# Sistemi 7

Avevamo visto come le shell implementano la redirezione di input e di output. Com’è implementato invece il meccanismo delle pipe? (A | B | C | D …)

Il tubo si crea attraverso la syscall pipe (int pipefd[2]): con questa si crea un canale unidirezionale e anonimo per passare dei byte (è come un socket ma monodirezionale). Il canale è un byte-stream (non c’è il concetto di messaggio / datagramma) e viene tipicamente usato per far comunicare i processi.

In pipefd sono contenuti i file descriptor della pipe, pipefd[0] è il lato lettura e pipefd[1] è il lato scrittura: quello che viene scritto nella pipe finisce in un buffer del kernel; quando questo è pieno le write vengono messe in attesa (o falliscono se il flag O\_NONBLOCK è abilitato). Se tutti i FD di lettura sono stati chiusi, write provoca SIGPIPE (che di default uccide il processo). Quando il buffer è vuoto, le read vengono messe in attesa (o falliscono se il flag O\_NONBLOCK è abilitato). Se tutti i FD di scrittura sono stati chiusi, read restituisce 0, come EOF. Vedere pipe(2) e pipe(7).

Quando questi due file descriptor non servono più è molto importante chiuderli.

Immaginiamo di voler implementare A | B. Bisogna innanzitutto aprire la PIPE (con pipe()), poi bisogna effettuare due fork (una per A e una per B), poi bisogna effettuare il dup dei file descriptors in modo da rimpiazzare stdout di A (1) con pipefd[1] (il lato scrittura) e stdin di B (0) con pipefd[0] (il lato lettura). A questo punto si possono chiudere i file descritptor aggiuntivi (pipefd[0] e pipefd[1]) su tutti i processi (la bash e i due originati dalle fork) e poi fare le exec di A e B.   
A e B vanno fatti partire contemporaneamente, sia per una questione di efficienza, sia perché se A riempie il buffer della Pipe, l’unico modo per far andare avanti il programma è che ci sia B a leggere da lì (altrimenti il kernel lo tiene bloccato, visto che vede i file descriptors aperti). Questo “svantaggio” è in realtà una feature molto utile, perché permette alle Pipe di non occupare effettivo spazio in memoria (se A produce troppo output, anziché riempire il disco si aspetta che B legga dalla Pipe e poi si sovrascrive il buffer).

Cambiamo argomento e passiamo allo scheduling.  
Abbiamo già accennato al fatto che la CPU viene virtualizzata utilizzando il time sharing; ovvero, ogni processo riceve l’uso “esclusivo” della CPU per un po’, poi si passa a un altro.

Il meccanismo che permette di cambiare processo è chiamato cambio di contesto (context switch) e il modo in cui viene implementato dipende dall’hardware sottostante (sebbene l’idea generale sia di salvare i registri da qualche parte e caricare quelli del nuovo processo, poi salvare anche quelli e continuare).   
Per realizzare questo meccanismo è anche necessario lo scheduler: è quella parte del kernel che si occupa di schedulare (cioè mandare in esecuzione) o de-schedulare (cioè sbattere fuori) i processi. Lo Scheduler è ciò che decide chi è il prossimo processo.

Non c’è una politica di scheduling migliore delle altre, quella che si usa dipende in gran parte dal sistema che si usa e dall’hardware.

Un problema è l’implementazione del context-switch in maniera efficiente (se si cambia contesto troppe volte al secondo l’overhead, cioè il tempo perso per fare lo switch, diventa complessivamente troppo). Un altro problema è relativo al controllo: bisogna assicurarsi che un processo rilasci effettivamente la CPU quando finisce il suo turno.

Per risolvere questo secondo punto, c’è l’approccio cooperativo e quello non cooperativo.

Nell’approccio cooperativo il SO si fida dei processi, questi ogni tanto effettuano una syscall, anche se non ne hanno bisogno, così quando il kernel prende il controllo può scegliere di effettuare il cambio di contesto. Però cosa succede se i processi non lo fanno, per malizia o per un bug? C’è il problema che un processo può tenere il controllo della CPU: in genere il processo cooperativo ha senso solo su quei dispositivi ad uso altamente specifico (come le automobili, o i frigoriferi), in cui il venditore sa esattamente quali programmi ci vanno in esecuzione.  
L’approccio non cooperativo prevede che centinaia di volte al secondo venga inviato un interrupt che fa tornare la CPU in modalità supervisore, permettendo al kernel di fare un context switch se necessario.

L’idea del context switch è semplice:

* Si salvano i registri del processo che si sta eseguendo (nel process control block datogli dal kernel)
* Si caricano i registri del processo che vogliamo mandare in esecuzione (includendo pure, per esempio, il registro che punta alla tabella delle pagine)

L’implementazione è meno semplice: molti dettagli dipendono dall’hardware.

In generale succede che ogni processo ha due stack diversi, uno per la modalità utente e uno per la modalità kernel, questo serve perché a seguito di push e pop lo stack utente potrebbe non puntare a un’area sensata, quindi, si usa lo stack kernel per salvare i registri durante il context switch (anche se questo richiede un passaggio in più durante il ripristino).

