UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea in Ingegneria Informatica

LINUX: ANALISI DELL'EVOLUZIONE DELLO SCHEDULER

Tutor:

Chiar.mo Prof. Letizia Leonardi

Elaborato di Laurea di:

Alberto Cristallo

Anno Accademico 2012 – 2013

Sommario

Introduzione

Il kernel Linux è presente su una enorme quantità di device e di tipi di device: elettrodomestici, smartphone (ad esempio con Android, quest'ultimo installato su quasi un miliardo di device), tablet, personal computer, server, supercomputer, ma anche automobili, aerei, stazioni spaziali. Per soddisfare questi requisiti, sono necessarie capacità estreme di scalabilità, portabilità, flessibilità. Per questo motivo, si ritiene interessante sia analizzarne la struttura generale, sia analizzare l'evoluzione di un singolo componente, lo scheduler.

Infatti, se è vero che ora il kernel Linux è utilizzato su tutta questa vasta gamma di device (e oltre), bisogna anche considerare le origini del kernel Linux. Il kernel Linux nasce come progetto di un hobbista nel 1991, ad opera di Linus Torvald, totalmente privo di portabilità e quindi anche senza necessità di operare in contesti diversi da quello di un pc desktop. Da allora, grazie soprattutto al rilascio del primo kernel come software libero e all'intervento di molte migliaia di sviluppatori, il kernel ha raggiunto le caratteristiche attuali.

La scelta di analizzare le caratteristiche dello scheduler è volta a scoprire come può un singolo componente adattarsi sia alle esigenze di sistemi quasi privi di interattività quali i supercomputer, sia alla necessità di fornire una risposta nei tempi stretti obbligati dei sistemi real-time, sia alla richiesta di reattività del normale utente su smartphone e personal computer.

Si è deciso di lavorare sul kernel 3.9.7, rilasciato il 20/06/2013, il quale è l'ultimo kernel stabile al momento della stesura dell'elaborato, per poter avere una visione quanto più aggiornata si potesse.

1 Cos'è un kernel

Ogni computer, per funzionare, ha bisogno di un insieme di programmi, chiamato sistema operativo. Nel sistema operativo c'è un programma fondamentale per il funzionamento di tutti gli altri, il *kernel*. Lo scopo del kernel è quello di fare da cuscinetto tra la macchina fisica, fatta di circuiti, e il resto dei programmi. Quando un'applicazione ha bisogno di interagire con la macchina, per esempio perché necessita di RAM, o di accedere al disco rigido, fa richiesta al kernel, il quale poi si interfaccerà direttamente con l'hardware per fornire il servizio richiesto. Il kernel quindi gestisce direttamente le risorse hardware, decidendo quali utilizzare o non utilizzare.

Tra i kernel è possibile operare una distinzione tra kernel monolitici e microkernel.

Figura 1: Schematizzazione di un kernel

La maggior parte dei kernel Unix presenta struttura monolitica; questo significa che ogni strato del kernel è integrato in un unico programma kernel. Con questa opzione, il kernel viene integralmente caricato in memoria RAM all'avvio del sistema. Questo comporta il vantaggio di rendere il sistema più rapido, ma consuma una quantità maggiore di RAM rispetto ai microkernel.

I microkernel invece caricano in RAM solo le parti fondamentali del kernel, generalmente quelle necessarie a caricare e gestire gli altri livelli del kernel, i quali, se non utili al momento, possono essere rimossi dalla RAM. La gestione dei vari livelli, però, si rivela solitamente più onerosa. Oltre al minor uso di RAM, la modularità del microkernel offre il vantaggio di offrire una più facile portabilità, dato che di solito le parti che dipendono da un determinato hardware sono ben separate.

Il kernel Linux è un kernel monolitico, ma internamente strutturato in moduli. Questa configurazione permette di coniugare i vantaggi del kernel monolitico e del microkernel. Dal microkernel mutua la modularità, dalla quale deriva maggiore facilità di sviluppo e di portabilità su altri hardware; oltre a ciò, i singoli moduli possono essere rimossi dalla RAM se non più necessari. Inoltre il kernel Linux conserva i vantaggi del kernel monolitico, e cioè che l'intero kernel viene gestito come un unico processo, eliminando le penalizzazioni dovute alle comunicazioni tra processi tipiche del microkernel.

