

Elementi di Architettura e Sistemi Operativi

Bioinformatica - Tiziano Villa

1 Luglio 2013

Nome e Cognome:

Matricola:

Posta elettronica:

| problema | punti massimi | i tuoi punti |
|------------|---------------|--------------|
| problema 1 | 7 | |
| problema 2 | 7 | |
| problema 3 | 6 | |
| problema 4 | 10 | |
| totale | 30 | |

1. Si propone una nuova tecnica di sincronizzazione che si basa sul meccanismo del numero di coda in un negozio o ufficio (l'utente arriva e prende il numero da una macchinetta emettitrice e poi aspetta fino a che il suo numero e' chiamato, potendo controllare da uno schermo qual e' il numero servito correntemente).

In particolare, s'introducono le variabili *numero_servito* e *numero_arrivo*.

Per *numero_servito* s'intende una variabile intera rispetto a cui sono definite tre operazioni:

- (a) *inizializza_servito(numero_servito)*, azzera il valore di *numero_servito*;
- (b) *incrementa_servito(numero_servito)*, incrementa atomicamente il valore di *numero_servito*;
- (c) *aspetta_servito(numero_servito, valore)*, fa attendere il processo chiamante fino a che il valore di *numero_servito* e' diventato maggiore o uguale al *valore* in argomento.

Per *numero_arrivo* s'intende una variabile intera rispetto a cui sono definite due operazioni:

- (a) *inizializza_arrivo(numero_arrivo)*, azzera il valore di *numero_arrivo*;
- (b) *incrementa_arrivo(numero_arrivo)*, incrementa atomicamente il valore di *numero_arrivo* e restituisce il suo valore precedente (cioe' modella la macchinetta emettitrice che assegna all'utente il numero corrente e lo incrementa per il prossimo utente).

Supponendo di avere a disposizione le operazioni precedenti, si scrivano tre procedure *Inizializza_semaforo(sema, valore)*, *P(sema)*, *V(sema)*, che usano una variabile di tipo *numero_servito* e una di tipo *numero_arrivo* per realizzare un semaforo. *Inizializza_semaforo* assegna un valore iniziale al semaforo, *P* e *V* sono le classiche operazioni sui semafori. **S'illustri con chiarezza il codice proposto.**

Si assuma che *sema* sia un puntatore a una struttura che contiene *numero_servito* e *numero_arrivo*, essendo tale struttura definita come segue

```
typedef struct {  
    int numero_servito;  
    int numero_arrivo;  
} semaforo;
```

Traccia per lo studente.

```
Inizializza_semaforo(semaforo *sema, int valore) {
    int fittizio;
    inizializza_servito(sema->numero_servito);
    inizializza_arrivo(sema->numero_arrivo);

    /* si simula un utente fittizio per iniziare
       a contare gli utenti da 1 e non da 0 */
    fittizio = incrementa_arrivo(sema->numero_arrivo);

    /* si inizializza numero_servito a valore */
    while (valore-- > 0)
        incrementa_servito(sema->numero_servito))
}
```

```
P{?) {
    ?

    ? = ?;
    ?;
}
```

```
V{?) {

    ?;
}
```

Traccia di soluzione.

Osservazioni preliminari:

- Le variabili *numero_servito* e *numero_arrivo* possono soltanto essere incrementate, ma non possono essere lette. Si può ottenere soltanto il valore di *numero_arrivo* (tramite *incrementa_arrivo*, che restituisce il valore di *numero_arrivo* prima d'incrementarlo).
- La funzione *aspetta_servito* realizza una forma d'attesa e quindi ha a che fare con l'operazione semaforica *P*.
- La funzione *incrementa_servito* ha a che fare con l'operazione semaforica *V* perché incrementa una variabile su cui un processo può essere in attesa.

