

Elementi di Architettura e Sistemi Operativi

Bioinformatica - Tiziano Villa

20 Settembre 2013

Nome e Cognome:

Matricola:

Posta elettronica:

problema	punti massimi	i tuoi punti
problema 1	8	
problema 2	5	
problema 3	7	
problema 4	10	
totale	30	

1. Si consideri il seguente codice incompleto per scrivere le primitive di un semaforo P (*Wait*) and V (*Signal*) per un sistema multiprocessore simmetrico.

```
typedef struct {
    int t; /* inizializza a 0 */
    int count; /* inizializza come appropriato */
    queue q;
} semaphore;

P(semaphore *S) {
    disabilita le interruzioni sul processore in uso;
    while (TestAndSet(S->t) ?) /* non fare nulla */;
    if (S->count ?) {
        S->count ?;
        S->t ?;
        abilita le interruzioni;
        return;
    }
    aggiungi il processo alla coda S->q;
    S->t ?;
    abilita le interruzioni;
    Redispatch; /* scegli il prossimo processo */
}

V(semaphore *S) {
    disabilita le interruzioni sul processore in uso;
    while (TestAndSet(S->t) ?) /* non fare nulla */;
    S->count ?;
    if (la coda S->q non e' vuota)
        risveglia il primo processo di S->q;
    S->t ?;
    abilita le interruzioni;
}
```

- (a) Si descriva brevemente che cos'è un semaforo e si mostri lo pseudo-codice della definizione classica delle operazioni P e V .

Traccia di soluzione.

Un semaforo è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: P (*Wait*) and V (*Signal*).

Le definizioni classiche di *Wait* e *Signal* in pseudo-codice sono le seguenti:

```
Wait(S) {
    while (S <= 0)
        ;
    S--;
}
```

```
Signal(S) {
    S++;
}
```

Realizzazione senza attesa attiva

```
typedef struct {
    int valore;
    struct processo *lista;
} semaforo;
```

```
Wait(semaforo *S) {
    S->valore--;
    if (S->valore < 0) {
        aggiungi questo process a S->lista;
        block;
    }
}
```

```
Signal(semaforo *S) {
    S->valore++;
    if (S->valore <= 0) {
        toglì un process P da S->lista;
        wakeup(P);
    }
}
```

```
}  
}
```

In questa realizzazione il valore del semaforo puo' diventare negativo per contare il numero dei processi in attesa (in altre realizzazioni il valore non puo' diventare negativo).

- (b) Si descriva l'operazione di `TestAndSet (i)` e come e' utilizzata nella sincronizzazione dei processi.

Traccia per lo studente.

`TestAndSet (i)` assegna alla variabile `i` in argomento il valore 1, ma restituisce il valore precedente della stessa variabile argomento.

Traccia di soluzione.

`TestAndSet (i)` assegna alla variabile `i` in argomento il valore 1, ma restituisce il valore precedente della stessa variabile argomento. Poi si usa un normale assegnamento per azzerare di nuovo il valore dell'argomento. Si puo' utilizzare `TestAndSet` per la mutua esclusione: e' come un semaforo binario al rovescio, tranne che non richiede attesa. Il valore 1 significa che qualcun altro sta gia' usando la sezione critica, il valore 0 significa che essa e' libera e si puo' procedere. `TestAndSet` per definizione previene la situazione che due processi eseguano contemporaneamente la transizione da 0 a 1.

Quando si usa `TestAndSet` per realizzare un semaforo, si definisce una variabile intera (azzerata all'inizio) relativa a `TestAndSet` (per noi `t`), oltre alla variabile intera del semaforo (per noi `count`), e alla coda dei processi in attesa.

Di solito `TestAndSet` non e' disponibile nei linguaggi di programmazione ad alto livello, ma si deve ricorrere a una procedura in linguaggio assembler.

- (c) Si completi il codice iniziale sostituendo i "?" con le parti mancanti, e lo si commenti spiegando perché realizza correttamente le primitive di un semaforo.

Traccia di soluzione.

```
typedef struct {
    int t; /* inizializza a 0 */
    int count; /* inizializza come appropriato */
    queue q;
} semaphore;

P(semaphore *S) {
    disabilita le interruzioni sul processore in uso;
    while (TestAndSet(S->t) != 0) /* non fare nulla */;
    if (S->count > 0) {
        S->count -=1;
        S->t = 0;
        abilita le interruzioni;
        return;
    }
    aggiungi il processo alla coda S->q;
    S->t = 0;
    abilita le interruzioni;
    Redispatch; /* scegli il prossimo processo */
}

V(semaphore *S) {
    disabilita le interruzioni sul processore in uso;
    while (TestAndSet(S->t) != 0) /* non fare nulla */;
    S->count +=1;
    if (la coda S->q non e' vuota)
        risveglia il primo processo di S->q;
    S->t = 0;
    abilita le interruzioni;
}
```

(d) Si spieghi come funziona il codice precedente.

