

Semafori

G. Lettieri

4 Maggio 2021

Nel sistema didattico tutti i processi utente condividono le sezioni `.text`, `.data`, `.bss` del modulo utente, e lo heap. Ciascun processo ha invece una sua pila utente privata, inaccessibile agli altri processi. Questo tipo di condivisione può essere ottenuta evitando di rimpiazzare la parte di memoria condivisa quando si salta da un processo ad un altro.

Un sistema del genere ha senso se i processi appartengono tutti allo stesso utente e fanno parte di un'unica applicazione, che l'utente ha deciso di strutturare in più attività concorrenti. Da ora in poi ci limiteremo a considerare solo questo caso.

L'utente che scrive una applicazione strutturata su più processi concorrenti deve affrontare dei problemi, che in parte abbiamo già visto, perché sono molto simili a quelli già affrontati a livello sistema.

In particolare, anche l'utente deve affrontare il problema dell'*interferenza*. Mentre un processo sta eseguendo delle modifiche su una struttura dati comune, un altro processo potrebbe inserirsi e cominciare anche lui a modificare la *stessa* struttura dati. Se l'utente non scrive il codice con attenzione, questo può causare malfunzionamenti, come abbiamo visto.

Si noti che nel codice di sistema abbiamo risolto il problema ricorrendo all'atomicità, realizzata disabilitando le interruzioni mentre è in esecuzione codice di sistema. Qui non possiamo fare la stessa cosa, in quanto le interruzioni devono restare abilitate mentre è in esecuzione il codice utente (altrimenti non riusciamo a realizzare un sistema multiprogrammato), e non possiamo dare all'utente la possibilità di disabilitare e riabilitare le interruzioni a suo piacimento, perché potrebbe disabilitarle e non riabilitarle più.

L'atomicità è un caso estremo di *mutua esclusione*. Con questa locuzione ci si riferisce alla proprietà secondo la quale alcune operazioni non possono mescolarsi tra loro (si escludono a vicenda). Quello che vogliamo fare è fornire al nostro utente delle primitive tramite le quali possa realizzare la mutua esclusione delle procedure che accedono alle sue strutture dati, senza però compromettere il funzionamento di tutto il sistema.

Esaminiamo ora il problema, diverso, della *sincronizzazione* tra i processi. Nello strutturare la sua applicazione, l'utente può aver bisogno di fare in modo che una certa azione *B* avvenga sempre dopo una certa azione *A*. Se però le due azioni sono svolte da processi diversi, l'utente ha il problema di non poter prevedere quando questi due processi svolgeranno le loro azioni. Un caso comune

è quello in cui un processo produce dei dati e li scrive in un buffer intermedio, da cui un altro processo li preleva per svolgere ulteriori elaborazioni. Normalmente questi due processi sono ciclici, con il produttore che produce continuamente nuovi dati e il consumatore che li preleva continuamente. Il buffer è una zona di memoria che può contenere un numero limitato di dati—supponiamo uno solo, per semplicità. La scrittura di un nuovo dato sovrascrive dunque il vecchio. L'utente vorrebbe poter garantire che il produttore non possa sovrascrivere un dato che il consumatore non ha ancora letto, e che il consumatore non legga mai due volte lo stesso dato.

Si noti come i problemi di sincronizzazione siano diversi da quelli di mutua esclusione: nella sincronizzazione vogliamo garantire un ordinamento tra alcune azioni, che deve essere sempre rispettato. Non è così nella mutua esclusione. Si pensi a due processi, siano P_1 e P_2 , che vogliono inserire un elemento nella stessa lista. L'unica cosa che non vogliamo è che i due inserimenti vengano tentati contemporaneamente. Se, per esempio, P_1 ha iniziato un inserimento, P_2 deve aspettare che P_1 abbia finito prima di iniziare il suo. Questa *sembra* la stessa cosa della sincronizzazione, ma dobbiamo tenere presente che ci va bene anche il contrario: se P_2 arriva prima e comincia il suo inserimento, è P_1 che deve aspettare che P_2 abbia finito. Inoltre, nessuno dei due deve aspettare niente, se quando vuole fare l'inserimento l'altro non sta usando la lista. Non stiamo dunque stabilendo un ordinamento tra le operazioni di P_1 e P_2 .