Per realizzare un semaforo, si userà *numero_servito* per contare quante *V* sono state eseguite sul semaforo, e *numero_arrivo* per contare quante *P* sono state eseguite sul semaforo. Quando un processo esegue *V* su un semaforo, incrementa il valore di *numero_servito*. Quando un processo esegue *P* su un semaforo, riceve il valore corrente di *numero_arrivo*, che indica al processo il suo numero d'ordine nella coda semaforica (e lo incrementa perché sia pronto per la prossima *P*); se $\text{numero_servito} < \text{numero_arrivo}$, il processo deve aspettare che *numero_servito* s'incrementi fino al valore di *numero_arrivo* ricevuto, altrimenti il processo può proseguire. Il valore del semaforo è $\text{numero_servito} - \text{numero_arrivo} + 1$.

```

Inizializza_semaforo(semaforo *sema, int valore) {
    int fittizio;
    inizializza_servito(sema->numero_servito);
    inizializza_arrivo(sema->numero_arrivo);

    /* si simula un utente fittizio per iniziare
       a contare gli utenti da 1 e non da 0 */
    fittizio = incrementa_arrivo(sema->numero_arrivo);

    /* si inizializza numero_servito a valore */
    while (valore-- > 0)
        incrementa_servito(sema->numero_servito))
}

P(semaforo *sema) {
    int numero_coda;

    numero_coda = incrementa_arrivo(sema->numero_arrivo);
    aspetta_servito(sema->numero_servito, numero_coda);
}

V(semaforo *sema) {
    incrementa_servito(sema->numero_servito);
}

```

Si noti che l'esecuzione di una *P* consiste in

- *incrementa_arrivo*: prende un nuovo biglietto dalla emettitrice (e incrementa tale numero per l'utente successivo)
- *aspetta_servito*: confronta tale numero con il valore di *numero_servito*, se tale numero è maggiore di *numero_servito* devo aspettare il mio turno (cioè ho un numero di coda maggiore di quello dell'utente servito adesso, per cui devo aspettare che arrivi il turno del mio numero).

2. Si consideri il seguente algoritmo di schedulazione con diritto di prelazione, e basato su priorit  variabili dinamicamente. I numeri di priorit  maggiori indicano una priorit  pi  alta. Quando un processo   in attesa del processore (nella coda dei processi pronti), la sua priorit  varia a un tasso α ; quando   in esecuzione, la sua priorit  varia a un tasso β . All'ingresso nella coda dei processi pronti, si attribuisce la priorit  0 a tutti i processi. I parametri α e β si possono impostare in modo da fornire algoritmi di schedulazione diversi.
- (a) Si descriva brevemente che cosa vuol dire schedulazione con diritto di prelazione.
 - (b) Si descriva l'algoritmo risultante da $\beta > \alpha > 0$.
 - (c) Si descriva l'algoritmo risultante da $\alpha < \beta < 0$.
 - (d) Si confrontino brevemente i pro e i contro dei due algoritmi.

Traccia.

Per rispondere ai quesiti b) e c), si simuli l'arrivo di 3 processi nella coda dei pronti con i due scenari contraddistinti dalle relazioni tra α e β indicate. Per il caso c) si ponga attenzione al fatto che i tassi di cambiamento delle priorit  sono negativi.

Traccia di soluzione.

Caso $\beta > \alpha > 0$. Il processo in esecuzione aumenta la sua priorita' piu' velocemente dei processi in attesa.

Il primo processo che arriva nella coda dei pronti *processo1* ha la priorita' definita a 0, acquisisce il processore, e poi incrementa la sua priorita' piu' in fretta degli altri processi *processo2* *processo3* ... che arrivassero nella coda dei pronti (anch'essi partirebbero da 0 e poi aumenterebbero la priorita' a un tasso $0 < \alpha < \beta$). Percio' il *processo1* continuerebbe a detenere il processore, essendo la sua priorita' sempre maggiore di quella dei processi *processo2*, *processo3*

In definitiva questa e' la politica di schedulazione FIFO (detta anche FCFS), cioe' chi arriva per primo nella coda dei pronti e' servito per primo.

Caso $\alpha < \beta < 0$. I processi in attesa decrementano la loro priorita' piu' velocemente del processo in esecuzione.

