Sistemi Operativi (modulo II / B) CON SOLUZIONE 5 settembre 2019

Esercizio 1

La funzione `pop()` legge un elemento da uno stack di interi positivi. Se lo stack è vuoto ritorna −1. Lo stack è realizzato con un array `s[]` di interi e un indice p, il cui valore corrisponde al numero di elementi nello stack. Ad esempio se p è zero lo stack è vuoto, se p è 2 ci sono due elementi: `s[0]` e `s[1]`.

```
int pop() {
    int tmp;
    P(mutex);
    if (p==0) {
        V(mutex);
        return -1;

    } else {

        p = p-1;

        tmp = s[p];
        V(mutex);
        return(tmp);
    }
}
```

La funzione pop() viene utilizzata di più thread concorrenti.

1. Illustrare almeno due problemi di interferenza causati dall'esecuzione concorrente di più letture dallo stack: Un primo problema si verifica se lo stack contiene un elemento (p==1), due thread eseguono una pop() e il primo viene interrotto dopo aver verificato il valore di p e subito prima di decrementarlo (all'inizio del ramo else). In tale caso il secondo thread troverà ancora il valore di p uguale a 1 e entrerà anch'esso nel ramo else. Come risultato uno dei due thread restituirà, erroneamente, il valore s[-1].
   Un secondo problema si verifica se un thread viene interrotto durante il decremento di p. Il valore di p, in tal caso, potrebbe essere stato caricato su un registro e decrementato ma non ancora salvato in memoria. Se, a questo punto, un secondo thread esegue il decremento di p, questo verrà 'sovrascritto' dal primo. Ad esempio si consideri p=2. Il primo thread carica 2 nel registro r1 e lo decrementa. Il secondo thread legge 2 dalla memoria, decrementa, e salva 1 in p. Quando il controllo torna al primo thread, il valore 1 nel registro r1 viene salvato in p e uno dei decrementi è quindi perduto.
   Il terzo problema si verifica se un thread viene interrotto dopo il decremento di p e prima della lettura del valore s[p]. In questo caso, infatti, un secondo thread potrebbe modificare il valore di p causando la lettura, da parte del primo thread, di un valore diverso da quello atteso.
2. Come si possono risolvere i problemi sopra indicati utilizzando i semafori? Aggiungere le relative P e V (wait e post) all'interno del codice, indicando l'inizializzazione dei semafori. Spiegare come i problemi di interferenza discussi al punto 1 vengono evitati dai semafori: L'utilizzo di un semaforo mutex, inizializzato a 1, permette di eseguire la funzione pop() in modo esclusivo (un solo thread alla volta). La sezione critica viene acquisita all'inizio dell'esecuzione e rilasciata subito prima del ritorno del valore. In questo modo il primo thread porta il semaforo a zero e eventuali altri thread rimarranno bloccati finché il primo thread non completa l'esecuzione. Notare come sia fondamentale copiare il valore di s[p] in una variabile temporanea locale (non condivisa). Se si eseguisse return(s[p]) la terza interferenza potrebbe verificarsi anche in presenza dei semafori.

Esercizio 2

Si considerino 3 processi `P1`, `P2` e `P3` che utilizzano 3 file `f1`, `f2` e `f3` in scrittura (`w`). La open in scrittura è bloccante se il file è già in uso (aperto) da un altro processo.

```
process P1 {
    open(f1,"w");
    open(f2,"w");
    open(f3,"w");
    ...
    close(f1);
    close(f2);
    close(f3);
}
```

```
process P2 {
    open(f2,"w");
    open(f3,"w");
    ...
    close(f2);
    close(f3);
}
```

```
process P3 {
    open(f3,"w");
    open(f2,"w");
    ...
    close(f3);
    close(f2);
}
```

1. Mostrare un'esecuzione in cui solamente `P2` e `P3` vanno in stallo: P1 apre f1, f2 e f3 li utilizza e poi li chiude. Successivamente P2 apre f2 e P3 apre f3. Quando P2 prova ad aprire f3 e P3 prova ad aprire f2 i due processi vanno in attesa circolare irrisolvibile (stallo).

2. Mostrare un'esecuzione in cui `P1`, `P2` e `P3` vanno in stallo: P2 apre f2 e P3 apre f3. Quando P2 prova ad aprire f3 e P3 prova ad aprire f2 i due processi vanno in attesa circolare irrisolvibile (stallo). A questo punto se P1 apre f1 e poi prova ad aprire f2 anch'esso va in stallo perché attende un evento che solo un altro processo in stallo (P2) può causare.

3. Mostrare come sia possibile prevenire lo stallo tramite allocazione gerarchica, effettuando una semplice modifica del codice di uno dei processi: Se ordiniamo le risorse f1 < f2 < f3, seguendo il principio dell'allocazione gerarchica, possiamo imporre che i processi aprano file solo se sono maggiori di quelli che hanno già aperto. In particolare P3 violerebbe questo principio aprendo f3 e poi f2. È quindi sufficiente modificare il codice di P3 come segue:

   ```
   process P3 {
       open(f2,"w");
       open(f3,"w");
       ...
       close(f2);
       close(f3);
   }
   ```

   questo garantisce che non si formi mai attesa circolare, prevenendo ogni situazione di stallo.