UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea in Ingegneria Informatica

SCHEDULING IN LINUX: DALLE ORIGINI AL COMPLETELY FAIR SCHEDULER

Tutor:

Chiar.mo Prof. Letizia Leonardi

Elaborato di Laurea di:

Imad Ayoub

Anno Accademico 2022-2023

Sommario

INTRODUZIONE

La crescente complessità degli applicativi software pone sempre nuove sfide in termini di potenza di calcolo e capacità fornite dai sistemi informatici attuali. Innanzitutto, i processori sono chiamati a fornire prestazioni sempre più elevate. Tuttavia, è importante sottolineare che non ci si può limitare solamente all'incremento di potenza di calcolo. Un aspetto altrettanto rilevante è rappresentato dalla gestione efficiente delle risorse *hardware* da parte dei sistemi operativi e in particolare delle CPU. Quindi, un aspetto altrettanto importante, oltre le prestazioni delle CPU, è la loro gestione cioè l'assegnazione delle CPU ai processi (scheduling).

Questo elaborato esplora il mondo dello scheduling nel kernel Linux, dando una panoramica generale dell'evoluzione dello scheduler a partire dalla versione 0.01 fino alla versione attuale, la 6.4.12 (al 23/08/2023). In particolare, nel primo capitolo si introduce in generale il concetto di processo e del suo descrittore, passando poi a porre l'attenzione sia sul descrittore di un processo in Linux che in quali stati un processo può trovarsi durante il suo ciclo di vita in Linux. Il capitolo seguente, in modo assolutamente generale, introduce il concetto di scheduling e le principali politiche con cui può essere attuato. Con il terzo capitolo, l'attenzione si sposta sul primo scheduler implementato in Linux, esplorandone la struttura e analizzandone il funzionamento. Il quarto capitolo analizza lo scheduler presente in Linux 2.4, versione rilasciata ufficialmente il 4/01/2001, con particolare attenzione alle strutture dati coinvolte, alle classi di processi, alle priorità e ai meccanismi di assegnazione del tempo di esecuzione. Il quinto capitolo è dedicato all'approfondimento dello scheduler O(1), introdotto con la versione 2.6 di Linux rilasciata il 17/12/2003. Infine, l'elaborato si conclude con una panoramica sul *Completely Fair Scheduler*, introdotto con la versione 2.6.23 di Linux, rilasciata il 9/10/2007, nel quale vengono esplorate l'organizzazione del codice, le dinamiche di scheduling tra diverse classi di processi e l'importanza del concetto di "*virtual runtime*" nel garantire un trattamento equo tra i processi in esecuzione. L'elaborato termina poi con capitolo conclusivo.

1 PROCESSI

Tutti i sistemi operativi usano una fondamentale astrazione: il *processo*. Un processo può essere definito come un'istanza di un programma in esecuzione. In altri termini, quando un programma è in esecuzione questo determina un flusso di operazioni denominato processo.

È importante distinguere i programmi dai processi, in quanto un programma può essere eseguito da più processi contemporaneamente.

Nel seguito della trattazione i termini attività, task e processo saranno da intendersi sinonimi.

1.1 Processi CPU-bound e Processi I/O-bound

Analizzando i processi si nota che essi richiedono un lasso di tempo per eseguire le istruzioni, successivamente si mettono in attesa di una richiesta di input/output (I/O) e questo schema si può ripetere più volte durante la vita di un processo; il periodo computazionale tra due richieste di I/O è detto *CPU-burst* **(si veda figura 1.1).**

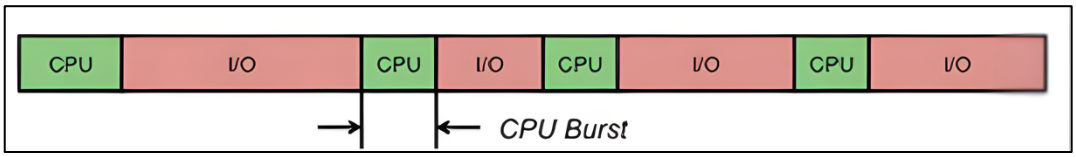

Figura 1.1 CPU-bursts

Un'attività che richiede brevi *CPU-burst* intervallati da richieste di I/O, come ad esempio un semplice editor di testo, prende il nome di I/O-*bound process*. Analogamente se un processo richiede lunghi *CPU-burst,* come ad esempio un software per il calcolo matematico, in cui l'unica operazione di I/O rilevante è il caricamento dei dati iniziali, è detto CPU-*bound process*.

1.2 Descrittore di processo

Per gestire i task, è fondamentale che il sistema operativo abbia un meccanismo di gestione per ognuno di essi. Affinché ciò sia possibile, alla nascita di una nuova attività, il sistema associa ad essa un descrittore di processo.

Il descrittore, in Linux, è una struttura, *task_struct*, che tiene traccia di tutti gli attributi e le informazioni riguardanti il processo stesso. Il sistema, inoltre, dispone di una *task_list*: una lista contenente tutti i descrittori dei task attivi.

Più in dettaglio, durante la vita di un processo, il descrittore tiene traccia del contesto di esecuzione, come lo spazio di indirizzamento e la tabella dei file aperti, lo stato di esecuzione, le relazioni con gli altri processi e altre informazioni utili al sistema per garantire che ogni attività operi in modo sicuro e isolata dalle altre.

Dando uno sguardo al kernel Linux, il descrittore è notevolmente più complicato. Oltre a contenere numerosi attributi di processo esso contiene molteplici puntatori ad altre strutture dati, le quali a loro volta contengono altri riferimenti a differenti strutture (**si veda figura 1.2).**

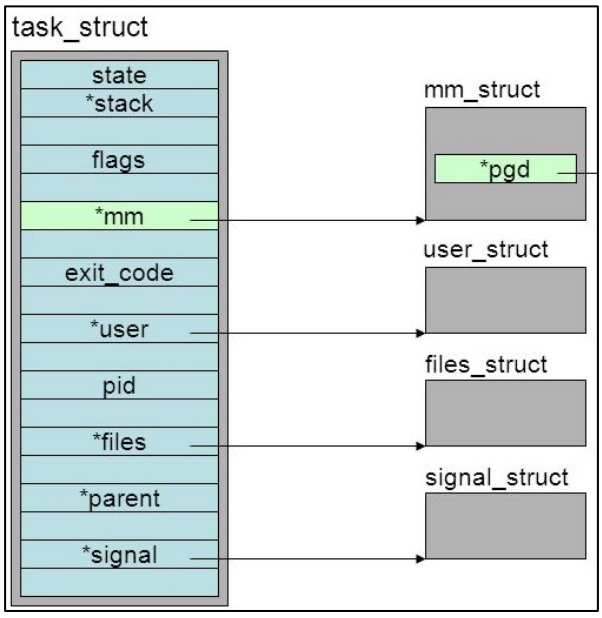

Figura 1.2 descrittore di processo Linux

1.3 Identificatore di processo

L'identificazione di un processo è fondamentale per l'intero sistema operativo, vari attributi possono identificarlo univocamente, quale, ad esempio, l'indirizzo in memoria del descrittore. Questa scelta, sebbene corretta, non sarebbe ottimale poiché utilizzare un identificativo a 32 bit in esadecimale non risulterebbe di facile comprensione. La scelta progettuale è stata semplice: associare ad ogni attività un numero progressivo partendo dall'identificatore "1" e successivamente ad ogni figlio, o nuovo task, associare il numero dell'ultimo processo creato incrementato di uno. Tale numero è denominato *Process ID*, o con la sigla *PID*.

Il *Process ID* è salvato in memoria all'interno del descrittore di processo, nel campo denominato *pid*.

1.4 Stato di un processo

Lo stato di un processo è memorizzato dentro al suo descrittore: nella versione attuale di Linux un task non può trovarsi in stati differenti contemporaneamente.

