2.2 Scheduling in Linux

In generale, i processi possono essere classificati secondo due schemi:

- CPU bound vs. I/O bound;
- interattivi vs. batch vs. real-time.

Le due classi sono indipendenti, in quanto possono esistere, ad esempio, processi batch sia I/O bound che CPU bound e lo stesso vale per i processi interattivi o real-time.

Rispetto a questa distinzione, la *politica* di scheduling Linux è in grado di:

- riconoscere processi soft real-time a cui è assegnata un'elevata priorità (ad esempio, processi che realizzano il play di un file multimediale);
- favorire implicitamente processi I/O bound (tuttavia, non è in grado di riconoscere processi batch o interattivi).

Più in dettaglio, la politica di scheduling Linux si propone il raggiungimento dei seguenti obiettivi, alcuni dei quali contrastanti tra loro:

- timesharing;
- gestione di priorità dinamiche;
- avvantaggiare processi I/O-bound;
- tempi di risposta bassi;
- throughput elevato per i processi in background;
- evitare starvation.

Distinguendo tra politica ed *algoritmo* di scheduling, gli obiettivi della politica sono perseguiti dall'algoritmo di scheduling nel modo seguente:

- suddivisione del tempo in epoche;
- dipendenza della priorità dal residuo del quanto di tempo dall'epoca precedente;
- definizione delle condizioni che determinano l'invocazione dello scheduler.

2.2.1 Politica di scheduling

In Linux lo scheduling si basa sul concetto di timesharing, per cui ad ogni processo è assegnato un quanto di tempo massimo per l'esecuzione. La selezione del prossimo processo da eseguire è basata sul concetto di priorità; questa può essere dinamica o statica. In particolare, la prima è introdotta per evitare il fenomeno della starvation, mentre la seconda è stata introdotta per consentire la gestione di processi real-time. Ai processi real-time (più precisamente soft real-time, utili ad esempio per riprodurre flussi audio e/o video in applicazioni multimediali) è assegnata una priorità maggiore di quella assegnata ai processi ordinari.

Di norma Linux prevede uno scheduling con prelazione (preemptive), per cui ad un processo viene tolta la CPU se:

- esaurisce il quanto di tempo a sua disposizione;
- un processo a priorità più alta è pronto per l'esecuzione (`TASK_RUNNING`).

La durata del quanto di tempo è tipicamente di 20 clock ticks, o 210 millisecondi. Questo è circa il più grande valore ammissibile senza che sia compromessa la reattività del sistema. Gli utenti possono variare la durata del quanto assegnato ad un processo tramite la funzione `nice(int n);` tuttavia, gli utenti comuni possono solo diminuirlo, mentre l'utente amministratore può anche aumentarlo. In particolare, l'incremento n del quanto può variare nell'intervallo $-20 \leq n \leq 19$, dove a valori negativi di n corrisponde un incremento del quanto (fino ad un massimo di 40), mentre valori positivi di n decrementano la priorità di un processo (fino ad un minimo di 1).

2.2.2 Algoritmo di scheduling

Il tempo è suddiviso in periodi, detti epoche, che si ripetono ciclicamente. All'inizio di ogni epoca, a ciascun processo viene assegnato un quanto di tempo (il valore di default è di 20 clock tick). Il quanto di tempo definisce un limite superiore al tempo di utilizzo della CPU nell'epoca. Comunque, in una stessa epoca, la CPU può essere assegnata ad un processo più volte, fino a quando il quanto non è esaurito. Un'epoca termina quando tutti i processi eseguibili (quelli nello stato `TASK_RUNNING`) hanno esaurito il loro quanto. Terminata un'epoca, ne inizia un'altra; all'inizio della nuova epoca vine calcolato il nuovo quanto di tempo da assegnare a ciascun processo. Questo è determinato come la somma tra il *base time quantum* (cioè, il valore predefinito del quanto di tempo) e metà dei clock tick eventualmente rimasti dall'epoca precedente. Infatti, processi che erano bloccati al momento in cui è terminata l'epoca, potevano non aver esaurito il quanto di tempo. Così facendo si assegna un bonus ai processi I/O-bound; sebbene un processo I/O-bound difficilmente esaurirà il quanto a sua disposizione, questo è utilizzato

per determinare la priorità del processo. In questo modo, il numero di clock tick di un processo che non ne consuma alcuni entro un'epoca può tendere asintoticamente a 40. Ai processi ordinari, cioè quelli che vengono gestiti con la politica `SCHED_OTHER`, è assegnata una priorità statica pari a 0; la priorità dinamica, invece, coincide con il numero di clock tick.

