Scheduling nei sistemi operativi

Index

Scheduling tradizionale di UNIX

Le politiche di scheduling mostrate finora al giorno d'oggi non sono più semplicemente applicate spesso infatti sono utilizzati come dei blocchetti su cui poi vengono costruiti gli scheduler moderni.
Nello scheduling di UNIX è stato introdotto il concetto di **priorità combinato con il round-robin**. Quindi l'idea è che un processo resta in esecuzione per al massimo un quanto di tempo pari ad un secondo (a meno che non termini o non si blocchi). Sono inoltre presenti diverse code, a seconda della priorità, e su ogni coda viene applicato il round-robin.
Per **risolvere il problema della starvation** di questo tipo di politica però si è fatto in modo che le priorità venissero ricalcolate ogni quanto di tempo in base a quanto tempo

un processo è rimasto in esecuzione (più è rimasto in esecuzione più diminuisce la priorità).

Le priorità iniziali sono basate sul tipo di processo:

- swapper (alta)
- controllo di un dispositivo di I/O a blocchi
- gestione di file
- controllo di un dispositivo di I/O a caratteri
- processi utente (basso)

Formula di Scheduling

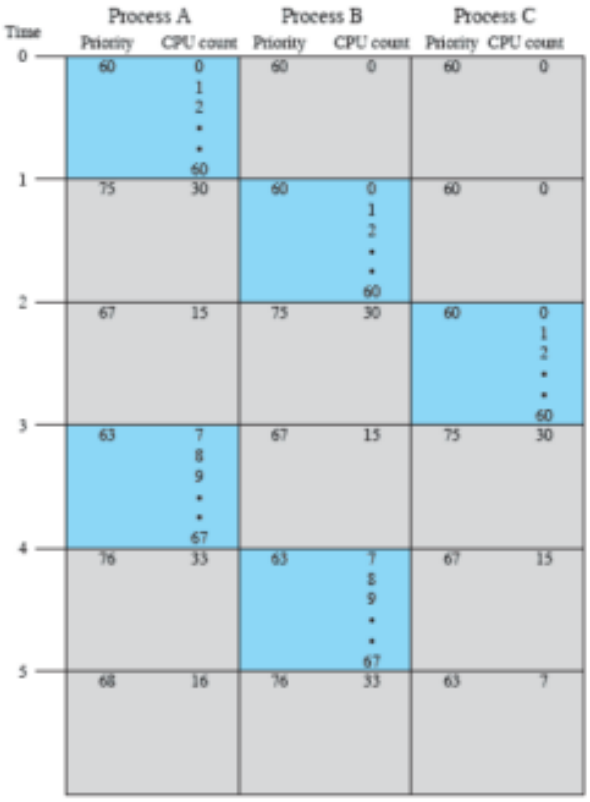
$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

$CPU_j(i) \rightarrow$ è una misura di quanto un processo j ha usato il processore nell'intervallo i, con exponential averaging dei tempi passati; per i running $CPU_j(i)$ viene incrementato di 1 ogni $\frac{1}{60}$ di secondo
$P_j(i) \rightarrow$ è la priorità del processo j all'inizio di i (più basso è il valore, più è alta la priorità)
$Base_j \rightarrow$ assegnato in base alle priorità iniziali del processo ($0 \leq Base_j \leq 4$)
$nice_j \rightarrow$ un processo può dire che il proprio valore di cortesia è maggiore di zero per auto-declassarsi, in modo tale da far passare avanti altri processi (usata prevalentemente per i processi di sistema)

Esempio di Scheduling su UNIX

$$Base_a = Base_b = Base_c = 60,\ nice_a = nice_b = nice_c = 0$$



Architetture multi-processore

Esistono varie architetture multi-processore al giorno d'oggi:

- Cluster
  architettura multiprocessore con memoria non condivisa (ogni processore ha la propria RAM) e la connessione con rete locale tra di essi è superveloce
- Processori specializzati (es. ogni I/O device ha un suo processore)
- **Multi-processore e/o multi-core**
  Questi condividono la RAM (ci sta un'unica RAM che tutti i processori utilizzano) e sono controllati da un solo SO (a differenza degli altri due)

Scheduler

Abbiamo sostanzialmente due possibilità: **assegnamento statico** e **assegnamento dinamico**. Per assegnamento si intende decidere quale processo va su quale processore

Assegnamento statico

Quando un processo viene creato gli viene assegnato un processore e fino alla terminazione del processo, questo viene eseguito sullo stesso processore.
La struttura dell'assegnamento statico è molto semplice, ci basterà infatti avere uno

scheduler per ogni processore e dunque l'overhead sarà molto basso; lo svantaggio sta nel fatto che un processore potrebbe rimanere idle, la distribuzione dei processi infatti potrebbe non essere equa

Assegnamento dinamico

Per migliorare lo svantaggio dell'assegnamento statico, un processo, nel corso della sua vita potrà essere eseguito su diversi processori. Seppur ragionevole la sua realizzazione è particolarmente complessa, specie se si vuole mantenere un overhead basso.
Per semplificare la realizzazione si potrebbe decidere di fare in modo di eseguire il SO su un processore fisso, lasciando che solo i processi utente possano cambiare. Lo svantaggio però sta nel fatto che questo potrebbe diventare un bottleneck e che mentre per i processi utente il sistema potrebbe funzionare con una failure di un processore, se fallisce quello del SO no.
Una seconda possibilità è quella di eseguire il SO non su un processore fisso ma seguendo le stesse regole dei processi utente, ma che richiederebbe più overhead