1 Scatole e gettoni

Per risolvere i problemi di mutua esclusione e sincronizzazione, supponiamo di avere delle scatole che possono contenere degli oggetti, tutti uguali, che chiamiamo gettoni. Su queste scatole possono essere eseguite solo due operazioni:

- inserire un gettone—non è necessario che il gettone sia stato precedentemente preso da una scatola;
- prendere un gettone—se non ce ne sono, bisogna aspettare che qualcuno ne inserisca uno, *senza poter fare nient'altro*.

Supponiamo di dover risolvere un problema di mutua esclusione: abbiamo più persone, P_1, P_2, \dots, P_n , e un corrispondente numero di azioni A_1, A_2, \dots, A_n che queste persone devono compiere. Vogliamo che le azioni non possano mai essere eseguite contemporaneamente. È sufficiente avere una scatola che inizialmente contiene un solo gettone e imporre la regola che solo chi ha il gettone può compiere una delle azioni: chiunque voglia agire, dunque, deve prima prendere un gettone e poi rimetterlo nella scatola quando ha finito. Dal momento che c'è un solo gettone, non è possibile che ci siano due azioni contemporaneamente in corso. Se qualcuno vuole iniziare una azione mentre ne è in corso un'altra, troverà la scatola vuota e dovrà aspettare che l'altro abbia finito.

Si noti che, mentre è in corso una azione, supponiamo da parte di P_1 , tante persone potrebbero volerne iniziare un'altra. Tutte queste dovranno aspettare.

Quando P_1 avrà finito poserà il suo gettone, che verrà preso da una delle persone in attesa, sia P_2 . Quando P_2 avrà a sua volta finito, il gettone passerà ad un'altra persona ancora, e così via. Si noti anche che la soluzione richiede una completa collaborazione da parte di tutti i partecipanti: se qualcuno compie una azione senza prendere il gettone, o si dimentica di rimetterlo a posto dopo aver finito, il sistema non funziona.

Vediamo ora come risolvere un problema di sincronizzazione. Abbiamo due persone, P_a che deve compiere l'azione A e P_b che deve compiere l'azione B . Vogliamo che l'azione B sia eseguita sempre dopo l'azione A . È sufficiente prevedere una scatola inizialmente vuota. Vogliamo che la presenza di un gettone nella scatola rappresenti il fatto che A è stata eseguita e B ancora no. Operativamente, *dopo* aver eseguito l'azione A , P_a deve lasciare un gettone nella scatola; *prima* di eseguire l'azione B , P_b deve prendere un gettone dalla scatola. Se P_a arriva per primo alla scatola, lascia il gettone e poi P_b potrà prenderlo e proseguire. Se invece arriva per primo P_b , trova la scatola vuota e deve aspettare che P_a inserisca un gettone. In entrambi i casi l'azione B non potrà partire prima che sia finita l'azione A .

In generale, per utilizzare le scatole di gettoni per risolvere un problema particolare, bisogna specificare quante scatole servono e quanti gettoni contengono inizialmente, quindi stabilire delle regole su chi deve inserire/estrarre gettoni, e quando. Il problema si può considerare risolto solo se le regole sono scelte bene e tutti le rispettano.

1.1 Esercizio

Come risolvere il problema del produttore e consumatore? Il produttore P invia dei messaggi al consumatore C tramite una lavagna. Per scrivere un nuovo messaggio cancella il precedente. Inoltre, non c'è modo di distinguere un messaggio nuovo da uno vecchio semplicemente guardandoli. Come fare per garantire che C legga tutti i messaggi di P e non legga mai due volte lo stesso messaggio?

2 Semafori

Nel nostro sistema, forniremo agli utenti l'astrazione delle scatole di gettoni, chiamate però, per motivi storici, *semafori*. Ogni semaforo è identificato da un numero. Forniamo le seguenti primitive di sistema:

- `natl sem_ini(natl v)`: crea un nuovo semaforo, che inizialmente contiene v gettoni, e ne restituisce l'identificatore (0xFFFFFFFF se non è stato possibile crearlo);
- `void sem_wait(natl sem)`: prende un gettone dal semaforo numero `sem`; blocca il processo se il semaforo è vuoto;
- `void sem_signal(natl sem)`: inserisce un gettone nel semaforo `sem`; risveglia uno dei processi bloccati in attesa di un gettone, se ve ne sono.

Si noti che, nonostante il nome, la primitiva `sem_wait()` non causa necessariamente una attesa con blocco del processo in esecuzione: se la scatola contiene già un gettone, il processo lo prende e va avanti senza bloccarsi.