Per i processi real-time non vale la suddivisione del tempo in epoche. Ai processi real-time è assegnato un livello di priorità statica compreso tra 1 e 99, che non cambia durante l'esecuzione. Come conseguenza, i processi real-time hanno una priorità statica sempre superiore a quella dei processi ordinari. Dato che lo scheduler seleziona il processo a priorità più alta, un processo ordinario può essere eseguito solo se non ci sono processi real-time pronti per l'esecuzione.

I processi real-time possono appartenere ad una di due categorie:

- `SCHED_FIFO`: processi real-time con quanto di tempo illimitato. Questi processi lasciano la CPU solo se: 1) si bloccano; 2) terminano; 3) un processo a priorità più alta passa nello stato di pronto; 4) terminano;
- `SCHED_RR`: processi real-time soggetti a scheduling di tipo Round Robin.

Le informazioni necessarie allo scheduler, comprese quelle relative alla priorità, sono contenute nel descrittore di processo `task_struct`. In particolare, queste sono:

- `long int state`: stato del processo;
- `long int need_resched`: indica se è necessaria l'invocazione dello scheduler. Questo campo viene controllato al termine di ogni routine di servizio delle interruzioni;
- `long int policy`: può assumere i valori `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`, `SCHED_YIELD`;
- `long int rt_priority`: indica il livello di priorità statica per i processi real-time;
- `long int priority`: rappresenta il quanto di tempo (espresso in clock tick) assegnato ad un processo soggetto a timesharing;
- `long int counter`: è il numero di clock tick residui. La variabile è aggiornata dalla funzione `update_process_times()` ad ogni interruzione dell'orologio di sistema.

Le variabili `counter` e `priority` non sono utilizzate per i processi soggetti alla politica `SCHED_FIFO`.

Invocazione dello scheduler

Lo scheduler viene invocato tramite la funzione `schedule()`, che ha lo scopo di individuare il processo a maggiore priorità fra quelli pronti. La sua invocazione può avvenire in due modi:

- invocazione diretta (o *direct invocation*). Questa modalità viene eseguita ogni volta che il processo si blocca (stati `TASK_UNINTERRUPTIBLE` o `TASK_INTERRUPTIBLE`);
- invocazione ritardata (o *lazy invocation*). Questa ha luogo quando, al termine dell'esecuzione di una primitiva di sistema, la variabile `need_resched` indica che è necessario eseguire lo scheduler.

La variabile `need_resched` viene impostata se:

- durante l'esecuzione della funzione `update_process_times()` si verifica che `counter <= 0`;
- durante l'esecuzione della funzione `wake_up_process()` ci si accorge che è passato nello stato di pronto (`TASK_RUNNING`) un processo a priorità più alta di quello attualmente in esecuzione;
- il processo chiama la funzione `sched_yield()`.

L'esecuzione della funzione `schedule()` comprende i seguenti passi:

- completamento di eventuali system calls in corso (nel caso di kernel non preemptible non è consentita la commutazione di processo mentre il kernel sta eseguendo delle operazioni per conto di un processo) e operazioni di manutenzione del sistema;
- se il processo è di tipo `SCHED_RR` ed ha esaurito il quanto, viene posto in fondo alla coda e gli viene assegnato un nuovo quanto (per questo tipo di processi non vale la suddivisione in epoche);
- si analizza la coda dei processi pronti per determinare qual'è il processo a priorità maggiore. Questa selezione viene fatta mediante la funzione `goodness()`.

