Sistemi Operativi

Note agli Esercizi d'Esame del 22/1/2003

Descrivere come funzionano i sistemi RAID di livello 4 e di livello 5. Quali sono i vantaggi del livello 5 rispetto al livello 4? Motivare la risposta.

Nota: molti hanno risposto bene a questa domanda tranne per un particolare. Quando tentano di spiegare perchè un livello 5 è più efficiente di un livello 4, citano il fatto che nel livello 4, quando si vuole modificare un settore, è necessario rileggere tutti i drive per ricalcolare la parità. In effetti il libro su questo è un po' sviante.

In realtà non c'è nessun bisogno di rileggere tutti i drive in quanto basta leggere il settore che contiene i bit di parità e il settore che si vuole modificare: da questi due e dal settore nuovo è possibile calcolare il nuovo settore di parità. Quindi per modificare un settore su un livello 4 bastano due letture e due scritture (il settore di parità e il settore vero e proprio che si vuole scrivere).

Anche su un livello 5 non si può fare di meglio, sono sempre necessarie due letture e due scritture, ma c'è una differenza sostanziale: nel livello 4 il disco di parità è sempre lo stesso. Se scrivo N settori su disco, questi richiederanno N letture ed N scritture dei dischi dati, ed N letture ed N scritture del disco di parità. Mentre le prime N letture ed N scritture sono in media divise equamente tra tutti i dischi del RAID, le N letture ed N scritture che riguardano la parità sono tutte concentrate sullo stesso disco. Quest'ultimo quindi è sovraccarico di lavoro e rallenta l'esecuzione di tutto il sistema. Nel livello 5, invece, siccome la parità è distribuita tra tutti i dischi, anche le N letture ed N scritture per la parità sono divise equamente tra i dischi, e non c'è quindi un solo disco che viene sovraccaricato più degli altri e che funge da collo di bottiglia.

Spiegare sinteticamente il funzionamento delle primitive di comunicazione `send` e `receive` nel caso di comunicazione sincrona e nel caso di comunicazione asincrona.

Nota: nessuno ha risposto perfettamente a questo esercizio, probabilmente non mi sono spiegato bene a lezione, anche perchè questo argomento non si trova nel libro di testo. Segue una spiegazione.

Le primitive `send` e `receive` hanno la forma

```
send(destinatario,&messaggio);
receive(sorgente,&messaggio);
```

dove "messaggio" è una variabile contenete il messaggio che si vuole spedire o dove si vuole mettere il messaggio che si riceve. La send spedisce il messaggio al destinatario e la receive lo riceve dal sorgente. Tipicamente, destinatario e sorgente sono degli identificatori di processo, ma sono ammesse anche altre possibilità (vedi il caso delle "mailbox").

Facciamo un esepio di uso della send e recevice

```
void P()
{
   int i=2;
   send(Q,&i);
}

void Q()
{
   int j=1;
   receive(P,&j);
   printf("%d",j);
}
```

Il processo Q stamperà il numero 2, che è il valore che gli viene passato dal processo P.

La primitiva send può essere di due tipi: bloccante o non bloccante. La send è bloccante quando aspetta che il destinatario legga il messaggio che essa sta inviando con la corrispondente primitiva receive. Nel caso precedente, se la send è bloccante, il processo P non termina finchè Q non ha letto il messaggio che gli viene inviato. La send è non bloccante quando il messaggio viene comunque inviato anche se il processo destinatario non sta eseguendo in quel momento la receive: il processo mittente continua normalmente con l'esecuzione e il destinatario leggerà con calma il messaggio quando eseguirà la primitiva receive.

Anche la primitiva receive può essere bloccante o non bloccante ma il caso non bloccante è molto atipico e non lo consideriamo. La receive è bloccante quando aspetta che arrivi un messaggio prima di restituire il controllo al programma chiamante. Ad esempio, nell'esempio precedente, la printf viene eseguita solo dopo che il messaggio di P è arrivato. Noi considereremo solo receive bloccanti.

In generale ogni forma di comunicazione tra processi può essere di due tipi: sincrona o asincrona. La comunicazione è sincrona quando il mittente spedisce nello stesso momento in cui il destinatario riceve, ed asincrona se la spedizione e la ricezione possono avvenire in momenti diversi. Notare che questa distinzione si applica anche in altri contesti nel mondo reale. La comunicazione telefonica, ad esempio, è sincrona, perchè i due interlocutori devono essere contemporaneamente al telefono. La comunicazione tramite posta cartacea, invece, è asincrona, perchè il momento in cui il ricevente riceve una lettera è diverso da quello in cui il mittente l'ha spedita.

Nel caso di primitive send e receive, ad una comunicazione sincrona corrispondono una send e una receive di tipo bloccante. Infatti, quando questo accade, i due processi che comunicano devono eseguire contemporaneamente l'uno la primitiva send e l'altro la receive, altrimenti la comunicazione non avviene. Alla comunicazione asincrona, invece, corrisponde una send di tipo non bloccante.

In quali situazioni è possibile simulare un semaforo tramite un mutex? Motivare la risposta.

Nota: un mutex può essere visto essenzialmente come un semaforo. La differenza fondamentale è che mentre il semaforo può assumere qualsiasi valore intero positivo, il mutex può assumere solo i valori 0 e 1 (oppure "bloccato" e "non bloccato"). Il mutex può quindi simulare un semaforo in tutti quei casi in cui il semaforo, per il modo in cui è utilizzato, assume soltanto i valori 0 e 1.

Ma quali sono le situazioni in cui questo accade? Fondamentalmente ciò avviene quando il semaforo è usato per regolare la mutua esclusione. Supponiamo che i processi P e Q accedano e modifichino dei dati condivisi: per evitare modifiche inconsistenti dei dati, occorre fare in modo che un solo processo alla volta possa effetture questi accessi. Per far questo possiamo usare i semafori per proteggere le rispettive regioni critiche di P e Q:

```
semaforo s=1;

void P()
{
   .....
   down(&s)
   ...regione critica...
   up(&s)
   .....
}

void Q()
{
   .....
   down(&s)
   ...regione critica...
   up(&s)
   .....
}
```

In questo contesto, il seamforo s parte dal valore 1 e sia P che Q sono fuori dalle sezioni critiche. L'unica primitiva che è possibile eseguire su s è allora la down di P o di Q che entra nella sezione critica. Questo porta il valore del semaforo a 0. A questa down non potrà che seguire, prima o poi, la corrispondente up che riporta il valore a 1 (o sblocca l'altro processo che eventualmente si era bloccato su s). In ogni caso si vede che s può assumere solo i valori 0 e 1. Quindi possiamo sostituire il semaforo s con un mutex e scrivere

```
mutex s;

void P()
{
   .....
   lock(&s)
   ...regione critica...
   unlock(&s)
   .....
}
```

```
void Q()
{
   .....
   lock(&s)
   ...regione critica...
   unlock(&s)
   .....
}
```