Puo' esserci dell'attesa attiva ?

Traccia di soluzione.

Nei sistemi uniprocessore, la disabilitazione delle interruzioni e' la tecnica piu' comune per garantire che una sequenza di operazioni abbia luogo senza interferenze, cioe' le interruzioni sono disabilitate all'ingresso della sezione critica e riabilite all'uscita. Invece in un sistema multiprocessore questa via e' preclusa, perche' anche se un processore potesse disabilitare le interruzioni di un altro (il che in generale non si puo' fare) il secondo processore potrebbe ancora eseguire un processo e violare (anche inavvertitamente) dei vincoli di mutua esclusione. La soluzione e' usare un'operazione atomica come `TestAndSet` (o altre simili) per controllare l'accesso a un semaforo tramite una variabile di guardia che e' appunto manipolata mediante l'istruzione `TestAndSet`.

Tale istruzione permette a un processore di leggere e scrivere atomicamente una locazione di memoria, prevenendo collisioni tra piu' processori. Si noti che prima di eseguire un'istruzione di tipo `TestAndSet`, il processore dovrebbe ancora disabilitare le sue interruzioni, come in un sistema uniprocessore, per una questione di efficienza.

L'istruzione `TestAndSet (S->t)` permette di realizzare la mutua esclusione, poiche' il suo argomento `S->t` e' 0 all'inizio e dopo la prima esecuzione diventa 1, bloccando processi successivi, finche' esso e' ri-azzerato.

Un problema di questa soluzione e' che ci puo' essere attesa attiva in P e in V (per il ciclo `while TestAndSet (S->t) != 0`). Tuttavia si tratta di solito di un'attesa attiva di breve durata, perche' non e' relativa al tempo in cui un'applicazione sta nella sezione critica, ma esclusivamente al (solitamente) breve tempo necessario a ottenere il via libera per aggiornare una variabile semaforica.

Sommario.

Si noti la differenza tra `count` che e' un semaforo che conta la disponibilita' della risorsa semaforica, e `t` che non e' un semaforo, ma una guardia che protegge il valore del semaforo.

In altri termini `t` controlla che un solo processo per volta verifichi la disponibilita' del semaforo e si registri nella coda se ora indisponibile,

mentre `count` denota quante unita' della variabile semaforica sono disponibili e quindi se un processo puo' entrare nella sezione critica o deve aspettare in coda il suo turno. Detto ancora una volta, τ e' come l'ufficio di accettazione di un servizio con un solo sportello, e controlla che un solo utente per volta si presenti allo sportello, mentre `count` e' il numero di prenotazione del servizio che lo sportello assegna all'utente (che aspettera' il suo turno per il servizio nella coda `queue` del semaforo).

Si ha attesa attiva rispetto a τ tra i processi che vogliono registrarsi con il semaforo, ma non si ha attesa attiva rispetto alla sezione critica salvaguardata dal semaforo (perche' se essa e' indisponibile, il processo richiedente va a dormire sino al suo risveglio).

La prima attesa attiva e' breve (il tempo per verificare lo stato del semaforo); la seconda attesa attiva sarebbe potenzialmente molto lunga (il tempo richiesto per lavorare nella sezione critica). Riprendendo la similitudine dell'ufficio accettazione, il tempo di stare in coda per arrivare allo sportello ed avere il numero del servizio richiesto e' probabilmente breve rispetto al tempo in cui terro' occupato il servizio richiesto quando sara' disponibile per me.

(e) (Domanda extra-credito: +2 punti)

Perche' si disabilitano le interruzioni e al contempo si usa `TestAndSet` ? Non basta il secondo meccanismo per assicurare la correttezza della soluzione ?

Traccia di soluzione

Si cominci con il notare che si disabilitano le interruzioni solo sul processore dove gira il processo che esegue le operazioni P o V.