Nel file <linux/sched.h> **(si veda figura 1.3)** sono definite cinque macro corrispondenti ai cinque possibili stati in cui si può trovare un processo:

1. **TASK_RUNNING:**
   il processo è in esecuzione o è pronto per essere eseguito.
   Nel contesto di Linux, è importante notare che non esiste alcuna differenziazione tra lo stato di un processo pronto per essere eseguito, comunemente denominato *runnable*, e lo stato in cui il processo è effettivamente in esecuzione, noto come *running*. Nonostante la loro distinzione concettuale, entrambi gli stati sono identificati tramite la stessa macro di sistema. Linux tiene traccia di tutti i processi nello stato TASK_RUNNING mediante una coda di processi denominata *runqueue*, eccezion fatta per l'unico processo in esecuzione sulla CPU.

2. **TASK_INTERRUPTIBLE:**
   il processo è sospeso finché certe condizioni non verranno soddisfatte. Per esempio, la gestione di un interrupt hardware, la ricezione di un segnale o il soddisfacimento di una richiesta di una risorsa precedentemente occupata da un altro processo.
3. **TASK_UNINTERRUPTIBLE:**
   il processo è sospeso in maniera analoga al precedente stato, tuttavia, la sua ripresa non è possibile tramite la semplice ricezione di un segnale.
4. **TASK_STOPPED:**
   l'esecuzione del processo è stata interrotta in seguito al ricevimento di un segnale, per esempio SIGSTOP.
5. **TASK_TRACED:**
   l'esecuzione del processo è stata interrotta da un debugger, per esempio con la system call *ptrace()*.

Oltre ai cinque stati sopra citati ne esistono due ulteriori che vengono memorizzati, sempre nel descrittore di processo, nel campo *exit_state*. Un task raggiunge i due ulteriori stati al termine della sua esecuzione, che sono:

1. **EXIT_ZOMBIE:**
   l'esecuzione del processo è terminata ma il processo padre non ha ancora chiamato la system call *wait()* per ricevere le informazioni del processo.
2. **EXIT_DEAD:**
   è lo stato finale, il processo viene rimosso dal sistema dopo la system call *wait()* del padre.

Il valore della variabile state è assegnato dal kernel con una semplice operazione di assegnazione:
tsk->state = **TASK_RUNNING**;

```
207   #define TASK_RUNNING              0
208   #define TASK_INTERRUPTIBLE        1
209   #define TASK_UNINTERRUPTIBLE      2
210   #define __TASK_STOPPED            4
211   #define __TASK_TRACED             8
212   /* in tsk->exit_state */
213   #define EXIT_DEAD                 16
214   #define EXIT_ZOMBIE               32
```

Figura 1.3 righe 207-214 del file <linux/sched.h>, kernel Linux 4.0

In seguito, al fine di semplificare, si farà riferimento a solo tre stati possibili: *runnable, running* e *waiting*. Un processo si trova nello stato *runnable* se è pronto per essere eseguito, ovvero è contenuto nella *runqueue*, nello stato *waiting* se è in attesa del soddisfacimento di opportune condizioni e infine nello stato *running* se è in esecuzione sulla CPU.

2 SCHEDULING

All'interno di un sistema multiprogrammato, la CPU risulta essere inattiva durante l'esecuzione di operazioni di I/O di un processo, a meno che non venga assegnata ad un altro task. L'entità preposta a determinare l'attribuzione della CPU ai differenti processi è lo *scheduler*.

Viene richiesto l'intervento dello scheduler principalmente al verificarsi di quattro eventi:

1. Il processo corrente passa dallo stato *running* allo stato *waiting* poiché ha effettuato una richiesta di I/O.
2. Il processo corrente termina.
3. Il processo ha esaurito il suo intervallo di tempo corrente di esecuzione.
4. Un processo ha terminato l'attesa di una certa condizione, il sistema può decidere di interrompere l'attività corrente e assegnare la CPU al processo pronto.

Uno scheduler è definito *preemptive* se tiene conto dell'evento 4. sopra indicato e quindi se ha la capacità di essere invocato da un interrupt e interrompere il processo *running* per fare spazio a un altro processo *runnable*. Viceversa, uno scheduler è detto *non-preemptive se* non può interrompere l'esecuzione di un processo.

2.1 Politiche di scheduling

Il compito principale dello scheduler consiste nel prendere una decisione riguardo l'assegnazione della CPU a un determinato processo, secondo una politica di scheduling ben definita.

Alcune politiche di scheduling che possono essere adottate sono:

1. **First In First Out:**
   come introdotto nel primo capitolo i processi in stato *runnable* sono memorizzati in una coda denominata *runqueue*. La scelta in questo caso è alquanto semplice, il primo processo ad entrare nella coda è il primo ad essere

posto in stato *running* e continua a eseguire finché non completa la sua attività o entra nello stato *waiting* **(si veda figura 2.1)**. L'uso di questo criterio di scelta può risultare svantaggioso in quanto il tempo di attesa medio di un processo per essere eseguito risulta essere elevato.

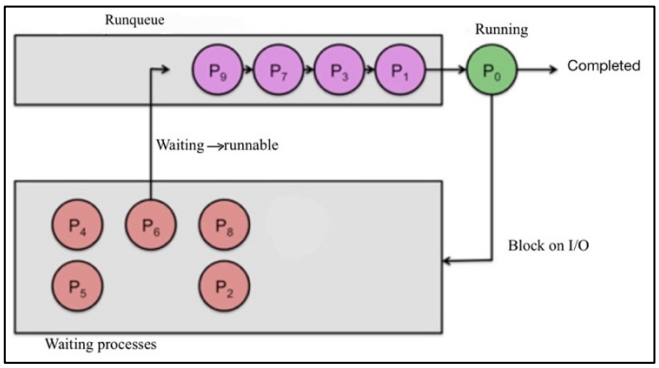

Figura 2.1 First In First Out

2. **Round Robin:**

   è una versione *preemptive* di First In First Out, i processi sono sempre disposti in una coda ma al processo *running* è concessa l'esecuzione per un tempo limitato, usualmente denominato *time-slice* o *quantum.* In altre parole, se l'attività in esecuzione non termina dopo un *time-slice* viene interrotta e riposta in fondo alla coda, poiché si è verificato l'evento 3. sopra indicato. Invece se il processo richiede un'operazione di I/O lo si pone nello stato *waiting* **(si veda figura 2.2)** come usuale e quando potrà tornare in esecuzione avrà a disposizione un nuovo *time-slice* intero.

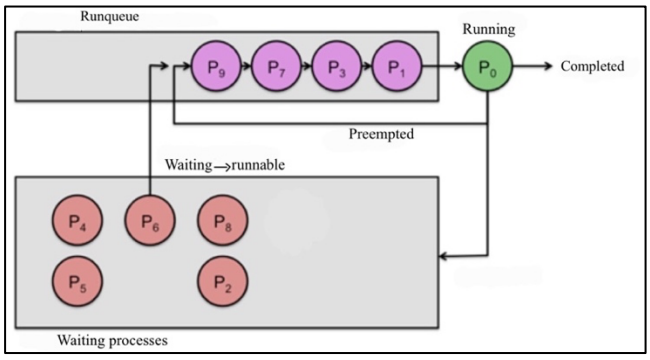

Figura 2.2 Round Robin

3. **Priority scheduling:**

   a differenza delle due politiche sopra menzionate, in questo approccio si assegna ad ogni attività una priorità, fissata staticamente alla nascita in base alle sue caratteristiche, e ad ogni istante è eseguito il processo *runnable* a priorità massima. Un problema che sorge usando questo tipo di criterio è noto come *starvation*, tale fenomeno si verifica quando un processo a bassa priorità non viene mai eseguito in quanto ci sono sempre processi con priorità maggiore pronti ad essere eseguiti. Una metodologia usata per risolvere il problema della *starvation* consiste nell'usare priorità dinamica, consentendo al sistema di modificare la priorità assegnata alla creazione per prevenire la *starvation* e quindi così facendo si permette l'esecuzione di processi creati con bassa priorità che nel corso del tempo però appunto viene innalzata.