---

Scheduling in Linux

Nel corso degli anni lo scheduling di Linux è cambiato molteplici volte, quello qui presentato è uno scheduling in disuso da qualche anno.
Linux, per quanto riguarda lo scheduling, è alla ricerca di velocità di esecuzione, tramite semplicità di implementazione così da mantenere un overhead il più basso possibile. Per questo motivo in questo SO non sono presenti né long-term scheduler (anche se un suo embrione ovvero se viene creato un nuovo processo ma il sistema è già saturo), né medium-term scheduler (ci torneremo quando si parlerà di gestione della memoria).
Questo è un sistema ad assegnamento statico, ma rimane presente una routine che periodicamente ridistribuisce il carico se necessario

Runqueues e wait queues

In Linux ci sono le *runqueues* (la coda dei processi ready) e le *wait queues*
Le *wait queues* (coda dei blocked, plurale perché ci sta una coda per ogni evento) sono proprio le code in cui i processi sono messi in attesa quando fanno una richiesta che implichi l'attesa
Le *runqueues* (coda dei processi ready, plurale perché ci sta una coda per ogni priorità) sono quelle da cui pesca il dispatcher (short-term scheduler)

> **Notare**
>
> Le *wait queues* sono condivide dai processori, invece per le *runqueues* ogni processore ha le proprie

Politica di scheduling

Per quanto riguarda la politica di scheduling è sostanzialmente **derivata da quella di UNIX**: preemptive a priorità dinamica (decresce man mano che un processo viene eseguito, cresce man mano che un processo non viene eseguito) seppur con **alcune modifiche** per poter **migliorare la velocità** e per poter servire nel modo più approrpiato i processi real-time (se ci sono).
Linux istruisce l'hardware di mandare un timer interrupt ogni $1\ \mathrm{ms}$ ovvero **ogni quanto tempo Linux si deve fermare per decidere se deve fare qualcosa**; è stato infatti studiato che se fosse più lungo creerebbe problemi per i processi real-time, mentre se fosse più corto arriverebbero troppi interrupt e si spenderebbe troppo tempo in Kernel Mode. Il quanto di tempo per il round-robin è dunque un multiplo di $1\ \mathrm{ms}$

Tipi di processi

Linux riconosce tre tipi di processi in quanto ogni tipo di processo ha necessità diverse. Ma tutti possono essere sia CPU-bound che I/O-bound

Interattivi

non appena si agisce sul mouse o tastiera, per dare l'illusione di una risposta reattiva, bisogna dare il controllo al processo corrispondente in al massimo $150\ \mathrm{ms}$ altrimenti l'utente se ne accorge

Batch

vengono tipicamente penalizzati dallo scheduler, l'utente è infatti disposto ad aspettare un po' di può (compilazioni, computazioni scientifiche etc.)

Real-time

sono gli unici riconosciuti come tali da Linux infatti nel loro codice sorgente viene usata la system call `sched_setscheduler`, gli altri invece sono distinti in base a quante richieste all'I/O vengono fatte. Questi sono ad esempio audio/video ma normalmente sono usati solo dai KLT di sistema. Questi **non cambiano mai la priorità**

Classi di scheduling

In Linux sono presenti 3 diverse classi di scheduling

- `SCHED_FIFO` e `SCHED_RR` fanno riferimento ai processi real-time
- `SCHED_OTHER` tutti gli altri processi

Prima si eseguono i processi in `SCHED_FIFO` e `SCHED_RR`, poi quelli in `SCHED_OTHER`. Questo poiché le prima 2 classi hanno un livello di priorità da 1 a 99, la terza da 100 a 139. Questo vuol dire che ci sono 140 runqueues per ogni processore. Si passa dall'esecuzione sul livello n al livello $n+1$ solo se o non ci sono processi in n, o se nessun processo in n è `RUNNING`

Per quanto riguarda `SCHED_OTHER` a differenza di UNIX la preemption può essere dovuta a due casi:

- si esaurisce il quanto di tempo del processo attualmente in esecuzione
- un altro processo passa da uno degli stati blocked a `RUNNING` (in modo tale da favorire i processi interattivi)

Molto spesso capita che il processo che è appena diventato eseguibile verrà effettivamente eseguito dal processore e a seconda di quante CPU ci sono, può soppiantare il processo precedente questo perché probabilmente si tratta di un processo interattivo, cui bisogna dare precedenza

Un processo `SCHED_FIFO` essendo FIFO, non viene bloccato da interrupt ma viene preempted e rimesso in coda solo se:

- si blocca per I/O (o rilascia volontariamente la CPU)
- un altro processo passa da uno degli stati blocked a `RUNNING` ed ha priorità più alta

Altrimenti, non viene mai fermato.
`SCHED_RR` invece, come `SCHED_OTHER`, lavora su quanti di tempo