Sospensione dei processi

G. Lettieri

8 Aprile 2024

Una funzionalità comune dei sistemi multiprogrammati è quella per cui un processo può chiedere di essere "sospeso" per un certo intervallo di tempo. Per esempio, in Unix, la funzione `sleep(x)` permette al processo che la invoca di "andare a dormire" per `x` secondi. Una funzione del genere viene tipicamente usata in processi che devono compiere azioni periodicamente, per esempio ogni minuto: il programma eseguito dal processo conterrà un ciclo che compie l'azione desiderata e poi si sospende per 60 secondi. Nel tempo in cui un processo è sospeso il sistema può eseguire altri processi.

Un processo sospeso è a tutti gli effetti bloccato e, come tutti i processi bloccati, è in attesa di un evento che lo sblocchi e lo rimetta in coda pronti (o in esecuzione): in questo caso l'evento è il passaggio del tempo richiesto. Per realizzare questa funzionalità il sistema deve dunque tenere traccia del tempo che passa, e il modo più semplice per farlo è di programmare un timer in modo che invii una richiesta di interruzione con un periodo fisso. Questa è la soluzione che adottiamo nel nostro sistema: programmiamo il timer 0 del PC AT (quello che ha il piedino di uscita collegato al controllore delle interruzioni) in modo che invii una richiesta ogni 50 ms[1] Forniamo una primitiva **`void`** `delay(natl n)` tramite la quale un processo può chiedere di essere sospeso per `n` cicli del timer. La primitiva inserirà il processo in una opportuna coda di processi sospesi, ricordando il valore iniziale di `n`. Chiamiamo *driver* (del timer) la routine che va in esecuzione ogni volta che il timer invia una richiesta di interruzione. Il driver dovrà decrementare `n` e rimettere il processo in coda pronti (o direttamente in esecuzione) quando `n` arriva a zero.

Supponiamo di avere tanti processi sospesi, siano $P_1, \ldots, P_n$, con P_i che deve ancora attendere c_i cicli, per $1 \leq i \leq n$. Se n è grande, può essere molto costoso richiedere che il driver debba decrementare tutti gli n contatori ogni volta che va in esecuzione. Possiamo però operare nel seguente modo: manteniamo i processi ordinati in una lista in modo che $c_1 \leq c_2 \leq \cdots \leq c_n$ e in ogni elemento della lista ricordiamo soltanto quanti cicli in più il processo corrispondente deve

[1] Attenzione: programmiamo il timer una volta sola all'avvio del sistema e, da quel momento in poi, riceveremo la richiesta di interruzione ogni 50 ms, anche quando nessun processo ha chiesto di sospendersi e dunque non servirebbe tenere traccia del tempo che passa. Quella di avere una interruzione periodica comandata da un timer era una soluzione un tempo comune, ma è caduta in disuso con la diffusione dei PC portatili, in cui si cerca risparmiare la batteria evitando operazioni inutili. Nel nostro sistema continuiamo ad operare come un tempo, per semplicità.

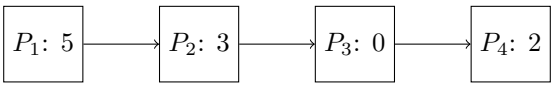

Figura 1: Esempio di lista di processi sospesi.

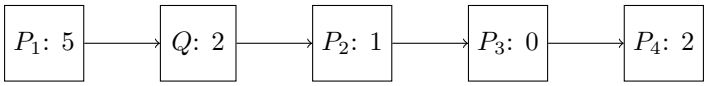

Figura 2: La lista di Figura 1 dopo l'inserimento di un nuovo processo.

attendere rispetto a quello che lo precede. In altre parole, l'elemento in cima alla lista, chiamiamolo r_1, memorizza c_1, mentre l'elemento r_i per $1 < i \leq n$ memorizza $c_i - c_{i-1}$. Per esempio, supponiamo di avere quattro processi con $c_1 = 5$, $c_2 = 8$, $c_3 = 8$ e $c_4 = 10$. La lista sarà nello stato mostrato in Figura 1. Il vantaggio di questa struttura dati è che il driver deve decrementare solo il primo elemento della lista: se il risultato è diverso da zero, sicuramente nessun processo deve essere risvegliato; altrimenti va risvegliato il processo in cima alla lista, più eventuali processi immediatamente successivi che abbiano già il contatore a zero. Per esempio, partendo dalla situazione di Figura 1, avremo che il processo 1 verrà risvegliato dopo $r_1 = 5$ esecuzioni del driver; a quel punto il processo 2 diventerà quello in cima alla lista e verrà risvegliato dopo altri $r_2 = 3$ cicli; insieme al processo 2 verrà risvegliato anche il processo 3 (avendo $r_3 = 0$); a quel punto il processo 4 diventerà quello in cima alla lista e verrà risvegliato dopo due ulteriori cicli ($r_4 = 2$). In generale, il processo che si trova in posizione i verrà svegliato dopo

$$\sum_{j=1}^{i} r_j = c_1 + (c_2 - c_1) + \cdots + (c_{i-1} - c_{i-2}) + (c_i - c_{i-1}) = c_i$$

cicli, come desiderato.