Il primo processo che arriva nella coda dei pronti *processo1* ha la priorita' definita a 0, acquisisce il processore, e poi decrementa la sua priorita' al tasso $\beta < 0$. Supponiamo ora che arrivi il processo *processo2*, ad esso sarebbe assegnata la priorita' $0 > \beta$ e quindi il *processo2* si prenderebbe il processore interrompendo l'esecuzione del processo *processo1* che sarebbe rimesso nella coda dei pronti. Non solo, il processo *processo2* continuerebbe a tenere il processore, perche' la sua priorita' diminuirebbe con tasso β meno in fretta di quella del processo *processo1* che diminuirebbe con tasso α (dato che $\alpha < \beta < 0$). Se poi arrivasse un terzo processo *processo3*, esso avrebbe la priorita' definita a 0 e percio' si prenderebbe il processore alle spese del processo *processo2* e continuerebbe a tenere il processore finche' gi servisse, mentre i processi *processo1* e *processo2* rimarrebbero nella coda dei pronti. Quando il processo *processo3* non avesse piu' bisogno del processore, sarebbe ripescato dalla coda dei pronti il processo *processo2* e solo al suo termine sarebbe ripreso il processo *processo1*.

In definitiva questa e' la politica di schedulazione LIFO, cioe' chi arriva per ultimo nella coda e' servito per primo.

FIFO: i lavori sono eseguiti nell'ordine di arrivo e non si esercita la prelazione; il tempo di risposta medio puo' essere grande se lavori brevi devono aspettare dietro quelli lunghi (effetto a convoglio). Puo' portare a sovrapposizione poco efficiente dell'uso del processore e delle operazioni di ingresso/uscita.

LIFO: migliora il tempo di risposta dei nuovi processi, ma può precludere all'infinito il processore ai processi vecchi (affamare, "starvation").

3. Un elaboratore fornisce ai propri utenti uno spazio di memoria virtuale di 2^{32} byte (abbreviato come B). L'elaboratore dispone di 2^{18} B di memoria fisica. La memoria virtuale e' realizzata tramite paginazione e la dimensione di una pagina e' di 4096 B. Un processo utente genera l'indirizzo virtuale 11123456. Prima si descriva brevemente il concetto di memoria virtuale e la sua utilita'. Poi si spieghi come il sistema determina la locazione fisica a partire da un indirizzo virtuale, e lo si esemplifichi nel caso dello specifico indirizzo virtuale generato, individuando i vari campi degl'indirizzi virtuale e fisico.

Traccia di soluzione.

Per la memoria virtuale si consulti il capitolo dedicato del libro di testo.

Il meccanismo della paginazione e' spiegato nel capitolo sulla memoria centrale.

L'indirizzo virtuale in binario e':

0001 0001 0001 0010 0011 0100 0101 0110

Poiche' la dimensione di una pagina e' $4096 = 2^{12}$ B, la dimensione della tavola delle pagine e' 2^{20} elementi. Percio' le 12 cifre binarie meno significative 0100 0101 0110 sono usate come indice nella pagina, mentre le rimanenti 20 cifre binarie piu' significative 0001 0001 0001 0010 0011 sono usate come indice nella tavola delle pagine.

Dato che la memoria fisica ha 2^{18} B, ogni indirizzo fisico ha 18 cifre binarie di cui le 12 meno significative sono l'indice del byte nella pagina, e le 6 piu' significative indicizzano la pagina, percio' ogni elemento della tavola delle pagine contiene 6 cifre binarie. La memoria fisica ha 2^6 pagine.

4. Si progetti un circuito sequenziale che realizza la seguente specifica:

- Ci sono un segnale binario d'ingresso X e un segnale binario d'uscita Z .
 - L'uscita Z vale 1 se su X si e' presentata una successione di un numero pari di 0 seguiti da un numero dispari di 1; l'uscita Z torna a 0 quando il numero di 1 diventa pari (per tornare a 1 se torna dispari) o si ripresenta in ingresso uno 0.
- (a) Si disegni il grafo delle transizioni di una macchina a stati finiti di tipo Mealy che corrisponde alla specifica. S'indichi lo stato iniziale.

- (b) Si minimizzi il numero degli stati della macchina proposta, applicando l'algoritmo di minimizzazione degli stati.

- (c) Si scriva la tavola delle transizioni con gli stati futuri e le uscite e la si codifichi.

- (d) Supponendo di usare bistabili di tipo D, si derivino le equazioni minimizzate di eccitazione degl'ingressi dei bistabili e le equazioni minimizzate delle uscite.

- (e) Si realizzi il circuito sequenziale corrispondente con bistabili di tipo D campionati sul fronte di salita, invertitori e porte NAND. Si etichettino con chiarezza i segnali.