di Andrea de Prisco

Parallel Processing: tutti in coda!

seconda parte

In questa seconda puntata dedicata al multitasking, parleremo brevemente dei processi, dello stato dei processi e delle code di quest'ultimi, utilizzate per tenere «ordine» nella memoria dei calcolatori di questo tipo, dando in questo modo la possibilità a tutti i lavori di avere la giusta fetta di «esecuzione»

Definiamo un processo

Se sembra abbastanza semplice definire cosa sia un programma o l'algoritmo da esso descritto, un tantino più ostico potrebbe essere definire cosa sia un processo. Potremmo ad esempio dire che un processo è un programma «che gira», qualcosa in un certo senso di vivo e vegeto all'interno di un calcolatore, capace cioè di «provocare eventi». Questa definizione, pur non essendo ancora perfetta, è certamente migliore della prima: infatti un processo può anche non «girare» se, ad esempio, sta attendendo (passivamente, vedi dopo) il verificarsi di un determinato evento.

La distinzione tra programma e processo è dunque sì filosofica, ma abbastanza importante da non essere mai sottovalutata. Così, partendo invece dal verso opposto, potremmo dire che una determinata funzione, in senso strettamente matematico, è certamente unica: l'algoritmo che la implementa già non lo è più in quanto algoritmi completamente diversi possono descrivere (diversamente) la medesima funzione.

Se passiamo poi alla descrizione dell'algoritmo attraverso un linguaggio di programmazione (ottenendo quindi un programma) l'unicità iniziale va a farsi friggere ancora di più in quanto vi sono infiniti modi diversi di scrivere un programma che implementa un determinato algoritmo. Ecco però che magicamente, quando mandiamo in esecuzione (lanciamo, facciamo girare, ecc.) il programma appena scritto, ciò che abbiamo ottenuto è stato di dare vita al processo che implementa la funzione dalla quale siamo partiti. Unico come la funzione di partenza, almeno dal punto di vista strettamente informatico. Così un processo di «sort» (ordinamento) sarà rappresentato sui nostri schemi come un bel «pallottolo» sul quale arrivano dati e fuoriescono risultati senza minimamente riferirsi all'algoritmo o al linguaggio di programmazione utilizzato (figura 1). Analogamente, un'applicazione «parallela» (realizzata con più processi cooperanti) potremo rappresentarla graficamente come un grafo in cui tutti i nodi sono i processi e gli archi tra i nodi le comunicazioni interprocess (figura 2).

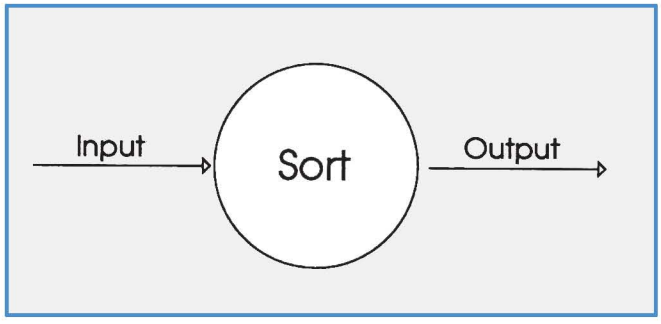


Figura 1 - Nei grafici rappresentati in questo e nei successivi articoli, ogni processo è rappresentato da una circonferenza con dentro il nome.