4. **Multilevel feedback queues:**

   con questo approccio i processi vengono suddivisi in classi di priorità e associati a differenti code per ognuna di esse. Così facendo, si categorizzano e si separano i processi in base alle loro caratteristiche. I processi con priorità alta vengono assegnati alle code con priorità maggiore e analogamente i processi con priorità bassa vengono assegnati alle code con priorità inferiore. In genere, ad ogni task viene associato un *time-slice* differente in accordo con la priorità della coda: più è alta la priorità, più il *quantum* di tempo è inferiore.

Lo scheduler quando deve scegliere il processo da porre in stato *running* sceglie il primo processo disponibile dalla coda con priorità maggiore.
In generale i processi I/O-*bound* sono posti nelle code a priorità alta e i processi CPU-*bound* sono posti nelle code a priorità minore. L'algoritmo usa due semplici regole:

1. Alla nascita un processo viene posto nella coda a priorità maggiore.
2. Se un processo non finisce il suo *time-slice* rimane nella stessa coda, viceversa se un processo utilizza tutto il suo *time-slice* viene spostato nella coda a priorità inferiore.

Siccome con questo approccio i processi con lunghi *CPU-burst* tendono a usare l'intero *quantum* di tempo a disposizione, essi verranno interrotti e posti in una coda con priorità inferiore. Invece i processi con piccoli CPU-*burst*, ovvero tutti i processi interattivi, non usando l'intero *quantum* rimarranno in code a priorità maggiore.

Nel caso in cui siano presenti numerosi processi interattivi risulta sempre disponibile un task nella coda a priorità maggiore, si incorre quindi nel problema della *starvation,* in quanto i processi CPU-*bound* non verranno mai posti in esecuzione. Per risolvere questa problematica, lo scheduler adotta una politica di assegnamento dinamica della priorità, garantendo ai processi delle code inferiori di risalire la gerarchia delle code e quindi essere eseguiti. Questo *aging priority* si ottiene alterando la regola 2: se un processo non esaurisce il suo *quantum* al suo risveglio viene inserito nella coda di priorità superiore.

2.2 Dispatcher

Dopo l'individuazione del processo da far eseguire sulla CPU da parte dello *scheduler,* il meccanismo attraverso cui si finalizza l'assegnazione della CPU al task selezionato è noto come *dispatcher*. Esso rappresenta un componente fondamentale del kernel, la cui funzione consiste nell'effettuare il passaggio dal modo supervisore a utente, nel gestire il cambio di contesto (*context switch)* tra i processi e quindi effettuare il giusto salto all'istruzione del processo che deve essere eseguito. Il *dispatcher* nella fase di

context switching si occupa di salvare il contesto di esecuzione del processo *running* nel suo descrittore e successivamente, di ripristinare lo stato del task selezionato dallo *scheduler.*

Una delle più grandi sfide che deve affrontare il *dispatcher* è l'*overhead*, ovvero il tempo impiegato per applicare il *context switch*. Nel caso in cui il sistema dovesse gestire un numero elevato di processi, l'*overhead* potrebbe risultare un problema poiché causerebbe grandi ritardi e ciò ridurrebbe notevolmente le prestazioni globali del sistema. Infatti, considerando la politica round-robin e sue derivate, se il *quantum* a disposizione per ogni processo fosse molto piccolo, la CPU risulterebbe utilizzata per la maggior parte del tempo ad effettuare *context switching*. Per ridurre l'*overhead*, è fondamentale minimizzare le operazioni necessarie per effettuare un context switch e scegliere un *time-slice* adeguato, molto maggiore del tempo impiegato per eseguire tale operazione.

2.3 Scheduling in sistemi multi-processore

La trattazione fino ad ora si è concentrata, per semplicità, sullo scheduling dei processi in sistemi con singoli processori. Tuttavia, attualmente la maggior parte dei calcolatori hanno architetture ben più complesse, composte da più processori. Si presenta, quindi, un nuovo problema di "schedulazione dei processi tra le CPU".

Un primo approccio alla schedulazione tra le CPU, in un sistema multi-processore, consiste nell'assegnare ad un singolo processore tutte le funzioni dello scheduler mentre i restanti eseguono solo il codice utente. Questo approccio viene detto *Asimmetric MultiProcessing* ed è relativamente semplice dato che un solo processore accede alla *runqueue* condivisa assegnando a mano a mano i processi ai vari processori.

Un secondo approccio utilizza il *Symmetric MultiProcessing* (SMP), in cui ciascun processore ha accesso alla *runqueue*. Nel caso in cui la *runqueue* fosse condivisa si verrebbero a creare problemi di concorrenza per l'accesso ad essa, per questo nella maggior parte dei sistemi attuali ogni processore ha la sua coda di processi *runnable*. L'uso di code differenti introduce però il problema del bilanciamento del carico,

ovvero tentare di mantenere uniforme il carico di lavoro distribuito tra tutti i processori. Per fare ciò esistono principalmente due tecniche: *push migration* e *pull migration*. Con la *push migration* un task dedicato controlla periodicamente il carico su ciascun processore e, se trova uno sbilanciamento, distribuisce il carico in modo uniforme (*push*) spostando i processi da code di processori sovraccarichi a processori inutilizzati o meno carichi. La *pull migration*, invece, si ha quando un processore si ritrova con una *runqueue* vuota; quindi, lo scheduler sottrae (*pull*) processi dalle altre code per spostarli nella *runqueue* del processore inutilizzato.

2.4 Affinità del processore

Quando è in esecuzione un processo accede ai dati più recenti contenuti nella cache del processore a cui è stato assegnato. Di conseguenza, gli accessi successivi alla memoria da parte del task sono spesso soddisfatti nella memoria *cache*. Se in un determinato istante, il processo passa a un altro processore, il contenuto della *cache* del vecchio processore deve essere invalidato e la *cache* del nuovo processore deve essere ripopolata. A causa dell'elevato costo di invalidare e ripopolare le *cache*, la maggior parte dei sistemi tenta di evitare la migrazione dei processi da un processore all'altro mantenendo un processo in esecuzione sullo stesso processore. Questo è noto come affinità del processore, ovvero un processo ha un'affinità per il processore su cui è attualmente in esecuzione.

L’affinità del processore ha due diverse realizzazioni:

1. *Soft affinity*: quando un sistema operativo tenta di mantenere un processo in esecuzione sullo stesso processore, ma non garantisce che lo farà. In questo caso, il sistema operativo tenterà di mantenere un processo su un singolo processore, ma è possibile che un processo esegua la migrazione tra i processori.
2. *Hard affinity*: quando un sistema operativo mantiene un processo in esecuzione sempre sullo stesso processore.

3 IL PRIMO SCHEDULER IN LINUX

Il primo *scheduler* Linux **(si veda figura 3.1)** nasce nel 1991 con la prima versione v0.01 del sistema operativo, usava un design minimale e ovviamente non era pensato per un elevato numero di processi gestiti dal kernel.

```
64   *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
65   * tasks can run. It can not be killed, and it cannot sleep. The 'state'
66   * information in task[0] is never used.
67   */
68  void schedule(void)
69  {
70          int i,next,c;
71          struct task_struct ** p;
72
73  /* check alarm, wake up any interruptible tasks that have got a signal */
74
75          for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
76                  if (*p) {
77                          if ((*p)->alarm && (*p)->alarm < jiffies) {
78                                  (*p)->signal |= (1<<(SIGALRM-1));
79                                  (*p)->alarm = 0;
80                          }
81                          if ((*p)->signal && (*p)->state==TASK_INTERRUPTIBLE)
82                                  (*p)->state=TASK_RUNNING;
83                  }
84
85  /* this is the scheduler proper: */
86
87          while (1) {
88                  c = -1;
89                  next = 0;
90                  i = NR_TASKS;
91                  p = &task[NR_TASKS];
92                  while (--i) {
93                          if (!*--p)
94                                  continue;
95                          if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
96                                  c = (*p)->counter, next = i;
97                  }
98                  if (c) break;
99                  for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
100                         if (*p)
101                                 (*p)->counter = ((*p)->counter >> 1) +
102                                                 (*p)->priority;
103         }
104         switch_to(next);
105 }
```

Figura 3.1 primo scheduler Linux v0.01

3.1 Struttura generale dell'algoritmo

Ad ogni chiamata di *schedule()* per prima cosa venivano scansionati tutti i processi nel sistema con il ciclo for dalla riga 75 alla riga 83, in particolare:

3. Tra le righe 76 e 79 si controllava se i processi avevano il *timer alarm* attivo e se fosse scaduto. Più in dettaglio, il *timer alarm* veniva impostato nel caso venisse usata la primitiva *alarm* **(si veda figura 3.2)**; quindi la scadenza del timer veniva controllata attraverso un confronto con la variabile *jiffies*, inizializzata a zero al *boot* del sistema e incrementata di un'unità ad ogni *timer interrupt.*

```
171  int sys_alarm(long seconds)
172  {
173          current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
174          return seconds;
175  }
```

Figura 3.2 routine di servizio primitiva alarm(), file sched.c Linux v0.01

In altri termini, quando *jiffies* raggiungeva il valore di *alarm* voleva dire che il timer era scaduto. In tal caso, alla riga 78 veniva aggiornata la *pending signal bitmap* con il segnale SIGALRM, ovvero il 14-esimo bit di *signal* a partire da destra viene posto a "1" e successivamente azzerato il timer.