2

3 Processi

Un *processo* è un'istanza di un programma in esecuzione. Ogni programma può essere eseguito da più processi, per esempio perché ci sono più istanze in esecuzione contemporaneamente. Generalmente, ad ogni processo è riservato il proprio spazio degli indirizzi in memoria, non accessibile dagli altri processi.

3.1 Sistemi multiprogrammati, multithread, multiutente

Nei primi sistemi operativi, tutte le risorse del sistema venivano dedicate ad un singolo programma, il quale le teneva occupate fino al termine della sua esecuzione. Dato che l'esecuzione di operazioni di I/O richiede generalmente tempi diversi ordini di grandezza maggiori rispetto alle operazioni della CPU (Central Process Unit), era frequente che la CPU rimanesse inutilizzata per lunghi periodi di tempo, in attesa della fine delle operazioni di I/O.

Per ovviare al problema dell'attesa, sono stati introdotti i sistemi multiprogrammati, ai quali sono seguiti i sistemi multithread e i sistemi multiutenti.

Nei sistemi *multiprogrammati*, l'elaboratore esegue contemporaneamente più processi, ognuno di norma dotato del proprio spazio degli indirizzi. La contemporaneità di esecuzione può essere reale o fittizia: reale (*overlapping*), se abbiamo più processori, ognuno dei quali esegue un diverso processo; fittizia (*interleaving*), se il processore alterna l'esecuzione di ognuno dei vari processi per quantità di tempo talmente piccole da rendere il procedimento trasparente alla percezione umana.

Anche con la suddivisione in processi, però, corriamo il rischio di andare incontro a tempi morti. Per questo, è stata introdotta una sottodivisione all'interno dei processi: il *thread*. Il thread è, come il processo, l'esecuzione di una sequenza di istruzioni, con la differenza che i thread appartenenti ad uno stesso processo condividono tutti lo stesso spazio degli indirizzi, utilizzando, quindi, gli stessi dati.

Un'altra funzionalità che il kernel può implementare è la multiutenza. Un sistema *multiutente* è un sistema che permette l'utilizzo contemporaneo e indipendente di diverse applicazioni da parte di due o più utenti.

Il kernel Linux è un sistema misto: è cioè un sistema multiprogrammato, multithread, multiutente.

Inoltre, definiamo *task* come l'unità di base di sequenza di istruzioni e dati, sia esso un thread o un processo.

3.2 User Mode/ Kernel Mode

Molte delle CPU moderne implementano due o più modalità di esecuzione per un processo. Il kernel Linux ne implementa due: la *User Mode* e la *Kernel Mode*.

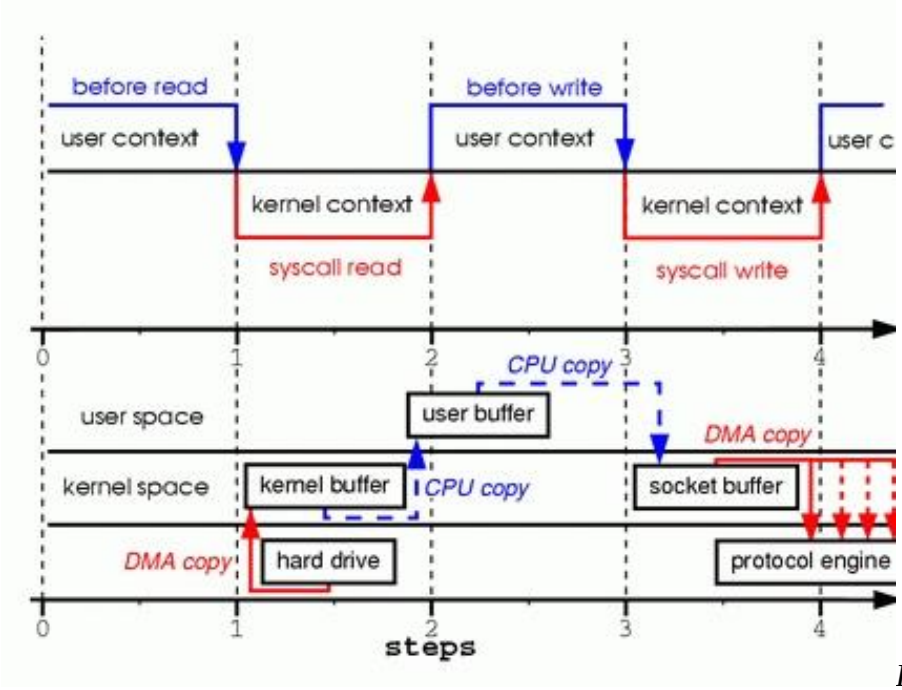

Figura 2: Passaggio tra User e Kernel Mode in caso di lettura e scrittura

Quando un processo è eseguito in User Mode, non ha accesso diretto all'hardware, né a strutture dati del kernel, né a programmi del kernel. Quando invece un processo viene eseguito in Kernel Mode, non si applica nessuna restrizione.