Per semplicità, non diamo la possibilità di distruggere semafori. Dal momento che abbiamo deciso di supportare solo il caso di un singolo utente che esegue una applicazione multi-processo, assumiamo che tutti i semafori creati servano fino al termine dell'applicazione.

2.1 Mutua esclusione

Il problema della mutua esclusione tra le azioni A_1, A_2, \dots, A_n dei processi P_1, P_2, \dots, P_n può essere risolto come abbiamo visto nella Sezione 1. Prima che i processi siano stati attivati, l'utente deve creare un semaforo inizializzato ad 1:

```
nat1 mutex = sem_ini(1);
```

Quindi, tutte le azioni A_i , $1 \leq i \leq n$, devono essere protette dal semaforo:

```
Pi:    sem_wait(mutex);
        Ai;
        sem_signal(mutex);
```

Se, per esempio, le azioni A_i sono tutte invocazioni di funzioni membro di una classe su uno stesso oggetto, è sufficiente aggiungere l'identificatore del semaforo ai membri della classe, creare il semaforo nel costruttore, quindi aggiungere le chiamate `sem_wait()` e `sem_signal()` ad ogni funzione membro. In questo modo tutte le operazioni sullo stesso oggetto verranno eseguite in mutua esclusione. Se chiamiamo “libero” il semaforo, possiamo leggere l'operazione `sem_wait(libero)` come “attendi che l'oggetto sia libero (nessun altro lo sta usando)” e `sem_signal(libero)` come “segnala che ora l'oggetto è libero”.

2.2 Sincronizzazione

Il problema della sincronizzazione tra due processi— P_a che deve compiere l'azione A e P_b che deve compiere l'azione B —può essere risolto creando, prima che i processi siano attivati, un semaforo inizializzato a zero:

```
nat1 sync = sem_ini(0);
```

Quindi, i due processi devono essere codificati nel seguente modo:

```
Pa:    A;
        sem_signal(sync);

Pb:    sem_wait(sync);
        B;
```

Se P_b sta aspettando il verificarsi di una condizione, inizialmente falsa e resa vera dall'azione A , è utile dare al semaforo un nome x che ricordi questa condizione

(per esempio, `nuovo_messaggio`). Allora l'azione di P_b si legge come “aspetto che la condizione x sia vera” e l'azione di P_a si legge come “segnalo che la condizione x ora è vera”. Anche in questo caso non è detto che la `sem_wait()` debba sempre aspettare (bloccare il processo): se P_a completa l'azione A ed esegue la `sem_signal()` prima che P_b arrivi alla sua `sem_wait()`, P_b troverà il gettone già nella scatola e potrà prenderlo senza dover aspettare niente.

2.3 Realizzazione dei semafori

Per realizzare i semafori prevediamo la seguente struttura dati, definita nel codice di sistema:

```
struct des_sem {
    int counter;
    des_proc *pointer;
};
```

Il campo `counter` conta i gettoni contenuti nel semaforo, se maggiore o uguale a zero. Permettiamo al campo `counter` di scendere anche sotto zero. In questo modo la `sem_wait()` può decrementarlo in ogni caso. Se è negativo, il suo valore assoluto indica quanti processi sono in attesa di un gettone. I processi in attesa sono inseriti nella lista `pointer`. Queste liste realizzano le code dei processi bloccati in attesa di un “evento”. L'evento non è altro che l'inserimento di un gettone nel semaforo. I processi bloccati su un semaforo verranno rimessi in coda pronti quando riceveranno un gettone, per effetto di un altro processo che invoca `sem_signal()` sullo stesso semaforo. La primitiva `sem_signal()` deve incrementare il numero di gettoni e, se ci sono processi nella lista `pointer`, estrarne uno e inserirlo in coda pronti. Si noti che abbiamo deciso di realizzare un sistema in cui ogni processo ha una priorità, e ora dobbiamo garantire che in esecuzione ci sia sempre il processo che ha maggiore priorità tra tutti quelli pronti. Se la `sem_signal()` inserisce un processo in coda pronti, deve anche controllare che questo processo non abbia priorità maggiore di quello attualmente in esecuzione (quello che ha invocato la `sem_signal()`). In tal caso, il processo corrente deve andare in coda pronti (preemption) e quello appena risvegliato deve andare direttamente in esecuzione.