4. Tra le righe 81 e 82 si risvegliavano tutti i task in stato TASK_INTERRUPTIBLE che avevano ricevuto almeno un segnale, ovvero che avevano almeno un bit a "1" nella *pending signal bitmap*.

Successivamente si aveva un ciclo while diviso in due fasi:

1. Tra le righe 92 e 98 venivano scansionati tutti i processi, cercando il task *runnable* con il valore di *counter* massimo e se trovato veniva poi posto in stato *running* con la chiamata *switch_to(next).* Se invece non ci fossero processi

runnable disponibili, poiché la variabile *next* è inizializzata al valore zero alla riga 89, viene posto in esecuzione il task "0", un task "*dummy*" sempre in stato *runnable* e con *counter* maggiore o uguale a zero.

2. Se tutti i task *runnable* hanno *counter* uguale a zero, tra le righe 99 e 103, si ha un ciclo for usato per re-inizializzare il valore della variabile *counter* di tutti i processi. In particolare, i processi *runnable* ricevono un nuovo valore di *counter* pari alla loro priorità, tutti gli altri che non sono *runnable* e che quindi non hanno *counter* uguale a zero ricevono la metà del valore di *counter* più la loro priorità.

Alla nascita di un processo, *counter* era inizializzata con il valore della priorità del task stesso. Il *counte*r del processo *running* viene decrementato fino ad un minimo di zero tutte le volte che si ha un *timer interrupt* tramite la funzione *do_timer()* **(si veda figura 3.3)** definita nel file sched.c. Nel caso in cui il valore *counter* del processo *running* raggiungesse il valore zero, viene richiesto l'intervento dello *scheduler*. Quindi, *counter* indica quanti colpi di clock di sistema ha a disposizione il processo, ovvero il suo *time-slice*.

```
159  void do_timer(long cpl)
160  {
161          if (cpl)
162                  current->utime++;
163          else
164                  current->stime++;
165          if ((--current->counter)>0) return;
166          current->counter=0;
167          if (!cpl) return;
168          schedule();
169  }
```

Figura 3.3 funzione do_timer() in sched.c Linux v0.01

3.2 Analisi del primo scheduler Linux

Dall'analisi del paragrafo 3.1 si osserva che già dalle prime versioni di Linux la scelta progettuale è stata quella di favorire l'interattività tramite una politica "Round Robin pesata", in quanto ogni processo che si blocca per I/O alla prossima schedulazione ha a disposizione un *quantum* di tempo maggiore e quindi veniva scelto per primo.

Alcuni problemi di questo primo scheduler potevano essere:

5. L'algoritmo aveva una complessità O(n) poiché ad ogni chiamata di *schedule()* bisognava scorrere tutta la lista dei processi attivi per determinare il migliore da eseguire.
6. La priorità era fissata staticamente alla nascita di ogni processo e non esisteva un meccanismo di modifica dinamica in base alla natura del processo da parte del kernel. L'unica modifica della priorità statica può essere effettuata con il comando *nice* da parte di un utente mediante il quale però la priorità può solo essere diminuita **(si veda figura 3.3)**.

```
207   int sys_nice(long increment)
208   {
209           if (current->priority-increment>0)
210                   current->priority -= increment;
211           return 0;
212   }
```

Figura 3.8 routine di servizio del comandi nice, file sched.c Linux v0.01

7. Esisteva un'unica lista globale di processi attivi, la quale in un sistema multiprocessore potrebbe causare problemi di concorrenza. Questo nelle prime versioni di Linux non era un vero problema dato che non supportava architetture SMP (*Symmetric multiprocessor*, si veda paragrafo 2.3).

4 SCHEDULER IN LINUX 2.4

I primi scheduler Linux presentavano un design simile tra loro, essi iteravano la *runqueue* in cerca del processo ottimale da avviare. In questo capitolo verrà esaminato lo scheduler nella release 2.4 di Linux rilasciata ufficialmente il 4/01/2001.

4.1 Strutture dati utilizzate

Per permettere una ricerca efficiente dei processi esistenti, il kernel Linux utilizzava differenti liste. Ogni lista consisteva in puntatori a descrittori di processo (*struct task_struct**). In particolare, esso usava una *circular doubly linked list* **(si veda figura 4.1)** per tenere traccia di tutti i processi esistenti nel sistema. Per implementare la struttura dati, i descrittori di processo contenevano due campi: *prev_task* e *next_task,* i quali erano, rispettivamente, i puntatori al precedente e successivo descrittore dei task nel sistema.

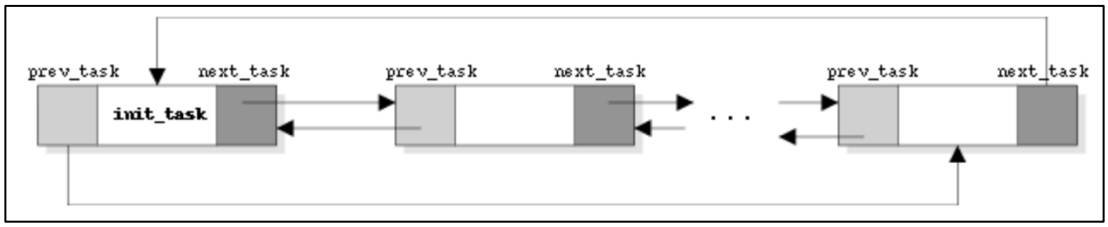

Figura 4.1 circular doubly linked list

Inoltre, il kernel teneva traccia dei processi in stato *runnable* in una differente *circular doubly linked list,* la *runqueue*. A tale scopo, per permetterne l'implementazione, i descrittori di processo contenevano due campi appositi: *prev_run* e *next_run*.

4.2 Classi di processi

In Linux 2.4 i task erano suddivisi in tre classi di processi: real-time *non-preemptible,* real-time *preemptible* e *default* ovvero non real-time.

Lo scheduler implementava tre *scheduling policy* **(si veda figura 4.2)**, una per ogni classe di processo.

```
113   #define SCHED_OTHER             0
114   #define SCHED_FIFO              1
115   #define SCHED_RR                2
```

Figura 4.2 scheduling policy, Linux 2.4 file sched.h

La classe di un processo era salvata nel campo *policy* all'interno del suo descrittore.

Lo scheduler operava in maniera differente in accordo con la *policy* di un task:

8. I task SCHED_OTHER erano i processi *default*, per questa categoria lo scheduler operava in maniera analoga alla *policy* "Round Robin pesata" descritta nel capitolo 3.
9. I task SCHED_FIFO seguivano la *policy* First-In-First-Out.
10. I task SCHED_RR seguivano la *policy* Round-Robin.