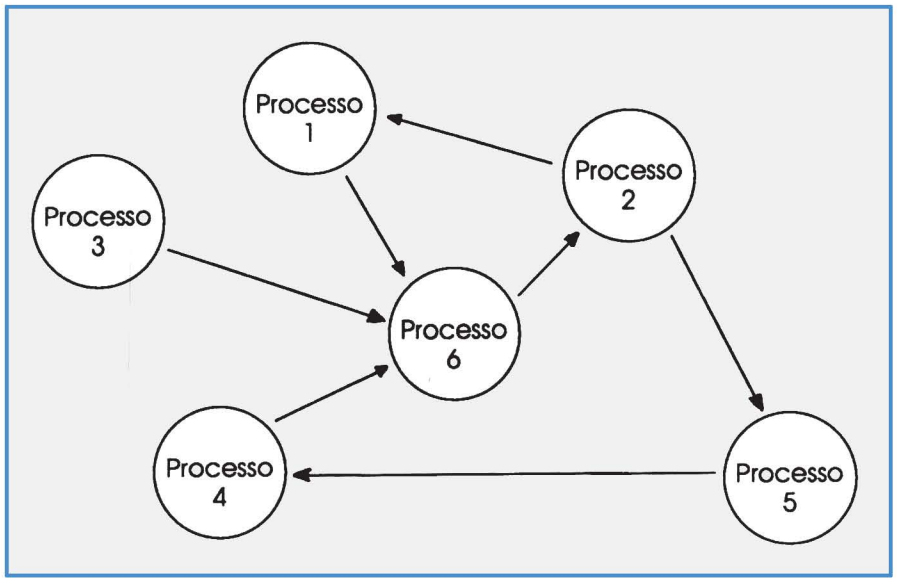


Figura 2 - Un'applicazione multitask è formata da più processi comunicanti. Può essere rappresentata da un grafo in cui i nodi sono i processi e gli archi le comunicazioni interprocess.

Stato di un processo

Tutti i processi lanciati in un calcolatore multitask, in un determinato istante possono stare solo in uno dei tre seguenti stati: esecuzione, pronto, attesa. Sempre nell'istante in cui ci stiamo riferendo saranno in esecuzione al massimo tanti processi quanti sono i processori attivi sul quella determinata macchina.

In stato di pronto, invece, sono i processi che possono essere eseguiti ma che in quel momento non sono in esecuzione. In quest'ultimo stato vi transiteranno appena la CPU (o una CPU per i multiprocessor) interrompe l'elaborazione del processo in corso per una delle cause che tra poco vedremo e sceglie dalla lista dei processi pronti (secondo uno schema prioritario o libero) il successivo programma da elaborare.

Nella lista dei processi in stato di attesa ci finiscono tutti i processi che, sempre in quell'istante, non possono essere elaborati perché manca qualche dato richiesto a una periferica di I/O o a un altro processo. In figura 3 sono rappresentati graficamente gli stati dei processi che abbiamo appena trattato: notare che si tratta di un diagramma di stati e non di processi, e le frecce che uniscono i nodi non sono comunicazioni (come nel grafo di figura 2), ma semplicemente passaggi di stato. Così un processo in stato di pronto può passare in stato di esecuzione e viceversa. Un processo in stato di attesa può transitare solo in stato di pronto e un processo in stato di esecuzione può anche essere posto in stato di attesa. Detto questo, diamo anche una spiegazione un po' più dettagliata a questo transito di stati.

Innanzitutto un processo in stato di esecuzione può cambiare il suo stato (quindi essere rilasciato dalla CPU) in due soli casi (più, naturalmente, in caso di terminazione o aborto dello stesso): se effettua una operazione di I/O verso dispositivi o inizia operazioni di comunicazione sincrona con altri processi oppure se scade il suo «quanto di tempo» di esecuzione. Infatti per implementare