Qualora un processo in User Mode necessiti un servizio dal kernel, passa in modalità Kernel Mode solo per le operazioni strettamente necessarie al servizio. Dopodiché, ritorna in modalità User Mode.

3.3 Scheduling

Nei sistemi multiprogrammati è necessaria una componente che gestisca l'esecuzione dei vari processi. L'azione si divide in due parti:

- *policy* (politica): metodo con cui si decide quando e a quale processo allocare quale risorsa. Il componente che prende le decisioni è lo *scheduler*;
- *meccanismo*: strumento con cui vengono attuate le decisioni prese in base ad una certa policy. Nel caso dello scheduling, il componente corrispondente è il *dispatcher*, che

attua il *context switch* (commutazione di contesto) tra i vari processi.

Nell'implementazione, le due parti non sono necessariamente distinte.

3.4 Preemption e Kernel Reentrant

In alcuni sistemi operativi, lo scheduler aspetta che un processo finisca da sé per poter allocare la CPU al processo successivo. Il kernel Linux, invece, prevede che un processo possa essere interrotto per poter allocare la CPU al processo successivo. Questa operazione si chiama *preemption* (prelazione).

Per la gestione dei processi, ogni processo è rappresentato da un *process descriptor* (descrittore di processo). Quando il kernel interrompe un processo, salva i contenuti correnti di diversi registri di processo nel process descriptor, tra cui:

- i registri PC e SP, che indicano rispettivamente quale istruzione del programma si era arrivati ad eseguire, e l'inizio della memoria stack;
- i registri di uso generale e floating point;
- i registri di controllo del processore, che contengono informazioni sullo stato della CPU;
- i registri per la gestione della memoria RAM.

Quando lo scheduler decide di riallocare la CPU ad un processo, i registri della CPU vengono ricaricati con i valori che avevano precedentemente, e l'esecuzione del task riprende dal punto in cui si era interrotta.

Il kernel Linux è un kernel *reentrant*. Questo significa che possano essere interrotti dei processi anche se sono in Kernel Mode.

4 Gestione della memoria

Come visto nel capitolo precedente, ogni task necessita di uno spazio degli indirizzi in memoria per poter operare. Il kernel Linux si occupa di gestire la memoria in modo trasparente per i task, e per far questo si avvale della *virtual memory*.

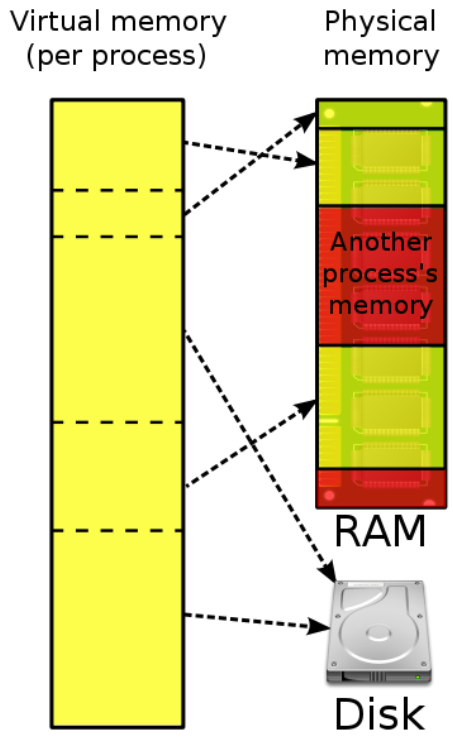

Figura 3: Rapporto tra Virtual Memory e memoria reale

La memoria virtuale è uno strato software che si interpone tra i processi e l'hardware, prendendo le richieste dei processi e facendo richiesta al Memory Management Unit (MMU), che è il componente hardware che gestisce la memoria.

