Sistemi operativi 2/ed
Paolo Ancilotti, Maurelio Boari, Anna Ciampolini, Giuseppe Lipari

SOLUZIONI DEI PROBLEMI DEL CAPITOLO 3

1 In una architettura multiprocessore due processi operanti su processori diversi, P1 e P2, possono interagire sull'uso di una risorsa R contenuta nella memoria comune con il vincolo che tale risorsa sia utilizzata in mutua esclusione. Indicando con mutex il semaforo di mutua esclusione, inizializzato al valore 1, occorre garantire che l'esecuzione della wait e della signal, che operano sulla stessa struttura dati rappresentata dal semaforo mutex, da parte dei processi possa avvenire in modo mutuamente esclusivo. Si possono usare a questo scopo le operazioni lock(x) e unlock(x), che in maniera indivisibile (uso della TSL per la lock) operano sull'indicatore x posto a protezione del semaforo mutex. Si suppone che la x assuma il valore 0 se nessuna delle due primitive wait e signal è in esecuzione, mentre il valore 1 indica che una tra wait e signal è in esecuzione.
La prima istruzione del codice sia della wait che della signal deve essere pertanto la lock(x) e l'ultima la unlock(x).
Si ha pertanto:

```
wait(mutex):
        lock(x);
            if (mutex= = 0) < sospensione del processo nella coda di mutex>;
            else  mutex=0;
        unlock(x);


signal (mutex):
        lock(x);
            if(< non ci sono processi sospesi su mutex>) mutex=1;
            else <viene riattivato il primo processo sospeso su mutex>
        unlock(x);
```

2 In problemi di sincronizzazione complessi, in particolare quando si voglia realizzare una particolare politica di gestione delle risorse, la decisione di consentire ad un processo di proseguire l'accesso ad una risorsa condivisa può dipendere dal verificarsi di una particolare condizione detta *condizione di sincronizzazione* che tiene conto , in generale, della compatibilità tra lo stato della risorsa ed il tipo di richiesta di utilizzo da parte dei processi. Può quindi accadere che più processi siano simultaneamente bloccati durante l'accesso ad una risorsa condivisa, ciascuno in attesa che la propria condizione di sincronizzazione sia verificata. In seguito alla modifica dello stato della risorsa da parte di un processo, le condizioni di sincronizzazione di alcuni dei processi bloccati possano essere contemporaneamente verificate. In questi casi, se l'accesso alla risorsa deve essere mutuamente esclusivo, nasce il problema di quale processo attivare. Nel caso in cui si desideri realizzare una particolare politica di gestione della risorsa la scelta del processo da riattivare deve avvenire sulla base di algoritmi specificati risorsa per risorsa che definiscono un criterio di scelta tra i processi.
Nell'ipotesi che i processi, per i quali la condizione di sincronizzazione non è verificata, vengano bloccati su un semaforo riservato a questo scopo, nasce il problema di come scegliere il processo da riattivare. Si ricorda infatti che l'algoritmo di risveglio realizzato nella signal è l'algoritmo FIFO e quindi non è possibile realizzare una diversa politica di risveglio.
Il problema viene risolto con l'introduzione dei *semafori privati*. Un semaforo privato di un processo è caratterizzato dal fatto che solo quel processo può eseguire la wait su di esso mentre altri processi possono eseguire la signal. Il semaforo privato ha valore iniziale 0.

Qualora la condizione di sincronizzazione non sia verificata per un processo che vuole accedere ad una risorsa, questo viene sospeso sul suo semaforo privato tramite l'esecuzione di wait sul semaforo. Questo consente, in fase di riattivazione dei processi sospesi di scegliere il processo da riattivare eseguendo una signal sul suo semaforo privato.

3 Alla classe di sezioni critiche (A,B,..) vengono associate due variabili logiche libero1 e libero2 inizializzate al valore *false (0):*

```
int libero1=0;
int libero2=0;
```

La soluzione al problema èuò essere scritta nel seguente modo:

```
/* processo P1: */
    main()
    {       ...
            libero2=1;
            while (libero1!=0);
            <sezione critica B>;
            libero2=0;

            ...
    }
/* processo P2: */
    main()
    {       ...
            libero1=1;
            while (libero2!=0);
            <sezione critica A>;
            libero1=0;

            ...
    }
```

Si può facilmente verificare che la soluzione assicura che un solo processo alla volta può trovarsi in una delle sezioni critiche
Possono tuttavia presentarsi condizioni in cui, a seconda della velocità relativa dei processi, questi non possono entrare nella loro sezione critica, pur essendo tali sezioni libere (deadlock).
Si supponga infatti che:

to : P1 pone *libero1= 1;*
t1 : P2 pone *libero2 = 1;*

P1 e P2 ripetono indefinitamente l'esecuzione di while senza poter entrare nelle rispettive sezioni critiche.

4 Si supponga che su una risorsa R sia possibile eseguire due operazioni A e B e che ciascuno degli m processi P1,..,Pm abbia la necessità di operare su R tramite una delle precedenti operazioni.

In un ambiente a scambio di messaggi la risorsa R è locale al solo processo S, il server di R, ed i singoli processi dovranno richiedere a S di eseguire per suo conto su R l'operazione desiderata, A o B. Il processo server può quindi essere programmato nel seguente modo.

```
process S{
        ……
        while (true) {
                <riceve una richiesta di servizio da un cliente>;
                <esegue su R la funzione A(..) o B(..) richiesta>;
                <restituisce al cliente i risultati>;
                }
}
```

Si supponga che esistano due diverse porte locali al server da cui ricevere separatamente le varie richieste di servizio. Il problema che nasce ora è come si programma la prima istruzione del server
<riceve una richiesta di servizio da un cliente>?. In altri termini da quale porta il server decide di ricevere inizialmente? Nell'ipotesi di receive bloccante può infatti capitare che il server si blocchi su una porta vuota, mentre sull'altra porta ci sono messaggi. Per risolvere questo problema si può pensare di avere a disposizione, oltre alla receive bloccante, anche un'altra primitiva che si limiti a verificare se su una porta ci sono messaggi in arrivo senza bloccare il processo se non ce ne sono.
Questa soluzione presenta l'inconveniente dell'attesa attiva, cioè il processo server dovrebbe continuare a testare le porte.
Per evitare l'attesa attiva il processo server deve sospendersi in attesa di un messaggio. Su quale delle due porte? Il meccanismo di ricezione ideale per risolvere questo problema alla radice dovrebbe consentire al processo server di verificare la disponibilità di messaggi sulle due porte, abilitando la ricezione di un messaggio da uno qualunque delle due porte contenenti messaggi, se ce ne sono, o bloccando il processo in attesa che arrivi un messaggio, qualunque sia la porta sul quale arriva, quando nessuna porta contiene messaggi.
Molti linguaggi di alto livello hanno adottato un meccanismo di questo tipo utilizzando la notazione dei comandi con guardia per la quale si rinvia alla bibliografia [3].