4.3 Priorità e time-slice

Linux con l'introduzione delle classi di processi, introdusse anche un meccanismo di assegnamento della priorità differente: per i processi SCHED_FIFO e SCHED_RR la priorità era determinata dal campo *rt_priority* contenuto nel descrittore di processo di ogni task nel sistema, il suo valore variava da 1 (priorità minima) a 99 (priorità massima). Per i processi SCHED_OTHER, il valore contenuto in *rt_priority* non era utilizzato e la loro priorità era salvata nel campo *nice* che può assumere valori compresi tra -20 (priorità massima) e 19 (priorità minima).

Inoltre, il campo *nice* era utilizzato dal sistema per l'assegnamento del *quantum*: il sistema assegnava a ciascun task un *time-slice*, salvato nel campo *counter*, che dipendeva dalla *nice* del task stesso. Il *time-slice* aumentava in maniera inversamente proporzionale alla *nice*.

4.4 Funzionamento

Ad ogni invocazione dello scheduler, esso controllava se il processo corrente era SCHED_RR, se così fosse stato e avesse terminato il suo *quantum* sarebbe stato posto in fondo alla *runqueue* con un nuovo *time-slice* in maniera proporzionale alla *nice*. Successivamente iterava su tutti i processi contenuti nella *runqueue*. Ad ogni processo era assegnata una "*goodness*" che determinava quanto era favorevole porlo in stato *running*. Il processo scelto era il task con "*goodness"* maggiore.

La "*goodness"* di un processo era determinata tramite la funzione *goodness()* che ritornava uno dei valori mostrati in **Tabella 4.1:**

Valore di ritorno	**Descrizione**
-1000	non selezionare il processo.
-1	il processo ha richiesto il rilascio della CPU tramite *sched_yield()*
0	il processo ha terminato il suo *time-slice.*
0 < x < 1000	g*oodness* per i processi SCHED_OTHER.
> 1000	*goodness* per i processi SCHER_FIFO e SCHER_RR.

Tabella 4.1 valori di ritorno funzione goodness()

La funzione **(si veda figura 4.3)** iniziava controllando se il bit SCHED_YIELD era impostato nella *policy* del processo, il bit veniva impostato dalla primitiva *sched_yield()* e indicava che il processo voleva essere de-schedulato. Successivamente la funzione calcolava il valore di ritorno differenziando il calcolo per i processi in accordo con la loro classe:

- per i processi SCHED_OTHER, in prima approssimazione il valore di ritorno era calcolato in accordo con gli intervalli di clock per l'esecuzione rimasti al processo. Questo indica che la *goodness* di un processo *default* diminuiva in maniera proporzionale al *time-slice*. Infine, al valore di ritorno veniva sommata la priorità del processo. Linux 2.4 supportava le architetture SMP. Per questo, il valore di ritorno era influenzato anche dal processore in cui veniva posto in esecuzione il task.
- Per i processi SCHED_FIFO e SCHED_OTHER il valore di ritorno era 1000 + *rt_priority*.

La *goodness* dei processi real-time non dipendeva dalla *nice* ma dal valore *rt_priority*, così facendo i processi real-time venivano sempre selezionati per primi dallo scheduler.

```
137  static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
138  {
139          int weight;
140
141          /*
142           * select the current process after every other
143           * runnable process, but before the idle thread.
144           * Also, dont trigger a counter recalculation.
145           */
146          weight = -1;
147          if (p->policy & SCHED_YIELD)
148                  goto out;
149
150          /*
151           * Non-RT process - normal case first.
152           */
153          if (p->policy == SCHED_OTHER) {
154                  /*
155                   * Give the process a first-approximation goodness value
156                   * according to the number of clock-ticks it has left.
157                   *
158                   * Don't do any other calculations if the time slice is
159                   * over..
160                   */
161                  weight = p->counter;
162                  if (!weight)
163                          goto out;
164
165  #ifdef CONFIG_SMP
166                  /* Give a largish advantage to the same processor...   */
167                  /* (this is equivalent to penalizing other processors) */
168                  if (p->processor == this_cpu)
169                          weight += PROC_CHANGE_PENALTY;
170  #endif
171
172                  /* .. and a slight advantage to the current MM */
173                  if (p->mm == this_mm || !p->mm)
174                          weight += 1;
175                  weight += 20 - p->nice;
176                  goto out;
177          }
178
179          /*
180           * Realtime process, select the first one on the
181           * runqueue (taking priorities within processes
182           * into account).
183           */
184          weight = 1000 + p->rt_priority;
185  out:
186          return weight;
187  }
```

Figura 4.3 funzione goodness, file sched.c, Linux 2.4

5 O(1) SCHEDULER

Dalla versione 2.6 di Linux, rilasciata ufficialmente il 17/12/2003, il modulo dello scheduler fu completamente riprogettato e si poneva l'obbiettivo di:

- Implementare un algoritmo con complessità O(1).
- Migliorare la scalabilità in architetture SMP assegnando ad ogni processore la sua *runqueue*.
- Migliorare l'affinità in architetture SMP assegnando gruppi di task allo stesso processore e spostandoli solo per risolvere problemi di carico.
- Favorire i processi interattivi indipendentemente dal carico.

5.1 Runqueue

Come discusso, la *runqueue* è la struttura dati base dello scheduler. In Linux 2.6 La struttura era definita nel file /kernel/sched.c come *struct runqueue*.

Ogni *runqueue* conteneva due priority array: *active* ed *expired*. Entrambi definiti nel file /kernel/sched.c come *struct prio_array* **(si veda figura 5.1)**. L'*active* array conteneva i task che non avevano esaurito il loro *time-slice*. Viceversa, l'*expired* array conteneva i task che avevano esaurito il loro *time-slice*.

Inoltre, ogni priority array era composto da:

- Una variabile *nr_active* per tenere traccia del numero di processi contenuti in esso.
- Una coda di processi per ogni livello di priorità, da 139 (priorità minima) a 0 (priorità massima). La struttura *list_head* era l'implementazione di una *doubly linked list* definita nel file /include/linux/list.h.
- Una bitmap di BITMAP_SIZE, questa conteneva un bit per ogni livello di priorità. Usando 5 parole da 32-bit, si aveva un totale di 160 bit disponibili. Se il bit i-esimo era "1", allora nella coda i-esima era presente un processo *runnable*.

```
136   struct prio_array {
137           int nr_active;
138           unsigned long bitmap[BITMAP_SIZE];
139           struct list_head queue[MAX_PRIO];
140   };
```

Figura 5.1 struct prio_array, file sched.c, Linux 2.6

La **figura 5.2** rappresenta in modo concettuale le due strutture dati sopra citate (si noti che dei 160 bit ne vengono utilizzati solo 140).

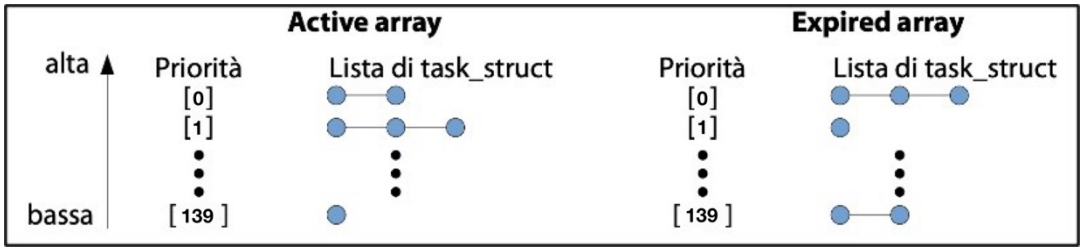

Figura 5.2 graficazione active ed expired array.

Per migliorare la scalabilità e l'affinità nelle architetture SMP ogni processore aveva la sua *runqueue*. Questo, quindi, permetteva di non creare concorrenza per l'accesso ad una struttura dati condivisa e di non perdere eventuali dati salvati nella cache di un determinato processore.

5.2 Un algoritmo O(1)

L'algoritmo si poneva l'obbiettivo di risolvere il problema dei precedenti scheduler, ovvero il bisogno di iterare l'intera lista dei processi, con complessità O(n), per trovare un processo da porre in stato *running*.