L'introduzione della memoria virtuale ha portato vari benefici:

- allocazione di quantità di memoria superiori a quella fisica reale, dando quindi la possibilità di eseguire programmi che altrimenti non potrebbero essere eseguiti;
- i processi possono eseguire del codice anche se questo codice non è completamente caricato in memoria fisica;
- una singola zona di memoria fisica, dove ad esempio sono memorizzati una libreria o dei dati di un programma, può essere utilizzata da più processi.

5 File system

Un file è una sequenza persistente di byte, generalmente utilizzato per contenere informazioni. Il kernel non ne interpreta il contenuto nella sua gestione. Per la gestione del suo contenuto, i programmatori generalmente si affidano a librerie che implementano strutture di un più alto livello di astrazione.

Il *filesystem* è il sistema con cui i file vengono archiviati, letti, modificati. In base alla complessità del filesystem, dipendono, ad esempio, le possibilità di manipolare i file, le dimensioni e il numero massimi consentiti di file, la possibilità di disporli in gerarchie più o meno complesse. Ad esempio, in Figura 4 abbiamo una struttura ad albero.

Figura 4: Esempio di filesystem ad albero

5.1 Virtual File System

Per la gestione del filesystem, Linux implementa un *Virtual File System* (VFS). Il VFS è un livello di astrazione, orientato agli oggetti, superiore al normale filesystem, che presenta vari vantaggi.

Il motivo iniziale per cui è stato deciso di creare un Virtual File System è il voler fornire il supporto a moltissimi filesystem diversi, oltre che uniformare la gestione di diversi componenti, come ad esempio:

- sequenze di byte su disco fisso, generalmente con lo scopo di contenere informazioni;
- le directory stesse in cui sono contenuti i file;
- periferiche hardware;
- filesystem particolari come i filesystem remoti (NFS, AFP, SMB,...) o i filesystem volatili in memoria RAM;
- strutture dati del kernel come ad esempio i processi.

Per far questo, il VFS possiede una lunga serie di API (Application Programming Interface) per poter operare sui file, come ad esempio:

- `creat(...)`, per la creazione di un file;
- `open(...)`, per l'apertura di un file;
- `rename(...)`, per rinominare un file;
- `lseek(...)`, per cambiare posizione all'interno di un file.

Quando una di queste API viene chiamata, il VFS controlla a quale tipo di filesystem il file appartenga e chiama la funzione specifica per quel filesysyem. Alcune di queste funzioni comportano un'operazione effettiva del filesystem, altre, come ad esempio `lseek`, no.

5.2 Strutture del VFS

Per operare, il VFS, ricorre ad alcuni oggetti o strutture.

Ne abbiamo quattro principali:

- `struct file`: contiene le informazioni riguardanti un file aperto, tra cui il path, la posizione corrente nel file (file pointer), e un set di istruzioni con cui è possibile operare con un file;
- `struct dentry`: quando un file viene aperto, viene istanziata una dentry per ogni nodo del path, compresa la foglia, e cioè il file stesso. La struttura contiene, ad esempio, dei puntatori al nodo padre e, se ci sono, ai nodi figli, oltre che il suo stesso nome e un puntatore al proprio inode;
- `struct inode`: è la struttura del VFS conservata in memoria principale, corrispondente all'inode conservato su disco. Contiene, tra le altre cose, una lista di dentry che fanno riferimento a questo inode, una lista di operazioni possibili, e un address space object, cioè un oggetto che gestisce le varie pagine per l'inode all'interno della page cache;
- `struct superblock`: è il descrittore di un filesystem montato, ne contiene i metadati e le operazioni.

6 Scheduling nel kernel Linux

I primi scheduler Linux avevano un design minimale, non particolarmente scalabile perché non focalizzato su nessuna architettura complessa.

Il kernel 1.2 utilizzava una policy di scheduling round-robin. Lo scheduling round-robin prevede che tutti i processi eseguibili siano inseriti in una coda circolare, chiamata *runqueue*. Lo scheduler prende il primo processo dalla coda e gli alloca la CPU per un *quanto* o *intervallo di tempo* o *timeslice* predeterminato. Se il processo sta ancora eseguendo delle operazioni al termine del quanto, la CPU riceve una richiesta di preemption e il processo viene interrotto e aggiunto al termine della coda. Se il processo invece termina o si sospende prima della fine del quanto, è il processo stesso a rilasciare la CPU. In entrambi i casi, lo scheduler alloca la CPU al processo successivo nella coda, come mostrato in Figura 5.

