ALVISE RIGO

SCHEDULING IN LINUX: APPLICAZIONE E BENEFICI IN PROCESSI INTERATTIVI

SCHEDULING IN LINUX: APPLICAZIONE E BENEFICI IN PROCESSI INTERATTIVI

ALVISE RIGO

analisi teorica e sperimentazione su BRC4

Dicember 2009

ALVISE RIGO: *Scheduling in Linux: applicazione e benefici in processi interattivi,* analisi teorica e sperimentazione su BRC4, © Dicember 2009

ABSTRACT

L'utilizzo di internet è in continua espansione, in ogni settore, da quello casalingo a quello professionale. Questo dirompente sviluppo e questa crescente necessità di apparati atti a fornire accessi ad Internet e a reti locali obbliga le aziende, che producono questi apparati, a essere assolutamente competitive e a ricercare intensivamente soluzioni per migliorare l'esperienza dell'utente finale. Ci si è prefissi questo scopo: migliorare una tecnologia che già esiste, che in determinati contesti mostra le proprie lacune. I routers in questione, animati da un chip `Broadcom`, sottoposti ad un elevato stress di rete (attraverso la porta uplink) non riescono a garantire un'usabilità sufficiente del servizio voce, realizzato con tecnologia *voip*. La "voce" non è l'unico elemento che ne soffre ma tutta la macchina ne risente, in particolar modo trasferimenti tramite le interfacce USB. Opportune modifiche allo scheduler del kernel Linux probabilmente avrebbero potuto perfezionare la situazione: questa considerazione è stata l'inizio di questo studio...

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

LISTINGS

ACRONYMS

GPL General Public License

FSF Free Software Foundation

ISR Interrupt Service Routine

IRQ Interrupt Request

PC Program Counter

PID Process ID

SDK Software Development Kit

CPE Customer Premise Equipment

DUT Device Under Testing

BRAS Broadband Remote Access Server

DSLAM Digital Subscriber Line Access Multiplexer

VOIP Voice over IP

SIP Session Initiation Protocol

Parte I

IL KERNEL E LO SCHEDULER LINUX

1 LINUX, IL KERNEL

Questo percorso di studi ha richiesto dapprima l'immediata analisi del kernel Linux; tale elemento software si è rivelato fondamentale fino alla fine della ricerca.

1.1 UN PÒ DI STORIA

Il kernel Linux, cuore di tutte le distribuzioni Linux, è open source quindi libero: tutti i suoi sorgenti possono essere facilmente scaricati, modificati e ricompilati nei termini della General Public License (GPL). Il motivo per cui un pezzo di storia informatica come il kernel Linux sia open source è da ricercarsi in un periodo precedente alle sue stesse origini, ed è attribuibile al genio di *Richard Stallman*, istitutore della Free Software Foundation (FSF), nonchè colui che ha steso la licenza GPL. Secondo gli ideali di Richard Stallman l'utenza informatica non aveva bisogno di grandi potenze industriali come AT&T o IBM per godere di un sistema operativo funzionante; lui solo dimostrò che questo era possibile, costruendo il primo sistema operativo *libero*. Certamente non era cosi completo e performante, ma la collaborazione della comunità e l'aiuto reciproco avrebbero fatto il resto.

Il progetto era molto più ampio del singolo sistema operativo; infatti non aveva solo finalità tecniche, ma sociali, politiche ed etiche. Stallman voleva che come lui, tutti i programmatori scrivessero codice senza fini di lucro, ovviamente nelle situazioni che lo permettevano. Fu cosi che in quegli anni nacque la FSF, fondazione per promuovere software libero, distribuzione, copia, modifica e comprensione del software, accomapagnata dalla nota licenza GPL, acronimo di *General Public License*. Questa nuova visione dello sviluppo software ha avuto un grande successo tanto da aver trovato diversi "mecenati" che tutt'oggi la finanziano.

E' in questo contesto che Linus Torvalds, nel 1991, scrisse Linux (ispirato dal sistema operativo *commerciale* UNIX) e lo rilasciò sotto GPL; unendo quindi il lavoro di Stallman e di Torvalds nacque *GNU/LINUX*, oppure, più comunemente, *LINUX*. Da ora in poi con Linux mi riferirò al sistema operativo completo, comprendente kernel e GNU; un'approssimazione che, al giorno d'oggi, è molto comune. I sistemi operativi Linux sono molteplici, tutti con le loro peculiarità, pregi e difetti; si parla infatti di distribuzioni Linux, come Ubuntu, Debian, RedHat, tutte accumunate da una stessa origine e dallo stesso nucleo: GNU, che è cresciuto grazie alla collaborazione di centinaia di sviluppatori. Ha avuto lo stesso destino il kernel che anno dopo anno è cresciuto, tanto che è difficile quantificare la sua grandezzza.