Come tutte le primitive, anche `sem_ini()`, `sem_wait()` e `sem_signal()` sono invocate tramite una istruzione **int** che, se eseguita da livello utente, causa un innalzamento del livello di privilegio del processore e un salto alla parte assembler della primitiva, che salva lo stato del processo corrente e chiama la parte C++, poi carica lo stato del processo puntato da esecuzione (che potrebbe essere cambiato) ed esegue una **iretq**.

La Figura 1 mostra la parte C++ della primitiva `sem_ini`. Per l'allocazione dei semafori riserviamo un array di strutture `des_sem` e ci limitiamo ad usarne una nuova, presa dall'array, ogni volta che l'utente ci chiede un nuovo semaforo. L'array ci permetterà di risalire facilmente alla struttura `des_sem` corretta, quando l'utente passerà l'identificatore del semaforo alle primitive `sem_wait()` e `sem_signal()`. Come vedremo, anche il modulo `io` utilizza dei semafori.

```

1 des_sem array_dess[MAX_SEM * 2];
2 extern "C" void c_sem_ini(int v)
3 {
4     natl i = alloca_sem();
5     if (i != 0xFFFFFFFF)
6         array_dess[i].counter = val;
7     esecuzione->contesto[I_RAX] = i;
8 }

```

Figura 1: Parte C++ della primitiva `sem_ini()`.

```

1 extern "C" void c_sem_wait(natl sem)
2 {
3     des_sem *s = &array_dessem[sem];
4     s->counter--;
5     if (s->counter < 0) {
6         inserimento_lista(s->pointer, esecuzione);
7         schedulatore();
8     }
9 }

```

Figura 2: La parte C++ della primitiva `sem_wait()`.

Per evitare che l'utente possa erroneamente usare i semafori allocati dal modulo `io`, i semafori vengono idealmente distinti in "utente" e "sistema". I semafori utente sono quelli che si trovano nelle prime `MAX_SEM` posizioni dell'array, e i rimanenti sono di livello sistema. La funzione `alloca_sem()` allocherà un indice appartenente ad una delle due parti dell'array, in base al livello di privilegio che il processo aveva quando ha invocato la primitiva (come salvato dal processore nella pila sistema). La parte C++ della primitiva `sem_wait()` è mostrata in Figura 2. Nel caso di semaforo senza gettoni (linea 5), il processo attualmente in esecuzione viene inserito nella coda del semaforo (linea 6) e ne viene scelto un altro invocando la funzione `schedulatore()`. Questa estrae dalla coda pronti il processo a più alta priorità e lo fa puntare dalla variabile `esecuzione`, in modo che la routine `carica_stato` (che verrà eseguita subito dopo) faccia saltare al nuovo processo, di fatto bloccando il precedente.

La Figura 3, infine, mostra la parte C++ della primitiva `sem_signal()`. Se ci sono processi in coda sul semaforo (linea 5), la primitiva ne estrae uno (linea 6). Si noti che la funzione `rimozione_lista`, nel caso vi fosse più di un processo, estrae quello a maggiore priorità. A questo punto la primitiva deve scegliere chi deve proseguire, tra il processo in esecuzione e quello appena estratto. La cosa più semplice è di inserire entrambi i processi in coda pronti e lasciar scegliere alla funzione `schedulatore()` (linee 7-9).

Sia la `sem_wait()` che la `sem_signal()`, prima di usare `sem`, devono

```

1 extern "C" void c_sem_signal(natl sem)
2 {
3     des_sem *s = &array_dessem[sem];
4     s->counter++;
5     if (s->counter <= 0) {
6         des_proc* lavoro = rimozione_lista(s->pointer);
7         inspronti();
8         inserimento_lista(pronti, lavoro);
9         schedulatore();
10    }
11 }

```

Figura 3: La parte C++ della primitiva `sem_signal()`.

controllare che questo sia un valido identificatore di semaforo, cioè un numero precedentemente restituito da una invocazione di `sem_ini()` per il livello (utente o sistema) corretto, e terminare forzatamente il processo in caso contrario. Questi controlli sono stati omessi dalle Figure 2 e 3 per semplicità, ma sono presenti nel codice del nucleo.

2.4 Utilizzo del debugger

Le estensioni del debugger contengono anche alcuni comandi relativi ai semafori:

`sem` mostra lo stato di tutti i semafori allocati.

`sem waiting`

mostra lo stato di tutti i semafori la cui coda non è vuota.

Il comando `sem waiting` viene eseguito automaticamente ogni volta che il debugger riacquisisce il controllo. Lo stato dei semafori è mostrato nella forma

$\{counter, lista\ processi\}$.