Il funzionamento del nuovo algoritmo era semplice ed efficiente:

- Per trovare quale lista di processi nell'*active* array aveva almeno un processo in stato *runnable*, lo scheduler usava la funzione *sched_find_first_bit()*. La funzione era implementata diversamente a seconda dell'architettura del processore, ad esempio per i processori x86 la funzione faceva uso

dell'istruzione assembly *bfsl*, che trova in tempo costante il primo bit non nullo in una bitmap. Lo scheduler, dunque, poneva in stato *running* il processo in testa alla coda con priorità più elevata individuata, garantendo così una complessità O(1).

- Quando un processo terminava il suo tempo di esecuzione, veniva ricalcolato un nuovo *time-slice* per esso e successivamente spostato nell'*expired* array in fondo alla coda corrispondente alla sua priorità. Tuttavia, vi era un'eccezione: se un processo era sufficientemente interattivo, poteva essere re-inserito nell'*active* array se non erano presenti processi in *starvation* nell'*expired* array.
- Se la funzione *sched_find_first_bit()* non trovava nessun bit a "1", voleva dire che non erano presenti processi da porre in esecuzione. A questo punto lo scheduler scambiava i due puntatori dell'*active* array e dell'*expired* array con una complessità sempre O(1).

5.3 Calcolo della priorità

Ogni task ha una *nice* associata ad esso, in Linux 2.6 il valore era salvato all'interno del campo *static_prio* nel descrittore di processo. La priorità statica era calcolata come:

$$static\_prio = 100 + nice + 20$$

Il sistema usava delle macro utili, definite nel file /kernel/sched.c, per ricavare il valore della *nice* dalla priorità statica e viceversa **(si veda figura 5.3)**.

```
47  /*
48   * Convert user-nice values [ -20 ... 0 ... 19 ]
49   * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
50   * and back.
51   */
52  #define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)
53  #define PRIO_TO_NICE(prio)      ((prio) - MAX_RT_PRIO - 20)
54  #define TASK_NICE(p)            PRIO_TO_NICE((p)->static_prio)
```

Figura 5.3 conversione nice - static_prio, file sched.c

Inoltre, ogni processo possedeva una priorità dinamica salvata nel campo *prio* anch'essa memorizzata nel descrittore di processo. Il kernel basava le sue decisioni in base alla priorità dinamica. Per esempio, un task veniva assegnato ad una determinata coda in base alla sua priorità dinamica.

La funzione *effective_prio()* ritornava la priorità dinamica di un task in funzione della priorità statica e dello *sleep avarage*. In prima approssimazione il valore di ritorno era il valore della priorità statica, successivamente assegnava un bonus compreso nell'intervallo [-5,+5] basato sull'interattività del task, stimata tramite metodi euristici. Più in dettaglio, Linux teneva traccia del tempo che un processo trascorreva in stato di *sleeping* rispetto al tempo che trascorreva in stato *running*. Questo valore veniva memorizzato nel campo *sleep_avg*, nel descrittore di ogni processo e variava da zero a MAX_SLEEP_AVG, che di default corrispondeva a dieci millisecondi. Quando un processo passava da stato *sleeping* a *running* si incrementava *sleep_avg* del tempo trascorso in *sleeping*. Viceversa, quando un processo passava da stato *running* a *sleeping* si decrementava *sleep_avg* del tempo trascorso in esecuzione.

5.4 Calcolo del time-slice

Uno dei principali problemi dei precedenti scheduler era il ricalcolo del *time-slice*. Infatti, si doveva iterare sull'intera lista dei processi nel sistema, con una complessità O(n). Inoltre, poiché esisteva una sola *runqueue* durante la fase di ricalcolo, alcuni processori risultavano inutilizzati nelle architetture multi-core. L'approccio utilizzato dallo scheduler in Linux 2.6, ovvero di ricalcolare il *time-slice* alla fine dell'esecuzione di un processo, garantiva una complessità costante indipendente dal numero di task.

Dalla versione 2.6 di Linux il *quantum* non era più memorizzato nel campo *counter*, nel descrittore di processo, ma nel campo *time_slice*.

Alla nascita di un nuovo processo, il tempo di esecuzione rimasto al processo padre veniva diviso tra padre e figlio. Questo, per garantire equità prevenendo di creare un numero elevato di figli e quindi ricevere maggiore tempo di esecuzione. Per quanto riguardava i processi già esistenti nel sistema, il calcolo del nuovo *time-slice* veniva effettuato tramite la funzione *task_timeslice()*, definita nel file /kernel/sched.c, che

dato un task ritornava un nuovo *time-slice* per esso. In particolare, la funzione basava il calcolo in funzione della priorità statica:

$$time\_slice = \begin{cases} (140 - static\_prio) \times 20\,, & static\_prio < 120 \\ (140 - static\_prio) \times 5\ \ , & static\_prio \geq 120 \end{cases}$$

Mediante questo approccio, il kernel garantiva un *quantum* più lungo ai processi con priorità elevata.

5.5 Bilanciamento del carico

Come discusso, Linux 2.6 implementava in architetture multi-core differenti *runqueue*, una per ogni processore. Tuttavia, potevano verificarsi situazioni in cui il carico di lavoro non era ben distribuito tra i diversi processori. Per risolvere questa problematica, la scelta progettuale fu stata di usare un meccanismo di bilanciamento del carico.

La funzione che si occupava del bilanciamento del carico era *load_balance()* definita nel file /kernel/sched.c e poteva essere invocata tramite due meccanismi: dalla funzione *schedule()* tutte le volte che la *runqueue* del processore attuale era vuota o via timer se il sistema era in stato *idle*.

Ad ogni invocazione dello scheduler, esso doveva scegliere quale processo porre in esecuzione sulla CPU da una determinata *runqueue*. Con *load_balance()*, veniva prima di tutto trovata la *runqueue* con maggior carico e con almeno il 25% di processi in più rispetto alla *runqueue* "attualmente" analizzata dallo scheduler, tramite la funzione *find_busiest_queue()* definita anch'essa nel file /kernel/sched.c. Successivamente *load_balance()* decideva da quale *priority array*, nella *runqueue* trovata, migrare i processi. Infine, controllava la lista a più alta priorità cercando processi con determinate caratteristiche:

- Che non erano in stato *runnable* o *running*.
- A cui era permesso migrare da una CPU all'altra; per fare ciò il sistema usava una bitmask *cpu_allowed* contenuta nel descrittore di processo.

- Che non erano stati posti in esecuzione sulla CPU per il tempo massimo in cui i dati rimanevano memorizzati nella cache.

Se un task rispecchiava i criteri, veniva invocata la funzione *pull_task()*, definita anch'essa nel file /kernel/sched.c, per finalizzare la migrazione dal processore più carico a quello meno carico e cioè con *runqueue* vuota.

6 COMPLETELY FAIR SCHEDULER

Il CFS (*Completely Fair Scheduler*) fu rilasciato ufficialmente con la *release* 2.6.23 del 9/10/2007 di Linux per una migliore gestione dei processi SCHED_OTHER, ovvero dei processi interattivi.

Come suggerisce la documentazione ufficiale Linux, l'80% del design del CFS può essere sintetizzato in un'unica frase: il CFS si propone di modellare "una CPU ideale multi-processo" su un *hardware* reale. Ovvero, una CPU nella quale tutti i processi vengono eseguiti contemporaneamente. Se nel sistema sono attivi *n* processi, a ognuno di questi si dovrebbe assegnare una porzione di potenza di CPU pari a *1/n*. Per esempio: supponendo 2 task, essi dovrebbero essere eseguiti in parallelo sulla stessa CPU garantendo ad ognuno una "fetta" di potenza pari al 50%.

Ovviamente, modellare una CPU ideale multi-processo non è possibile nella realtà. Tuttavia, tramite alcuni dettagli implementativi che verranno illustrati in questo capitolo, lo scheduler riesce ad ottenere un comportamento simile.

Il CFS è attualmente lo scheduler usato in Linux, nel seguito di questo capitolo si farà riferimento alla versione 6.4.12. del 23/08/2023 la quale è l'ultima versione stabile disponibile.