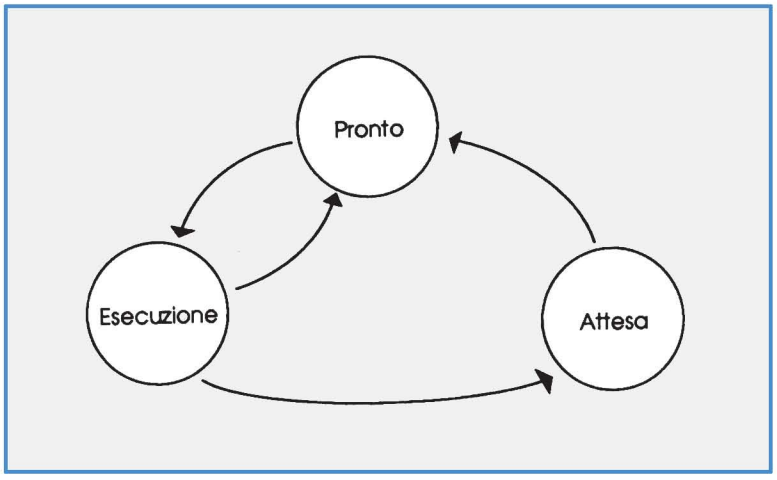
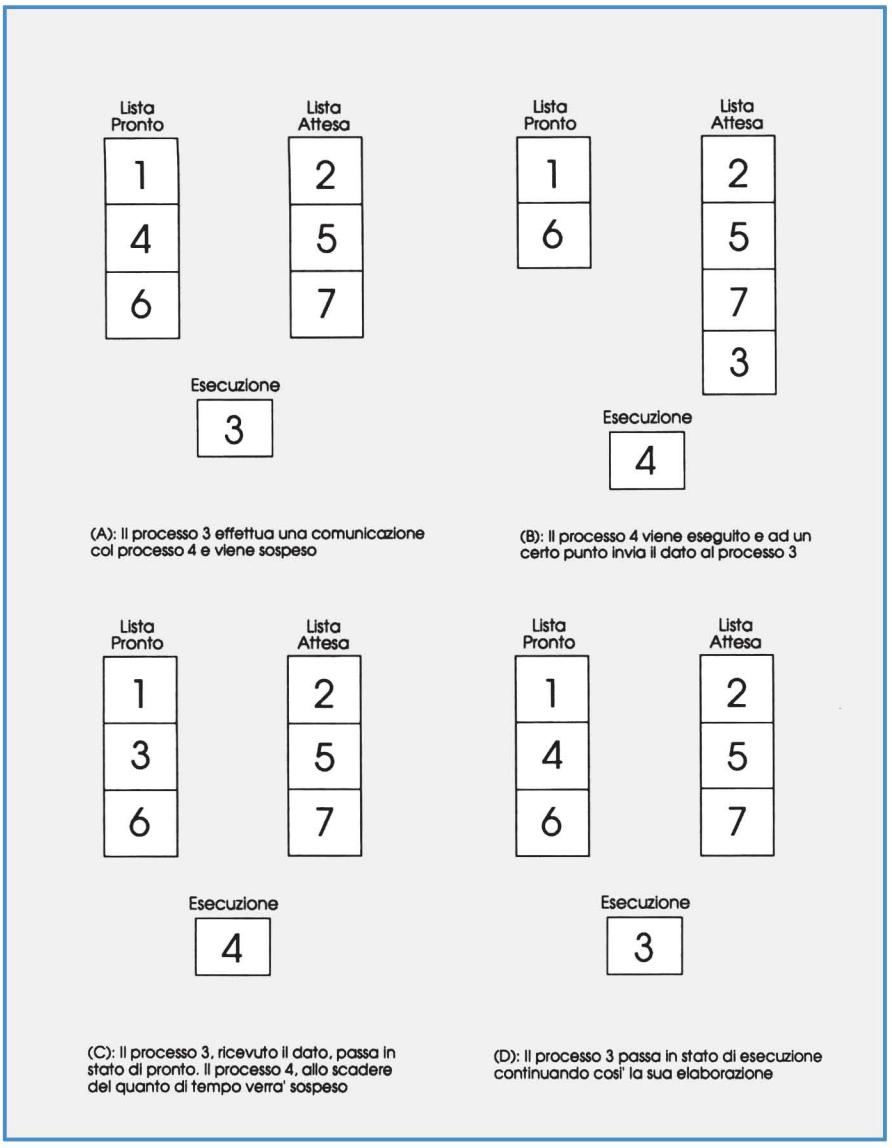


Figura 3 - Grafo degli stati di un processo. In questo grafo i nodi rappresentano i possibili stati di un processo e gli archi le possibili transizioni di stato.



▲ *Figura 4 - Sequenza di cambiamento di stato per due processi che eseguono una comunicazione interprocess.*

Figura 6 - Nel descrittore del processo (PCB, Process Control Block) vengono memorizzate alcune informazioni riguardanti il processo descritto.

il multitasking su un singolo processore, i processi in esecuzione vengono elaborati in «divisione di tempo».

La CPU ad esempio esegue per qualche centesimo di secondo il processo 1, poi passa al processo 2, poi al processo 3 e così via salvando di volta in volta, a ogni cambio di contesto, lo stato dei processi eseguiti e rilasciati. E naturalmente, un processo interrotto perché è scaduta la sua frazione di tempo di CPU, rimane in stato di pronto (può comunque ripartire non appena la CPU lo riacciuffa). Se invece la sospensione avviene perché l'elaborazione non può continuare a causa di una richiesta di I/O inoltrata, ma non ancora soddisfatta, viene accodato nella lista dei processi in attesa.