Figura 5: CPU con scheduling round-robin

Lo scheduler del kernel 2.2 introdusse la divisione dei processi in classi, dividendoli in task

real-time, task non-preemptible e task non-real-time. Introdusse anche il supporto per il multiprocessore simmetrico (*SMP*), cioè per sistemi dotati di più CPU o core che condividono una stessa memoria centrale come mostrato in Figura 6, utilizzando un'unica runqueue che alloca di volta in volta una CPU libera. Il problema di questa soluzione è che, nell'allocazione, non si considera la CPU su cui il task era stato eseguito precedentemente, rendendo inutili i dati precedentemente caricati nella cache della CPU.

Figura 6: Sistema con architettura SMP

Nel kernel 2.4 troviamo uno scheduler che operava in un tempo O(n), e cioè in un tempo direttamente proporzionale al numero dei task. Il tempo viene diviso in epoche, e, in ogni epoca, ad ogni processo viene assegnato il proprio intervallo di tempo. Se un task non utilizza pienamente il suo intervallo di tempo, metà del tempo rimanente viene aggiunto al suo successivo intervallo di tempo, così da avere un maggiore tempo di esecuzione nell'epoca successiva. Per scegliere il task successivo, lo scheduler valuta ogni task con una funzione di bontà, impiegandoci quindi un tempo direttamente proporzionale al numero dei task. Questo scheduler quindi è semplice quanto inefficiente. Inoltre è privo di scalabilità, è inadatto ai sistemi real-time, e non sfrutta le nuove architetture come i processori multi-core.

Il primo scheduler del kernel Linux 2.6.0 è lo O(1) scheduler, progettato con lo scopo di eliminare la necessità di dover valutare ogni task per poter decidere il successivo. Il suo obiettivo ideale è quindi quello di avere un tempo decisionale indipendente dal numero dei task.

A partire dal kernel Linux 2.6.33 è stato introdotto invece il *Completely Fair Scheduler* (*CFS*), scheduler di default anche nel kernel 3.9.7.

7 Lo scheduler O(1)

L'introduzione dello scheduler O(1) nel kernel 2.6 aveva come scopi principali l'aumento della scalabilità e e la capacità di sfruttare meglio l'architettura SMP.

7.1 Struttura dello scheduler

Nello scheduler O(1), ogni CPU ha una sua runqueue, chiamata *active runqueue*, composta da 140 liste di priorità, ognuna della quale rispetta una politica FIFO[1], come si può vedere in Figura 7. Le prime 100 liste sono riservate a task real-time, le ultime 40 ai task dell'utente. Inoltre, ogni task ha la sua timeslice.

Quando un task rilascia la CPU, viene posto nella *expired runqueue*, e gli vengono assegnate

[1]First In, First Out: i task vengono seguiti in ordine di arrivo

una nuova priorità e una nuova timeslice. Quando una determinata lista di priorità della active runqueue viene esaurita, avviene uno scambio tra questa lista e quella presente nella expired runqueue avente la stessa priorità[2], rendendo quindi quest'ultima una lista di priorità della active runqueue.

Per scegliere il task a cui allocare la CPU per la sua timeslice, lo scheduler prende il primo processo dalla lista non vuota con priorità più alta[3]. In questo modo, lo scheduler non deve scorrere l'intera lista degli n task, bensì al massimo per le 140 liste di priorità, rendendo il tempo di ricerca indipendente dal numero dei task.

Figura 7: Struttura delle liste di priorità dello scheduler O(1)

7.2 Gestione della priorità

Un problema che può sorgere negli scheduler con priorità è la *starvation*. Un task va incontro a starvation qualora non riesca ad andare mai in esecuzione. In questo caso, se un task avesse sempre davanti a sé task con priorità superiore, rischierebbe di non poter mai prendere controllo della CPU. Per prevenire questo fenomeno, lo scheduler O(1) ha la possibilità di cambiare la priorità di un task. Questo avviene favorendo i task *I/O bound* (il cui tempo di computazione

[2]Tramite scambio di puntatori
[3]Processo ottimizzato con l'utilizzo di una bitmap, che permette l'utilizzo di istruzioni hardware del tipo find-first-bit-set

dipende principalmente da attese di Input e Output) e penalizzando i task *CPU bound* (quelli in cui il tempo è speso principalmente in utilizzo di CPU). Questo perché i task I/O bound generalmente utilizzano la CPU solo per avviare un'operazione di I/O, andando in stato di sleep in attesa del completamento.