6.1 Organizzazione del codice

Il codice dello scheduler ha un'organizzazione modulare e gerarchica. Più in dettaglio, il codice è diviso in diversi file, ognuno dei quali rappresenta l'implementazione di un algoritmo per una determinata classe di processi.

Il codice risiede nella *directory* /kernel/sched e alcuni importanti file contenuti in essa sono:

- core.c: in questo file è presente il codice "generale" dello scheduler (nelle prime versioni era denominato sched.c).

- fair.c: in questo file è presente l'implementazione del *Completely Fair Scheduler* per la gestione dei processi SCHED_OTHER.
- rt.c: in questo file è presente il codice per la gestione dei processi real-time, ovvero i processi SCHED_FIFO e SCHED_RR.
- deadline.c: in questo file è presente il codice per la gestione dei processi hard real-time, ovvero tutti i processi che hanno delle scadenze temporali da rispettare. La *policy* di questi processi è denominata SCHED_DEADLINE e fu introdotta nella versione 3.14 di Linux.
- idle.c: in questo file è presente il codice per la gestione dei processi *idle* denominati SCHED_IDLE.

A tratti l'implementazione è simile ad una gerarchia delle classi nella programmazione ad oggetti: il *core* dello scheduler viene "esteso" dai vari moduli e "implementato" differentemente per ogni classe di processo, all'occorrenza tramite polimorfismo viene richiesta una funzionalità implementata differentemente nei diversi moduli.

In altri termini, lo scheduler permette la coesistenza di più algoritmi di scheduling. I diversi moduli incapsulano i dettagli implementativi senza esporre il codice al *core* dello scheduler.

In figura 6.1, per maggiore chiarezza, una rappresentazione grafica della gerarchia.

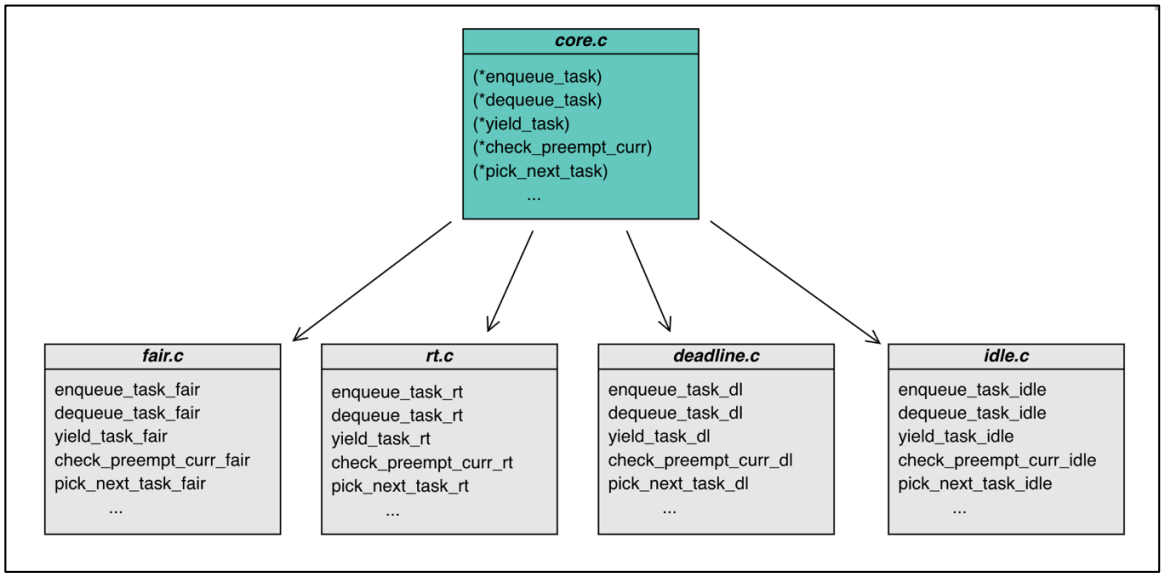

Figura 6.1 rappresentazione grafica della gerarchia nello scheduler

6.2 Scheduling tra classi

Lo scheduler fa uso del concetto di scheduling tra classi: ad ogni sua invocazione itera tra le varie classi di processo cercando il processo a priorità più elevata appartenente alla classe a priorità più alta.

Le classi di scheduling sono implementate nei diversi file citati nel paragrafo 6.1. Ogni file contiene la definizione di una *struct sched_class* nella quale si hanno i puntatori a funzione per la gestione dei task appartenenti ad una classe, alcune delle quali sono:

- *pick_next_task(...)*: seleziona il task da porre in esecuzione dalla *runqueue*.
- *enqueue_task(...)*: inserisce un task nella *runqueue*.
- *dequeue_task(...)*: rimuove un task dalla *runqueue*.
- *check_preempt_curr(...)*: controlla se un nuovo processo *runnable* deve attuare *preemption* nei confronti del processo *running*.

Le classi disponibili in Linux 6.4 sono quattro e sono in stretta correlazione con la *policy* di un processo **(si veda tabella 6.1).**

Classe	**Policy**
dl_sched_class	SCHED_DEADLINE
rt_sched_class	SCHED_RR, SCHED_FIFO
fair_sched_class	SCHED_OTHER
idle_sched_class	SCHED_IDLE

Tabella 6.1 correlazione tra classe e policy di un processo, in ordine prioritario decrescente.

In figura 6.2 un frammento di codice dal file core.c, in particolare dalla funzione *schedule(),* in cui avviene l'iterazione tra le classi.

```
5988            for_each_class(class) {
5989                    p = class->pick_next_task(rq);
5990                    if (p)
5991                            return p;
5992            }
```

Figura 6.2 iterazione tra le classi di scheduling

Nel seguito di questo capitolo si farà riferimento al codice contenuto nel file fair.c, ovvero all'implementazione del *Completely Fair Scheduler*.

6.3 Tempo di esecuzione e virtual runtime

Il *Completely Fair Scheduler* non fa uso del concetto di *time-slice* introdotto nei capitoli precedenti, bensì ad ogni task assegna una percentuale di tempo di CPU.

Inoltre, lo scheduler fa uso di due importanti parametri:

- *Target latency*: è il lasso di tempo in cui un processo dovrebbe ottenere la CPU almeno una volta.
- *Minimum granularity*: il tempo di esecuzione minimo per i task.

La percentuale di tempo di CPU assegnata al j-esimo task viene calcolata come segue:

$$CPU\_time = target\_latency \times \frac{weight_j}{\sum_{i=0}^{n\_tasks} weight_i}$$

Dove il peso (*weight*) è un valore che dipende dalla priorità del processo, calcolato sperimentalmente **(si veda figura 6.3)**.

```
11459   const int sched_prio_to_weight[40] = {
11460    /* -20 */     88761,     71755,     56483,     46273,     36291,
11461    /* -15 */     29154,     23254,     18705,     14949,     11916,
11462    /* -10 */      9548,      7620,      6100,      4904,      3906,
11463    /*  -5 */      3121,      2501,      1991,      1586,      1277,
11464    /*   0 */      1024,       820,       655,       526,       423,
11465    /*   5 */       335,       272,       215,       172,       137,
11466    /*  10 */       110,        87,        70,        56,        45,
11467    /*  15 */        36,        29,        23,        18,        15,
11468   };
```

Figura 6.3 relazione tra peso e priorità di un processo

Quindi, il CFS, a differenza dei precedenti scheduler, non assegna un "*time-slice"* indipendente e uguale ad ogni processo ma calcola il tempo necessario all'esecuzione del processo in funzione del numero di processi nel sistema. Considerando due processi con la stessa priorità e una *target latency* di 20ms, lo scheduler allocherebbe un tempo di CPU pari a 10ms ad ognuno, con cinque processi ognuno verrebbe eseguito per 4ms e a mano a mano che il numero di processi tende a infinito la percentuale di CPU assegnata ad ogni processo tenderebbe a zero, per questo viene in aiuto la *minimum granularity*. Inoltre, per valori molto piccoli della *target latency* si tende ad avere maggiore interattività ma si rischia di avere anche un aumento dell'*overhead* dovuto ai numerosi *context switch*. Nel CFS, la *target latency* cresce all'aumentare del numero di processi nel sistema.