1.2 COS'È UN KERNEL?

In questa sezione sarà analizzato il fondamentale ruolo del kernel e le sue principali funzionalità.
Qualsiasi elaboratore per essere fruibile ha bisogno di un sistema operativo, nonchè di insieme di programmi che permettano di sfruttare le sue potenzialità di calcolo. Il programma più importante che esegue in un sistema operativo è il kernel. Questo viene caricato nella memoria RAM all'avvio dell'elaboratore e contiene le istruzioni per eseguire varie operazioni necessarie al sistema per funzionare correttamente, in primis quelle relative al boot del sistema operativo. Esso è anche l'elemento software più vicino (in termini funzionali) all'hardware che costituisce il sistema e, come tale, interagisce con la componentistica hardware; è per questo motivo che molto del codice di Linux è stato scritto direttamente in linguaggio assembly. La sua seconda funzionalità è quella di provvedere ad un contesto software per le applicazioni (programmi) del sistema, in poche parole ha il compito di garantire la corretta esecuzione dei processi, più di uno contemporaneamente. Questo non è un compito facile, anzi, è molto complesso perchè la contemporanea esecuzione di processi apre delle problematiche che devono essere opportunamente gestite. I processi, durante la loro esecuzione, hanno spesso necessità di accedere a periferiche; essendo il sistema operativo Linux multiprocesso, deve esserci un'attività che regoli la cessione delle risorse ai processi. Ad esempio, se il processo A ha appena ottenuto una risorsa, un processo B non può accedere alla stessa risorsa, a meno che questa non permetta accessi multipli da parte di più processi. Un'altra funzione esempio che un sistema operativo deve garantire è quella che permette ad un processo, che attende un input da una periferica, di essere sospeso a favore di un altro processo che magari ha appena acquisito un dato da un'altra periferica ed aspetta di elaborarlo: sarebbe inutile sospendere il processore solo perchè il processo che esegue è in attesa. Il kernel dunque deve fornire un meccanismo di interruzione che consenta il salvataggio del contatore (Program Counter (PC)) del programma interrotto e trasferisca il controllo ad un'altra locazione in memoria.

Si origina una problematica: con che criterio vengono scelti i processi da eseguire? Questo è il compito del DISPATCHER e dello SCHEDULER; il primo passa effettivamente il controllo della CPU ai processi *ready* (pronti) scelti dal secondo. Mentre il dispatcher ha un ruolo piuttosto meccanico, lo scheduler è un vero e prorpio programma che gira costantemente e che, secondo un preciso algoritmo, sceglie i processi da "passare" al dispatcher.
Sarà questo il principale argomento trattato in questo elaborato: lo scheduler del kernel Linux. La trattazione, a causa della vastità e complessità dell'argomento, manterrà un certo grado di astrazione e generalità, presentando solo i concetti fondamentali (lo scheduler, per esempio, conta più di 10000 righe di codice).

1.3 PUNTI DI FORZA

Verranno ora presi in considerazione gli elementi che hanno reso Linux così popolare.

- KERNEL MONOLITICO: è un unico grande programma, composto

da varie parti logicamente differenti; il kernel monolitico offre un'interfaccia virtuale di alto livello sui componenti hardware del sistema, cosi da poter fornire un set di primitive per realizzare le funzioni prime di un sistema operativo. Il più grande svantaggio di questa architettura è l'impossibilità di utilizzare una periferica senza aver compilato il necessario modulo. Un'eccezione è OSX, che, seppur sia basato su Unix, adotta un kernel con architettura ibrida.

- KERNEL THREADING: alcuni kernel Linux sono organizzati attraverso dei thread, che sono dei contesti di esecuzione di alcune funzioni che necessitano di eseguire periodicamente. Questi thread sono schedulati indipendentemente, come gli altri processi. Un cambio di contesto di questi task è tuttavia molto più rapido di quello previsto tra thread convenzionali, questo, al livello computazionale, è un netto vantaggio.

- SUPPORTO PER LE APPLICAZIONI MULTITHREADING: il kernel Linux permette la realizzazione di programmi multithreading, ossia di programmi sviluppati su più thread che sono schedulati indipendentemente, ma che attingono ad una memoria comune.