Supponiamo che, partendo dalla situazione in Figura 1, un nuovo processo Q chieda di essere sospeso per 7 cicli. Il processo dovrà essere inserito in lista in modo che alla fine si ottenga la situazione in Figura 2. Si noti che è necessario decrementare il contatore di P_2, che si trova in lista subito dopo Q, in modo che non cambi il suo tempo di attesa, ma non è necessario modificare i contatori dei processi successivi (P_3 e P_4) e precedenti (P_1).

1 Implementazione nel nucleo

La lista dei processi sospesi è definita in Figura 3. Ogni elemento della lista contiene, oltre al puntatore `p_rich` che serve a realizzare la lista, un puntatore `pp` al (descrittore del) processo sospeso e un contatore `d_attesa`, usato come spiegato sopra. La variabile `sospesi` punta alla testa della lista.

```
1  struct richiesta {
2          natl d_attesa;
3          richiesta *p_rich;
4          des_proc *pp;
5  };
6  richiesta *sospesi;
```

Figura 3: La lista dei processi sospesi.

```
1  void inserimento_lista_attesa(richiesta *p)
2  {
3          richiesta *r, *precedente;
4          r = sospesi;
5          precedente = nullptr;
6          while (r && p->d_attesa > r->d_attesa) {
7                  p->d_attesa -= r->d_attesa;
8                  precedente = r;
9                  r = r->p_rich;
10         }
11         p->p_rich = r;
12         if (precedente)
13                 precedente->p_rich = p;
14         else
15                 sospesi = p;
16         if (r)
17                 r->d_attesa -= p->d_attesa;
18 }
```

Figura 4: Inserimento di una nuova richiesta nella lista del timer.

```
1  extern "C" void c_delay(natl n)
2  {
3          if (!n)
4                  return;
5          richiesta* p = new richiesta;
6          p->d_attesa = n;
7          p->pp = esecuzione;
8          inserimento_lista_attesa(p);
9          schedulatore();
10 }
```

Figura 5: La parte C++ della primitiva `delay()`.

La Figura 4 mostra l'implementazione della funzione che inserisce un nuovo elemento nella lista dei processi sospesi. La funzione usa la tecnica dei due puntatori, `r` e `precedente`, con `precedente` che resta sempre un passo indietro rispetto a `r`. Il ciclo alle linee 6–10 avanza i due puntatori fino a quando non puntano ai due elementi tra i quali va inserito il nuovo. Man mano che si avanza nella lista, al campo `d_attesa` del nuovo elemento vengono sottratti i campi `d_attesa` degli elementi attraversati (linea 5), in modo che alla fine contenga solo il numero di cicli in più da attendere rispetto agli elementi che lo precedono. All'uscita del ciclo, le linee 11–15 aggiustano i puntatori in modo da inserire l'elemento in lista nella posizione individuata. La linea 17 serve ad aggiustare il contatore dell'eventuale elemento successivo, in modo da non modificare il suo tempo di attesa nonostante l'inserimento del nuovo elemento.

Alla lista accedono due routine di sistema: la primita `delay()` e il driver del timer. Il driver, come la primitiva, è eseguito atomicamente, e questo garantisce la mutua esclusione nell'accesso alla lista. La primitiva `delay()`, con argomento `natl n`, è una normale primitiva di sistema, con un gate nella IDT e una parte scritta in Assembly analoga a quella delle altre primitive. In Figura 5 mostriamo solo la parte C++. Il caso `n` pari a zero non ha senso, ma dobbiamo gestirlo lo stesso perché non possiamo fare alcuna ipotesi sugli argomenti che l'utente decide di passare alle primitive. Decidiamo di trattarlo come una "nop" (linee 3–4). Nei casi normali allochiamo una nuova struttura `richiesta` e la inizializziamo con i valori opportuni: il numero di cicli da attendere e il processo che ha fatto la richiesta (linee 5–7). Si noti che usiamo **`new`**: l'operatore **`new`** è definito nel file `sistema.cpp` e usa una funzione di `libce` che gestisce una zona della memoria di sistema usata come heap. Alla linea 8 usiamo la funzione di Figura 4 per inserire il nuovo elemento nella lista dei processi sospesi. La sospensione vera e propria avviene perché chiamiamo la funzione `schedulatore()` che cambia il valore di `esecuzione`: al ritorno dalla funzione `driver_td()` verrà eseguita la coppia **`call`** `carica_stato`**`; iretq`** che cederà il controllo al nuovo processo, mentre quello che era precedentemente in esecuzione resterà bloccato in attesa che il driver del timer lo risvegli.

```
1         .extern c_driver_td
2 driver_td:
3         call salva_stato
4         call c_driver_td
5         call apic_send_EOI
6         call carica_stato
7         iretq
```