Per capire se un task è I/O bound o CPU bound, lo scheduler utilizza un'euristica basata sull'interattività, controllando quanto tempo ha speso in esecuzione rispetto a quanto tempo ha speso in attesa. A questo punto, lo scheduler può incrementare o ridurre la priorità di massimo 5 punti.

Questo meccanismo, inoltre, migliora la reattività del sistema, dato che i processi di I/O tendono a guadagnare priorità.

Da notare però che questo meccanismo di premiazione dei task I/O bound vale solo per le ultime 40 liste di priorità, vale a dire vale solo per i task dell'utente.

7.3 Miglior supporto delle architetture SMP

Gli scheduler Linux precedenti allo O(1) hanno un moderato supporto delle architetture SMP. C'è un'unica runqueue condivisa da tutti i processori. Non appena una CPU si libera, prende il task successivo dalla runqueue comune. Un problema è che, durante il processo di scelta da parte di una CPU, la runqueue è inaccessibile alle altre CPU, il che comporta una possibile attesa da parte delle altre CPU. Un altro problema è che i task sono completamente slegati dalle singole CPU, rendendo generalmente inutili i dati memorizzati in cache per un dato task nel caso in cui il task venga eseguito in una CPU diversa rispetto alla precedente.

Nello scheduler O(1), invece, abbiamo una active runqueue per CPU, con la rispettiva expired runqueue. Questo elimina le possibili attese per l'accesso alla runqueue, dato che ognuno possiede la propria. Inoltre, un task tende ad essere eseguito su una stessa CPU, rendendo efficace l'uso della memoria cache.

Come detto, ogni task viene collocato in una certa active runqueue di una determinata CPU. Al momento dell'assegnazione, non si sa quale sarà la durata del task, il che generalmente porta, dopo un po' di tempo, a una sostanziale differenza del carico tra le varie CPU del sistema. Per questo motivo, ogni 200ms, una funzione di *load balancing* (bilanciamento del carico) controlla i carichi di ogni CPU. Se sono sbilanciati, lo scheduler riassegna i task nel tentativo di bilanciare il carico delle CPU.

Il problema di questa soluzione è che, durante il cambio di CPU, invalidiamo i dati presenti nella cache della CPU.

8 Il Completely Fair Scheduler

A partire dal kernel 2.6.23, nel kernel Linux troviamo di default il Completely Fair Scheduler. L'idea alla base del CFS è quella di modellare una ideale CPU multi-processo, nella quale tutti i task vengono eseguiti in contemporanea, su un hardware reale, dove i task vengono eseguiti uno per volta, e lo fa cercando di mantenere una certa equità nel tempo di esecuzione dei vari processi.

Per ottenere l'equità nel tempo di esecuzione dei vari processi, lo scheduler tiene traccia della quantità di tempo in cui un task ha avuto la possibilità di utilizzare la CPU, tramite una variabile del task chiamata *vruntime* (da *virtual runtime*, cioè tempo di esecuzione virtuale). Il virtual runtime è calcolato come il tempo che il task avrebbe passato in esecuzione in una CPU multi-processo ideale, e cioè è dato dal tempo di runtime reale diviso il numero di task in esecuzione.

Inoltre, il CFS prevede una *sleeper fairness*, e cioè i processi ritenuti sleeper, ovvero quelli che passano molto tempo inattivi, possono utilizzare la CPU per una timeslice superiore rispetto a quella che riceverebbero in uno scheduler che non tenesse conto dei processi sleeper.

8.1 Struttura generale dello scheduler CFS

Al contrario degli scheduler precedenti, che inseriscono i task in una coda, la runqueue del CFS è strutturata come un albero rosso-nero, un albero in cui il figlio di un nodo nero è rosso, e viceversa. Una caratteristica fondamentale di un albero rosso-nero, rappresentato in Figura 8, è che tutti i percorsi dalla radice a una qualsiasi delle foglie contengono lo stesso numero di nodi neri. La diretta conseguenza di questa caratteristica è che le operazioni su tale albero sono di una complessità di ordine O(log n) anche nel caso peggiore.