Non e' indispensabile disabilitare le interruzioni su tale processore, poiche' `TestAndSet` e' sufficiente per garantire la mutua esclusione; ma e' utile disabilitarle per un obiettivo di efficienza. Se infatti non si disabilitassero le interruzioni, potrebbe succedere che un processo P1 eseguisse `TestAndSet` e poi fosse interrotto (ad esempio perche' e' scaduto il suo quanto di tempo) da molti altri processi prima di poter girare di nuovo. Durante tutto questo tempo la variabile t di guardia rimarrebbe a 1, percio' qualsiasi altro processo sullo stesso processore che cercasse di eseguire una P o V consumerebbe i suoi cicli macchina in un'attesa occupata. Anche i processi sugli altri processori che eseguissero P o V sprecherebbero cicli macchina in attesa occupata in attesa che si sbloccasse la situazione sul primo processore (e questo e' facilitato dal fatto che sul primo processore so disabilitano le interruzioni da parte degli altri processi del medesimo processore). Disabilitando e riabilitando le interruzioni si minimizzano (ma non si eliminano) i tempi di attesa occupata. Si noti che t non e' un semaforo, ma una guardia che protegge il valore del semaforo.

In definitiva, la disabilitazione/riabilitazione delle interruzioni e' il meccanismo usato per uniprocessori e si utilizza anche per multiprocessori per ragioni di efficienza, mentre `TestAndSet` e' necessario per multiprocessori poiche' si assume che non si possono disabilitare le interruzioni sui processori diversi da quello su cui gira il processo che esegue P o V.

2. Lo schedulatore tradizionale di UNIX ricalcola le priorit  dei processi una volta al secondo. Esse sono calcolate come l'inverso dei *numeri di priorit * definiti dalla seguente equazione:

$$\text{numero di priorit } = (\text{uso recente del processore} / 2) + \text{base}$$

dove $\text{base} = 60$, e *uso recente del processore* quantifica l'uso del processore da parte del processo dall'ultima determinazione delle priorit . In conclusione, piu' alto   il *numero di priorit * di un processo, piu' bassa   la sua priorit .

S'ipotizzi che l'*uso recente del processore* per il processo $P1$ sia 40, per il processo $P2$ sia 18, per il processo $P3$ sia 10. Quali saranno le nuove priorit  di questi tre processi ?

Per conseguenza, la priorit  relativa di un processo con uso intensivo del processore aumenta o diminuisce ?

Traccia di soluzione.

I nuovi *numeri di priorit * per i tre processi calcolati dalla formula sono rispettivamente: $(40/2 + 60) = 80 > (18/2 + 60) = 69 > (10/2 + 60) = 65$, e le nuove priorit  sono gl'inversi $(1/80 < 1/69 < 1/65)$.

La priorit  relativa di un processo con uso intensivo del processore diminuisce.

3. (a) Si spieghi il meccanismo della segmentazione nella memoria centrale.

Traccia di soluzione.

Si veda la sezione relativa nel libro di testo.

- (b) Si confronti la paginazione con la segmentazione rispetto alla quantità di memoria richiesta dalle strutture di traduzione degli indirizzi da logici a fisici.

Traccia di soluzione.

La paginazione richiede più memoria per le strutture di traduzione. La segmentazione richiede solo due registri per segmento, uno per la base e l'altro per la dimensione del segmento. La paginazione richiede un elemento per ogni pagina, per memorizzare l'indirizzo della pagina fisica.

(c) Assumendo la dimensione di pagina di 1 KB quali sono i numeri di pagina e gli scostamenti per i seguenti indirizzi (si presti attenzione se i numeri sono indicati in base decimale o esadecimale):

- i. 899_{10}
- ii. 23456_{10}
- iii. $3F244_{16}$
- iv. $0017C_{16}$

Traccia di soluzione.

Si noti che $1024_{10} = 400_{16}$.

- i. 899: pagina = 0, scostamento = 899
($899 \% 1024 = 899$; $899 / 1024 = 0$)
- ii. 23456: pagina = 22, scostamento = 928
($23456 \% 1024 = 928$; $23456 / 1024 = 22$)
- iii. $3F244_{16}$: pagina = FC_{16} , scostamento = 244_{16}
($3F244_{16} \% 400_{16} = 244_{16}$; $3F244_{16} / 400_{16} = FC_{16}$)
[per verificare si noti che $3F244_{16} = 258628_{10}$, $FC_{16} = 252_{10}$, $244_{16} = 580_{10}$]
- iv. $0017C_{16}$: pagina = 0_{16} , scostamento = $17C_{16}$
($17C_{16} \% 400_{16} = 17C_{16}$; $17C_{16} / 400_{16} = 0_{16}$)
[per una verifica si noti che $3F244_{16} = 258628_{10}$, $FC_{16} = 252_{10}$, $244_{16} = 580_{10}$]

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.

E' ragionevole l'ipotesi che per avere un numero pari di zeri servano almeno due zeri.

- (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.

Traccia di soluzione.

Si vedano le sezioni 6.3.1 e 6.4.1 nel libro di testo di Progettazione Digitale.

- (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.