Figura 6: La parte assembler del driver del timer.

```
1  extern "C" void c_driver_td(void)
2  {
3          inspronti();
4          if (sospesi)
5                  sospesi->d_attesa--;
6          while (sospesi && sospesi->d_attesa == 0) {
7                  inserimento_lista(pronti, sospesi->pp);
8                  richiesta *p = sospesi;
9                  sospesi = sospesi->p_rich;
10                 delete p;
11         }
12         schedulatore();
13 }
```

Figura 7: La parte C++ del driver del timer.

Il driver stesso segue sostanzialmente lo stesso schema di una primitiva, ma va in esecuzione per effetto di una richiesta di interruzione e non di una istruzione **int**. La parte in Assembly è mostrata in Figura 6. L'indirizzo `driver_td` sarà puntato da un gate della IDT. In fase di inizializzazione l'indice di questo gate deve essere anche scritto nel registro dell'APIC associato al piedino di interruzione collegato al timer. Rispetto ai gate di una primitiva, il gate del driver avrà DPL pari a sistema, perché vogliamo che il driver vada in esecuzone *solo* per effetto di una richiesta di interruzione da parte del timer e non vogliamo che l'utente lo faccia partire tramite una **int**.

Il codice in Assembly in Figura 6 è del tutto analogo a quello di una primitiva, con una sola differenza: prima di terminare inviamo l'EOI all'APIC (linea 5), che altrimenti non farebbe più passare richieste di interruzione con priorità minore o uguale di quella del timer. Nel nostro caso non passerebbe più nessun tipo di richiesta, in quanto abbiamo assegnato al timer la massima priorità (si veda la definizione di `TIPO_DRIVER` in `include/costanti.h`).

La parte C++ del driver è mostrata in Figura 7. Il driver potrebbe risvegliare processi che hanno precedenza maggiore di quello che era in esecuzione all'arrivo dell'interruzione, quindi dobbiamo prevedere una possibile preemp-

tion. Per gestire in modo uniforme tutti i casi, inseriamo preventivamente il processo in `esecuzione` in testa alla coda `pronti` (linea 3) e chiamiamo `schedulatore()` alla fine (linea 12): se il codice nel mezzo non inserisce altri processi in coda `pronti` le due operazioni si annullano a vicenda e il contenuto di `esecuzione` non cambia, altrimenti alla fine punterà al processo con priorità maggiore.

Ogni volta che va in esecuzione, il driver decrementa il contatore `d_attesa` dell'eventuale processo in testa alla lista `sospesi` (linea 5), quindi estrae dalla testa tutti gli eventuali elementi con `d_attesa` pari a zero (ciclo alle linee 6–11) inserendo i corrispondenti processi in coda `pronti` (linea 7) e facendo attenzione a deallocare (linea 10) la struttura `richiesta` che era stata precedentemente allocata con **`new`** durante la `c_delay()` (linea 5 di Figura 5).

Come sempre, il processo interrotto o uno di quelli risvegliati (in caso di preemption) andrà in esecuzione al termine di `c_driver_td()`, che ritornerà a `driver_td`, che alla fine eseguirà la coppia **`call`** `carica_stato`**`; iretq`**.

2 Chi esegue il driver

Come ce lo siamo chiesti per le primitive, chiediamoci ora chi esegue il driver. Probabilmente, rispetto al caso di una primitiva, è più semplice convincersi che c'è *nessun processo* che esegue il driver. Questo perché l'unico candidato (il processo che era in esecuzione all'arrivo dell'interruzione) non ha volontariamente chiamato il driver e, molto probabilmente, non è neanche interessato a ciò che il driver deve fare. Il driver usa la pila sistema del processo interrotto (e anche il resto della memoria privata) e questa volta, rispetto al caso della primitiva, è naturale pensare che l'abbia presa in prestito, proprio perché il processo interrotto non l'ha invocato volontariamente.

Eppure il contesto di esecuzione del driver è identico a quelle delle primitive: si tratta di un contesto atomico in cui si usano temporaneamente le risorse di un processo che era in esecuzione prima dell'attraversamento di un gate, e in tutte e due i casi eseguiamo una `salva_stato` all'ingresso e una `carica_stato` (seguita da **`iretq`**) alla fine. Conviene dunque pensare all'esecuzione delle primitive nello stesso modo in cui si pensa all'esecuzione del driver: in entrambi i casi l'esecuzione non va associata ad alcun processo.