Figura 8: un albero rosso-nero

Quando un nuovo task viene inserito nell'albero, la sua variabile vruntime viene inizializzata con il valore della variabile della runqueue *min_vruntime*, che contiene il valore minimo delle vruntime dei task della runqueue. In questo modo, il nuovo task viene posizionato il più possibile a sinistra nell'albero senza però possedere un valore troppo basso rispetto agli altri task.

Quando il CFS deve scegliere quale task eseguire, sceglie quello posizionato più a sinistra nell'albero, che è quello con il vruntime più basso, e quindi quello che è stato eseguito di meno dalla CPU. Dopo ogni esecuzione, il vruntime viene incrementato e il task si sposta verso destra, lasciando la possibilità di esecuzione ad altri task.

8.2 Policy di scheduling

Il CFS implementa tre policy di scheduling per i task, come si può vedere nel file linux-3.9.7/kernel/sched/fair.c:

- SCHED_NORMAL: la policy di scheduling per tasks regolari;
- SCHED_BATCH: in questa modalità viene concesso più tempo consecutivo ad ogni processo, migliorando lo sfruttamento della cache e minimizzando il numero di task switch, riducendo però l'interattività. Ottimo per i processi di batch;
- SCHED_IDLE: i task idle sono i task con minore priorità possibile. I task con policy SCHED_BATCH o SCHED_IDLE non attuano mai preemption nei confronti di task non idle.

8.3 Alcuni dettagli implementativi del CFS

Per analizzare la struttura del Completely Fair Scheduler, si farà riferimento, come detto, al kernel 3.9.7.

Tutti i task sono rappresentati da una struttura, definita nel file linux-3.9.7/include/sched.h, di cui è riportato un pezzo della definizione:

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;
#ifdef CONFIG_SMP
    struct llist_node wake_entry;
    int on_cpu;
#endif
    int on_rq;
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
…
};
```

Questa struttura descrive completamente un task, contenendo ad esempio le variabili di stato del task, le variabili con i flag di processo, le variabili per la gestione degli handler dei segnali, le variabili per la gestione della priorità, le strutture contenenti la classe di scheduling, e altro. Dato che un task non è necessariamente eseguibile, non ci sono campi relativi al CFS. Per tenere traccia delle informazioni relative al CFS, abbiamo la struttura `sched_entity se`.

La struttura `sched_entity se`, riportata qui sotto, presenta i campi utili per la gestione da parte del CFS, come ad esempio il vruntime, valore che permette la gestione equa del CFS.

```
sched_entity {
    struct load_weight  load;       /* for load-balancing */
    struct rb_node      run_node;
    struct list_head    group_node;
    unsigned int        on_rq;
    u64         exec_start;
    u64         sum_exec_runtime;
    u64         vruntime;
…
};
```

Importante per la gestione dell'albero rosso-nero è la semplice struttura `rb_node`, composta da tre valori, e cioè il colore del padre e due puntatori ai figli: se il puntatore ad un figlio ha valore NULL, il figlio è una foglia dell'albero rosso-nero.

Per indicare la radice dell'albero, invece, abbiamo la struttura `rb_root`, che contiene come unico valore un puntatore ad una struttura di tipo `rb_node`.

Entrambe le strutture le troviamo contenute in linux-3.9.7/include/rbtree.h:

```
struct rb_node {
    unsigned long  __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
};
struct rb_root {
    struct rb_node *rb_node;
};
```

In linux-3.9.7/kernel/sched/sched.h troviamo invece definita la struttura `cfs_rq`, che presenta i

campi relativi alla runqueue del CFS. Da notare i campi load, che conserva la somma dei pesi dei task della runqueue, nr_running che conserva il numero dei task in coda, min_vruntime, che contiene la più bassa vruntime presente nella runqueue, task_timeline che contiene la radice dell'albero rosso-nero e *rb_leftmost che punta al task più a sinistra nell'albero.