Da questa lista possono transitare in stato di pronto non appena il dato richiesto è recapitato al processo sospeso. In figura 4 è mostrata una sequenza di transiti di stato per due processi che eseguono tra loro una operazione di comunicazione interprocess sincrona.

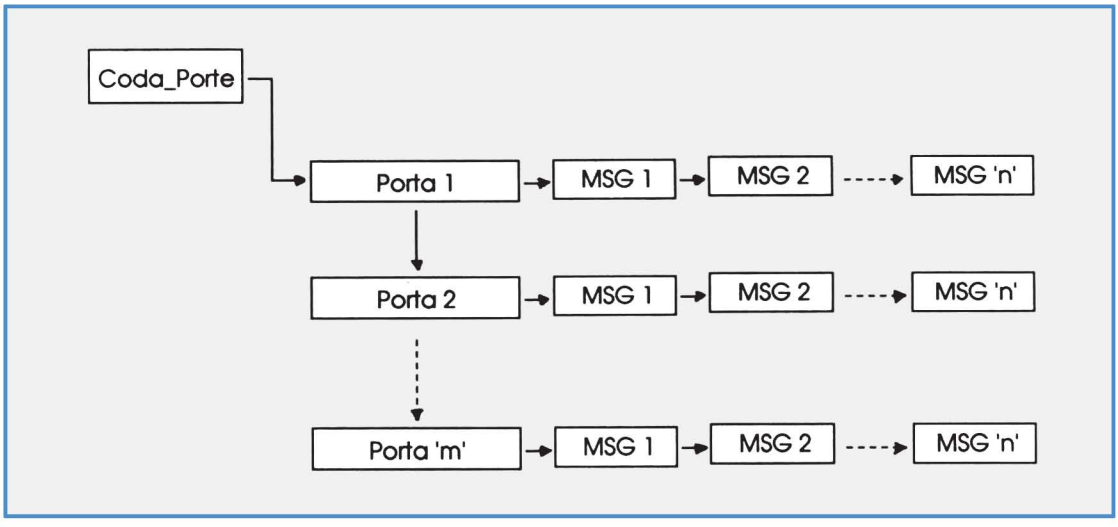


Figura 5 - Lista delle porte dichiarate dai vari processi e relative code dei messaggi in arrivo non ancora letti. ▶

Attesa attiva e attesa passiva

In un calcolatore multitasking, uniprocessore o multiprocessore che sia, è importante sfruttare sempre al massimo (nonché intelligentemente) la/le CPU. L'intelligenza va adoperata principalmente per far lavorare nel migliore dei modi le CPU in maniera tale da ottenere il massimo sfruttamento possibile delle risorse di calcolo disponibili. Così assolutamente da dimenticare saranno i loop di attesa sugli eventi che andranno sostituiti da opportune chiamate al sistema che pongono in stato di attesa il processo chiamante e lo risveglieranno (ponendolo quindi in stato di pronto) appena l'evento atteso si sarà verificato. Facciamo un esempio: il processo «A» deve attendere sulla sua porta «PortaDiA» un dato in arrivo dal processo «B». Il modo sbagliato per implementare tale attesa è, come detto, quello di scrivere nel processo «A» un loop di questo tipo:

```
A::
   .
   .
   .
   while (msg_non_arrivato("PortaDiA"))
     nop
   end
   .
   ricevi("PortaDiA",messaggio)
   .
   .
```

In questo modo, infatti, il processo «A» assorbirà tempo di CPU anche durante l'attesa (attiva, in questo caso) mentre potrebbe comodamente andare «a dormire» a condizioni di essere svegliato non appena il messaggio arriva. Una soluzione è quella di utilizzare una funzione di wait sull'evento «PortaDiA». La stessa porzione di programma appena vista sarà scritta in questo modo:

```
A::
   .
   .
   .
   wait ("PortaDiA")
   .
   ricevi("PortaDiA",messaggio)
   .
   .
```

Ed in pratica «A» quando esegue la «wait» se il messaggio atteso è già presente sulla porta continua e lo preleva con la funzione successiva, altrimenti viene messo in stato di attesa su quell'evento. Parallelamente il processo «B» ad un certo punto della sua elaborazione invierà il messaggio ad «A»:

```
B::
   .
   .
   invia("PortaDiA",messaggio)
   .
   .
```

La domanda da farsi è certamente questa: cos'è che effettivamente sveglia il processo «A» non appena «B» esegue la funzione «invia»?