Infine, ad ogni processo viene assegnato un *virtual runtime*: il tempo effettivamente utilizzato da un processo rispetto a una CPU ideale multi-processo. Il *virtual runtime* di un processo viene incrementato del tempo di esecuzione effettivo moltiplicato per un fattore di decadimento che dipende dalla priorità del processo, tutte le volte che rilascia la CPU. Più in dettaglio, per i processi a priorità elevata il *virtual runtime* cresce lentamente e per i processi a priorità minore cresce velocemente.

All'occorrenza il task ad essere posto in esecuzione sulla CPU è il processo con il *virtual runtime* minore, ovvero il processo che è stato posto in esecuzione di meno.

Il CFS è "*fair*" perché conferisce a ogni processo una divisione equa della CPU in relazione agli altri processi.

6.4 Red-black tree

Una delle più interessanti caratteristiche del CFS è la *runqueue*, infatti, invece che usare una *doubly linked list*, come negli scheduler precedenti, utilizza un *red-black tree*, un particolare albero di ricerca binario bilanciato. Il nome deriva dal fatto che ciascun nodo può essere o rosso o nero. Oltre ai requisiti di un albero binario di ricerca, un *red-black tree* soddisfa le seguenti proprietà:

- Ogni nodo ha colore rosso o nero.

- La radice è di colore nero.
- Ogni foglia è nera e contiene un elemento NULL.
- Entrambi i figli di ciascun nodo rosso sono neri.
- Ogni cammino da un nodo a una foglia nel suo sottoalbero contiene lo stesso numero di nodi neri.

Questi vincoli rafforzano una proprietà critica degli alberi rosso-neri: il cammino più lungo dal nodo root a una foglia è al massimo lungo il doppio del cammino più breve. Ne risulta dunque un albero fortemente bilanciato dove le operazioni di inserimento, cancellazione e ricerca vengono effettuate nel caso peggiore con complessità $O(log\ n)$, dove n è il numero di processi nell'albero.

La chiave di ogni nodo è la differenza tra il *virtual runtime* di un processo e il *virtual runtime* minore nella r*unqueue*. Quindi, il task che deve essere posto in esecuzione sulla CPU si trova sempre nel nodo più a sinistra dell'albero **(si veda figura 6.4)**.

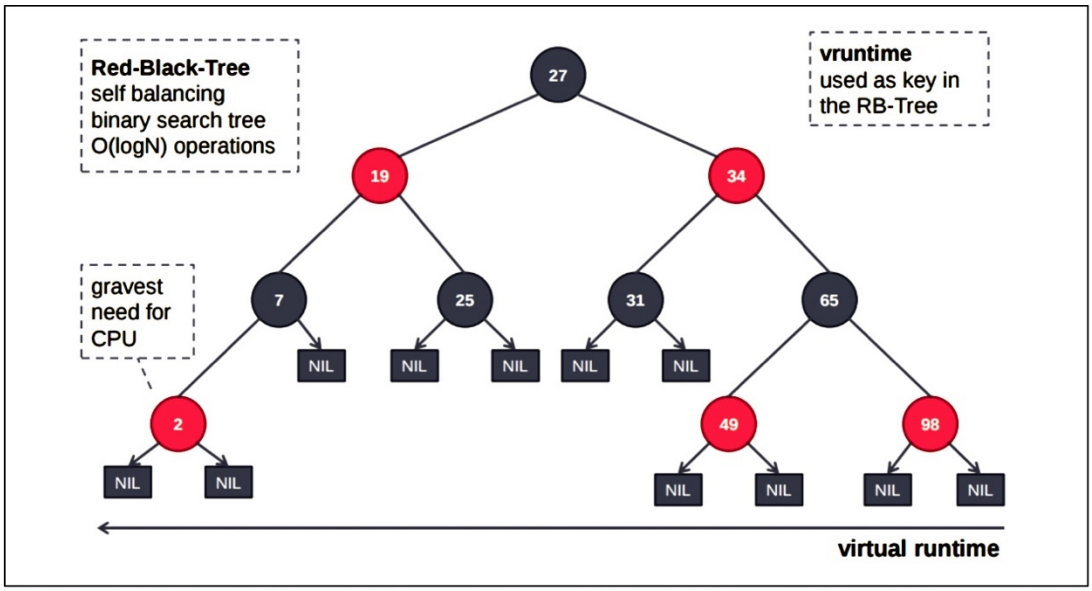

Figura 6.4 red-black tree

Siccome all'occorrenza bisognerebbe scorrere tutto l'albero per selezionare il processo da porre in esecuzione, lo scheduler, per questioni di efficienza, tiene traccia del nodo più a sinistra in una variabile, la quale viene aggiornata ad ogni cambio di contesto. Così facendo la scelta del prossimo processo da eseguire sulla CPU avviene, in realtà, con complessità O(1).

6.5 Scheduling entities

Tutti i task del sistema sono rappresentati da un descrittore di processo. Tuttavia, il CFS lavora con entità più generali: le *entity*.

Le entità sono definite come *struct sched_entity* nel file /linux/include/sched.h e al suo interno si trovano i campi utili per la gestione del task da parte dello scheduler. Ad esempio, il campo *vruntime*, il quale contiene il *virtual runtime* dell'entità. Il riferimento alla struttura si trova nel descrittore di processo come *struct sched_entity se*.

Un singolo task è sempre un'entità, viceversa un'entità può rappresentare anche gruppi di task. Questo torna particolarmente utile allo scheduler, ad esempio, se più utenti sono attivi sulla stessa macchina. Supponendo 25 processi nel sistema, di cui 20 appartenenti all'utente A e 5 all'utente B, poiché ha l'obbiettivo di essere equo, il CFS assegnerebbe la medesima percentuale di CPU tra i processi. Tuttavia, così facendo, l'utente A otterrebbe una maggiore potenza di CPU rispetto all'utente B.

Con l'utilizzo delle entità, lo scheduler riesce a garantire inizialmente l'equità tra i gruppi e successivamente tra i singoli task. Facendo riferimento all'esempio precedente, questo viene realizzato allocando il 50% della capacità di CPU a ciascun utente, che verrà poi distribuita in parti uguali tra i singoli task.

CONCLUSIONE

Il presente elaborato ha analizzato l'evoluzione dello scheduler in Linux. In particolare, sono stati trattati il primo scheduler Linux, lo scheduler nella versione 2.4, lo scheduler O(1) e, infine, il *Completely fair scheduler* che da quando è stato introdotto nella versione 2.6.23 (in data 9/10/2007), è lo scheduler attualmente anche nella versione corrente che è la 6.4.12 (al 23/08/2023).

L'analisi dei diversi algoritmi di scheduling presentati in questo elaborato illustra come l'evoluzione tecnologica e le crescenti esigenze computazionali abbiano spinto allo sviluppo di politiche e meccanismi di gestione dei processi sempre più sofisticati.

L'evoluzione da algoritmi di facile intuizione, come il primo scheduler Linux, a complessi strumenti di scheduling, come il *Completely Fair Scheduler*, ha dimostrato l'impegno costante nella ricerca di soluzioni che massimizzino l'utilizzo delle risorse e minimizzino il tempo di attesa per i processi, garantendo al contempo una distribuzione equa delle risorse tra i diversi task in esecuzione nei sistemi Linux.

Bibliografia e sitografia

Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*, Third Edition. O'Reilly, 2005.

Paul Krzyzanowski. *Process Scheduling: Who gets to run next?* February 18, 2015.

Pramode C.E, Gopakumar C.E. *The Linux Kernel 0.01 Commentary*, 2003.

Robert Love. *Linux Kernel Development, Third Edition*. RR Donnelley, Crawfordsville, Indiana: Addison-Wesley, 2010.

PABLA, Chandandeep Singh. *Completely fair scheduler*. Linux Journal, 2009, 2009.184: 4.

https://it.wikipedia.org/wiki/RB-Albero.

https://www.science.smith.edu/~nhowe/262/oldlabs/sched.html.

http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch04lev1sec2.html.

https://elixir.bootlin.com/linux/latest/source