```
struct cfs_rq {
    struct load_weight load;
    unsigned int nr_running, h_nr_running;

    u64 exec_clock;
    u64 min_vruntime;
#ifndef CONFIG_64BIT
    u64 min_vruntime_copy;
#endif

    struct rb_root tasks_timeline;
    struct rb_node *rb_leftmost;

    /*
     * 'curr' points to currently running entity on this cfs_rq.
     * It is set to NULL otherwise (i.e when none are currently running).
     */
    struct sched_entity *curr, *next, *last, *skip;
...
};
```

La funzione schedule(), che serve a invocare una preemption nei confronti del task corrente, invece, è situata all'interno di linux-3.9.7/kernel/sched/core.c.

In Figura 9 è rappresentato il collegamento tra le strutture sopradescritte.

Figura 9: Collegamenti tra le strutture in riferimento all'albero rosso-nero

8.4 Classi di scheduling

Insieme all'introduzione del CFS, sono state introdotte le classi di scheduling. Ogni classe definisce una serie di funzioni per la gestione, come ad esempio una funzione per aggiungere

un task, una per eliminarlo, una per decidere quale task sarà il prossimo ad essere eseguito. Nel kernel 3.9.7, oltre allo scheduler CFS, troviamo, nel file linux-3.9.7/kernel/sched/rt.c, l'implementazione delle policy FIFO e round-robin (SCHED_FIFO e SCHED_RR), che seguono lo standard POSIX. Queste ultime due policy utilizzano 100 runqueue per i 100 livelli di priorità (contrariamente al precedente scheduler, che ne aveva 140) e non utilizzano la expired runqueue.

L'utilizzo di queste classi di scheduling è trasparente per il core dello scheduler: esiste una serie di funzioni che il core chiama per l'esecuzione dello scheduler, le quali attivano una serie di operazioni dipendente dal tipo di classe a cui il task appartiene. Ogni `task_struct`, infatti, contiene la struttura `sched_class`, che specifica la policy con cui un task dovrà essere trattato. Ecco una serie non completa di queste funzioni con la relativa descrizione:

- `enqueue_task(...)`: chiamata quando un task diventa eseguibile. Inserisce il task, nella forma di una struttura `sched_entity`, in coda all'albero rosso-nero, e incrementa il valore della variabile `nr_running`;
- `dequeue_task(...)`: quando un task diventa non eseguibile, il task viene tolto dall'albero ma ne viene tenuta traccia. La variabile `nr_running` viene decrementata;
- `yield_task(...)`: questa funzione è una combinazione di `dequeue_task` e `enqueue_task`, il che porta il task alla destra dell'albero;
- `check_preempt_curr(...)`: quando un task passa dallo stato non eseguibile allo stato eseguibile, questa funzione controlla se questo task deve attuare preemption nei confronti del task attualmente in esecuzione;
- `pick_next_task(...)` : questa funzione seleziona il task che dovrà essere eseguito successivamente al corrente;
- `set_curr_task(...)`: questa funzione viene chiamata quando un task cambia la sua classe di scheduling o gruppo di task.

Conclusione

Dall'analisi generale del kernel, è emerso che la grande portabilità del kernel Linux è dovuta alla presenza di livelli di astrazione tali da rendere possibile una trattazione uniforme di ogni device. L'organizzazione modulare, inoltre, rende molto più semplice l'estensione del kernel, ad esempio tramite driver.

Dall'analisi dell'evoluzione dello scheduler è emerso come le nuove esigenze di scalabilità e reattività abbiano portato ad una evoluzione del kernel che però non ne ha pregiudicato la sostanziale semplicità di base. Con l'introduzione del CFS si è introdotto un altro livello di astrazione, quello delle classi di scheduling, rendendo così più semplice lo sviluppo di nuovi scheduler e dando la possibilità di implementare policy di scheduling ad hoc senza dover modificare sostanzialmente il core dello scheduler.

È interessante notare inoltre come il principio di equità alla base del CFS sia, per quanto semplice, funzionante ed efficiente su una grande varietà di device, fornendo sia buona reattività nel caso desktop o mobile, sia rapidità nei sistemi real-time, sia la possibilità di eseguire processi batch senza l'eccessiva intrusione dello scheduler.

Bibliografia

- Daniel P. Bovet, Marco Cesati - *Understanding the Linux Kernel*, Third Edition, O'Reilly, 2005
- M. Tim Jones - *Inside the Linux Scheduler* <http://www.ibm.com/developerworks/library/l-scheduler/>
- M. Tim Jones - *Inside the Linux 2.6 Completely Fair Scheduler*

<http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/>

- Kernel Linux, versione 3.9.7 <https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.9.7.tar.xz>