Ovviamente il sistema operativo: infatti tanto la funzione «wait» quanto la coppia «invia» e «ricevi» sono funzioni di quest'ultimo quindi eseguite in ambiente supervisore. «B» non lo sa, ma con la sua «invia» ha il potere di svegliare «A» se questo sta dormendo in attesa del messaggio da ricevere.

Per capire meglio vediamo brevemente una possibile implementazione di queste tre funzioni di sistema operativo.

Naturalmente esisteranno delle opportune strutture dati di sistema non visibili al livello dei processi ma da questi indirettamente modificabili.

La prima funzione «wait» potrebbe essere scritta in questo modo:

```
WAIT(arg1)::
   if coda(arg1) <> VUOTA then return;
   else sospendi(PROCESSO, arg1);
return;
```

La funzione di sistema «sospendi» non fa altro che mettere il processo chiamante in stato di attesa marchiando opportunamente il PCB (Process Control Block) riguardo al motivo della sospensione, in questo caso per attesa sulla porta «arg1».

```
SOSPENDI(processo, evento)::
   PCB(processo).stato  := ATTESA;
   PCB(processo).evento := evento;
   appendi(coda_attesa, PCB(processo));
   elimina(coda_pronto, PCB(processo));
return;
```

«Appendi» ed «elimina» sono altre due funzioni di sistema: la prima permette di inserire un elemento in testa ad una coda, la seconda elimina dalla coda indicata come primo parametro l'elemento passato come secondo.

La funzione «invia» sarà scritta approssimativamente nel seguente modo:

```
INVIA(nome_porta, messaggio)::
   porta := cerca(coda_porte, nome_porta);
   appendi(porta.coda_messaggi, messaggio);
   sveglia(porta.titolare, nome_porta);
return;
```

in questa funzione sono chiamate a loro volta altre tre funzioni di sistema: la prima, «cerca», serve per ritrovare nella lista delle porte definite dai vari processi (e mantenuta dal sistema operativo) la porta passata come primo parametro alla funzione «invia». La seconda fa un inserimento nella coda (l'abbiamo già vista nella funzione «sospendi»), la terza serve per risvegliare il processo ricevente (ovvero quello che ha dichiarato la porta) se questo effettivamente è stato sospeso su quell'evento. Ecco una possibile implementazione:

```
SVEGLIA(processo, evento)::
   if PCB(processo).stato = PRONTO then return;
   else if PCB(processo).evento = evento
        then
           PCB(processo).stato := ATTESA;
           appendi(coda_pronto, PCB(processo));
           elimina(coda_attesa, PCB(processo));
        end
return;
```

Per finire, la funzione «ricevi» (che in questo esempio abbiamo pensato «non bloccante») potrebbe essere facilmente implementata in questo modo:

```
RICEVI(nome_porta, messaggio)::
   porta := cerca(coda_porte, nome_porta);
   if porta.coda = VUOTA then messaggio = NULL;
   else messaggio := preleva(porta.coda);
return;
```

Qui la funzione «cerca» è la stessa che abbiamo visto nella funzione «invia», la funzione «preleva» permette di prelevare il primo messaggio presente nella coda messaggi di quella porta eliminandolo, conseguentemente, da questa.

Riassumendo, il processo destinatario che esegue una wait viene sospeso (dalla stessa wait) se la coda dei messaggi di quella porta è vuota. Resta in tale stato (attesa) fino a quando il processo mittente non effettua un «invia» sulla stessa porta.

Conclusioni

In questa puntata di Multitasking abbiamo dato un po' di definizioni che ci saranno molto d'aiuto nelle puntate successive. Ciò che maggiormente dovremo tenere d'occhio è lo schema di transito di stato dei processi e delle strutture dati del sistema operativo (le code dei processi, le porte, le code dei messaggi sulle porte, ecc.) che permettono l'implementazione del multitask sui moderni sistemi di calcolo. Nelle prossime puntate vi mostreremo alcune tecniche di programmazione multitasking e i vari meccanismi di comunicazione e sincronizzazione interprocess solo anticipati in questa e nella precedente puntata della rubrica. Arrivederci... MC