Durante la sua vita, un processo può essere in diversi stati: quando il processo è in esecuzione sulla CPU il suo stato è “Running”; c’è un altro stato, che alcuni sistemi chiamano “Runnable”/”Ready”, che vuol dire che il processo potrebbe essere eseguito ma la CPU non lo sta eseguendo (solo un processo per ogni core della CPU sarà Running). Per passare da Ready a Running, un processo deve essere selezionato dallo scheduler come il successivo per il context switch. C’è un altro stato per il processo, quello di Init, che corrisponde all’inizializzazione del processo subito prima di diventare Ready.   
Lo stato “Blocked”/”Waiting”/”Sleeping” indica invece che un processo ha chiesto di interagire con una periferica che richiede tempo (come un dispositivo di IO) e quindi adesso, finché non riceve i dati che ha richiesto, non ha senso tenerlo Running sulla CPU. I processi possono diventare Waiting da Running, quando hanno finito di aspettare questi diventano di nuovo Ready, starà quindi allo scheduler riportarli sulla CPU e renderli Running. Un processo Running diventa Zombie quando termina, ma lo può diventare anche da Waiting o da Ready se arriva un segnale di kill al processo (anche se per i processi in waiting è complicato perché può capitare che il sistema voglia aspettare che l’interazione con le periferiche di IO sia terminata).  
Lo stato “Stopped” è quello che si ottiene quando ad esempio si preme Ctrl+Z (quando è temporaneamente in pausa, ma lo si può far ripartire).

Come scelgo il prossimo processo da mettere in esecuzione? Ci sono più algoritmi di scheduling, che sono valutati secondo delle metriche.   
Nota: spesso i processi che girano in un sistema sono detti workload, e i singoli processi sono chiamati job.

Partiamo con assunzioni irrealistiche:  
- Ciascun job dura lo stesso tempo  
- Tutti i job arrivano allo stesso momento  
- Una volta iniziato uno lo si porta fino in fondo  
- I job usano solo CPU, no I/O  
- Il tempo di ciascun job è noto a priori.

Per misurare la “bontà” di un algoritmo di scheduling si usano come metriche la performance e la fairness (cioè l’efficienza e il fatto che siano a turno eseguiti tutti i processi, senza situazioni di “starvation”).

Definiamo il tempo di Turnaround come il tempo tra quando un job arriva e quando termina (Nell’ipotesi in cui tutti i job arrivino all’istante 0 il tempo di Turnaround equivale al tempo di completamento). Se ciascun job richiede 10 secondi e abbiamo 3 jobs il Turnaround medio sarà = (10 + 20 +30)s / 3 = 20s.

Togliamo l’ipotesi che ciascun job duri lo stesso tempo, se un job dura molto più degli altri può fare “effetto convoglio”: se A dura 100, B e C 10 abbiamo che il Turnaround medio sarà = (100 +110 +120) / 3 = 110s.

Un algoritmo di scheduling migliore, a questo punto, può, assumendo che i job arrivino allo stesso momento, eseguire prima i job più veloci: nella situazione analoga a prima il Turnaround diventa = (10 +20 + 120) / 3 = 50

Con l’assunzione che tutti i job arrivino nello stesso momento si può provare che SJF (Shortest Jobs First) è ottimo. Ma funziona bene anche se togliamo questa assunzione?  
Se A arriva all’istante 0, mentre B e C all’istante 10, per via dell’assunzione irrealistica n.3 si ha che: si inizia A all’istante 0, ma B e C vengono iniziati a 110 e 120 e si ritorna all’effetto convoglio. In questo caso si può usare un algoritmo che dà priorità al processo che terminerà per primo (Shortest to Completion First) e che ignora l’assunzione ridicola di non interrompere i jobs:

In questo caso A partirebbe a 0, verrebbe interrotto a 10 per eseguire B e poi C, e riprenderebbe solo dopo: il tempo di Turnaround diventerebbe (120 + (20-10) + (30-10)) / 3 = 50  
  
Nei sistemi interattivi c’è un’altra metrica molto importante, Il tempo di risposta, che non è il tempo che ci mette a terminare, bensì al tempo che ci mette a far vedere che sta facendo qualcosa (sennò noi umani ci mettiamo a spam doppio cliccare su un’icona del desktop).

Per minimizzare il tempo di risposta, quello che possiamo fare è il cosiddetto round-robin: ogni job ottiene una “fetta/un quanto di tempo” e poi si passa al prossimo job. Questo peggiora le altre metriche, supponiamo che A, B e C arrivino al tempo 0 e durino 5 secondi; il tempo medio con Round-Robin (con fettine da 1 secondo) sarebbe:

T-risposta = (0+1+2)/3 = 1.

Con L’algoritmo STCF, ottimo per il tempo di Turnaround, il tempo di risposta medio però sarebbe:

T-risposta = (0+5+10) / 3 = 5

Più si restringe la fetta di tempo, minore diventa il tempo di risposta, tuttavia, aumenta il tempo perso per via dell’overhead. Inoltre, rispetto al Turnaround il Round-Robin si comporta malissimo (perché l’esecuzione viene diluita nel tempo, quindi anche Job corti ci mettono parecchio a terminare).

Ci rimangono due assunzioni irrealistiche da discutere: che i job usano solo la CPU e che il loro tempo di completamento sia noto a priori. Abbiamo già detto che non ha senso lasciare sulla CPU un processo che aspetta dati da disco, quindi, basta mettere i processi in modalità Waiting quando prova ad accedere a una periferica di I/O. C’è però da tener conto che il tempo di esecuzione di un job non è noto a priori.   
  
Nel mondo reale, ci sono algoritmi che provano di fare un compromesso e di ottenere buoni tempi di risposta senza avere eccessivi tempi di Turnaround medi, ma li vedremo un’altra volta.