La funzione `goodness()` opera nel modo seguente:

- Se è in grado di selezionare un processo, allora viene invocato il dispatcher per eseguire l'assegnazione;
- altrimenti si inizia una nuova epoca.

Per un processo, la funzione `goodness()` può produrre uno dei seguenti risultati:

- $G = -1000$, ad indicare che il processo non deve essere selezionato;
- $G = 0$, se il processo ha esaurito il quanto;
- $0 < G < 1000$, se si tratta di un processo ordinario che non ha ancora esaurito il quanto di tempo. Il valore restituito rappresenta la sua priorità dinamica;
- $G >= 1000$, per un processo real-time. Se $G = 1000 + g$, allora g è il livello di priorità statica.

Nel caso in cui più processi assumano il valore massimo di `goodness`, viene selezionato il primo nella scansione della coda dei processi pronti.

In Fig.2.4 è mostrata l'organizzazione delle liste con priorità dei processi Linux. Lo scheduler ha come obiettivo di mantenere in esecuzione i processi nella coda a maggiore priorità.

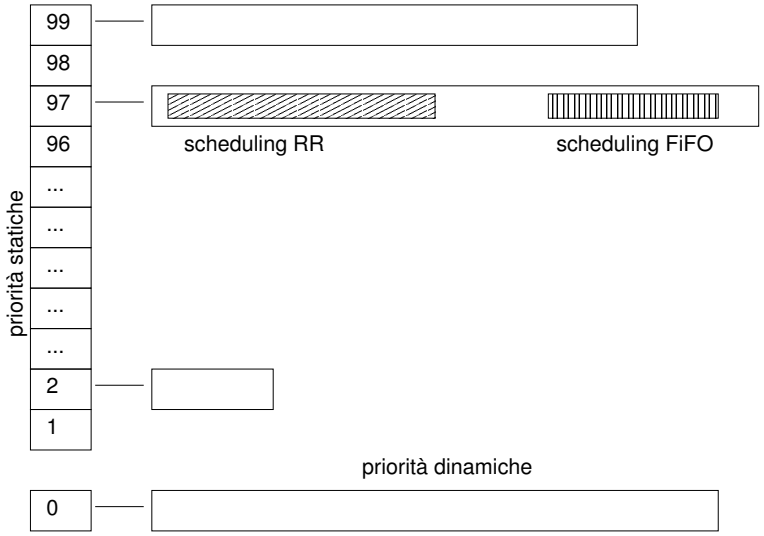

Figura 2.4: Code di priorità per i processi Linux. Le priorità statiche dei processi real-time sono sempre maggiori delle priorità statiche dei processi "normali" (non real-time). Infatti, i processi normali pur variando le loro priorità dinamiche in base all'uso dei clock tick all'interno di un'epoca, hanno comunque tutti priorità statica uguale a 0. Notare, infine, come processi real-time ad una stessa priorità possono essere schedulati con algoritmi diversi (parte con FIFO, e parte con RR).

Problemi

I principali problemi dell'algoritmo di scheduling di Linux sono:

- le prestazioni dello scheduler degradano al crescere del numero di processi. Infatti, le priorità (cioè i quanti) devono essere ricalcolate per tutti i processi al termine di ogni epoca;

- il valore predefinito per il quanto può essere eccessivo nei sistemi con carico elevato;
- il meccanismo di boost dei processi I/O-bound non è ottimale. Infatti, non tutti i processi richiedono interazione con l'utente e quindi processi per l'accesso a database, il trasferimento dati via rete, etc., potrebbero essere gestiti in modo diverso;
- il supporto per i processi real-time è limitato.

L'algoritmo di scheduling di Linux, a partire dalla versione 2.6 del kernel, è stato in parte modificato (in particolare, per quanto attiene la gestione delle priorità e dei processi real-time) al fine di risolvere alcuni dei problemi indicati.

Bibliografia

[1] S. Piccardi. http://gapil.firenze.linux.it.

[2] A.S. Tanenbaum. "I Moderni Sistemi Operativi," *Jackson Libri,* 2002.