anche chiamato spinlock perché il processo gira (spins) mentre attende il lock.

Soluzione

Possiamo modificare la definizione di wait() e signal() per evitare l'utilizzo del busy waiting.
Recap def. wait(): quando un processo esegue la wait e trova il valore di S non positivo, deve attendere.

Invece di entrare in busy waiting, il processo può bloccarsi autonomamente; l'operazione "block" pone il processo in una coda di attesa associata al semaforo, e lo stato del processo viene cambiato nello stato di attesa.

Il controllo viene trasferito allo scheduler CPU, che seleziona un altro processo da eseguire. In questo modo la CPU può essere usata da altri processi, invece di "perdere tempo" in loop inutili.