- PREEMPTIVE KERNEL: il kernel 2.6 introduce questa grande novità, che ha portato ad un sensibile incremento prestazionale. Definire cosa significa "preemptive" non è semplice ed è fuorviante: lo stesso concetto è utilizzato per indicare concetti leggermente differenti. In qualsiasi kernel un processo che esegue in kernel mode può rilasciare volontariamente il controllo della CPU, perchè, ad esempio, deve aspettare una risorsa ancora non disponibile. La vera novità che ha introdotto il kernel preemptive è la possibilità di eseguire preemption a dei flussi di esecuzioni in kernel mode, che siano gestione di interrupt sincroni (eccezioni lanciate da programmi) o asincroni. Un esempio sarà chiarificatore: un processo A sta eseguendo in kernel mode per gestire un interrupt; nel contempo un altro processo diventa runnable poichè risvegliato dalla Interrupt Service Routine (ISR) di un'altra Interrupt Request (IRQ): se il kernel è di tipo "preempitivo" lo scheduler può sospendere il task A a favore del task B.

- SUPPORTO MULTIPROCESSORE: Linux 2.6 supporta perfettamente più di una CPU, rendendo possibile l'esecuzione su sistemi multiprocessore, come i grandi mainframe e supercomputer. È significativo il fatto che al giorno d'oggi, quasi tutti i 500 più veloci supercomputer (vedi *www.top500.org*) montano varianti di Linux.

2 PANORAMICA DEL KERNEL LINUX

In questo capitolo verranno presentati alcuni concetti fondamentali utili per la comprensione delle funzioni del kernel.

2.1 MODI DI ESECUZIONE

Ogni sistema operativo moderno non permette a programmi eseguiti dall'utente di interagire con risorse hardware; in generale, si può affermare che tutte le specifiche e anche dettagli di basso livello sono oscurati ai programmi di alto livello, cioè quelli eseguiti dagli utenti. Il motivo di questo impedimento è ovvio: non si può garantire l'interazione con l'hardware da parte dei processi in un contesto non controllato, ma ci deve essere un supervisore (il kernel) che valuti la cessione delle risorse ai programmi onde evitare errori di sistema. Qualora un processo volesse accedere ad una risorsa (ad esempio una periferica hardware), lo fa presente esplicitamente al kernel attraverso una richiesta; sarà il kernel a valutare quest'ultima e fare in modo che il processo trovi la risorsa voluta operativa. Per formalizzare questi concetti, il sistema operativo prevede due *execution modes*: una non privilegiata per gli *user programs* (`User Mode`) ed una privilegiata riservata al kernel (`Kernel Mode`).

Le due modalità di esecuzione sono totalmente trasparenti all'utente, il passaggio da/a `Kernel Mode` viene fatto, per esempio, ad ogni aggiornamento del puntatore del mouse

Quando un programma è eseguito in modalità `User Mode` non può accedere direttamente alle strutture e ai programmi del kernel. Quando invece un programma esegue in `Kernel Mode` la restrizione sopra citata non vale, ed il programma non ha limitazioni di esecuzione. Ci sono delle particolari funzioni che permettono ad un programma di passare da `User Mode` a `Kernel Mode`, queste funzioni sono chiamate *System Calls*. Oltre ai processi convenzionali che sottostanno alle leggi che sono state descritte precedentemente, esistono dei processi particolari che hanno una modalità di esecuzione privilegiata, i *kernel threads*. Essi hanno le seguenti caratteristiche:

- eseguono sempre in `kernel mode` nello spazio di memoria (*address space*) del kernel;
- non interagiscono con l'utente, e per questo non necessitano di un'interfaccia di controllo;
- di solito sono creati all'avvio del sistema e si interrompono alla chiusura dello stesso.

Si dice che un processo sta eseguendo in `kernel mode` se, attraverso una chiamata di sistema (*system call*), ha richiesto l'esecuzioni di una specifica funzione che neccessita l'intervento del kernel per eseguire delle operazioni critiche (*routine*). Nello specifico possono essere distinti quattro casi in cui si passa l'esecuzione alla routine del kernel:

- un processo invoca una *system call*;
- la CPU che sta eseguendo un determinato processo segnala una *exception*, che è una situazione inusuale in cui si è trovato il processo, magari a causa di un'istruzione errata. In questi casi, il

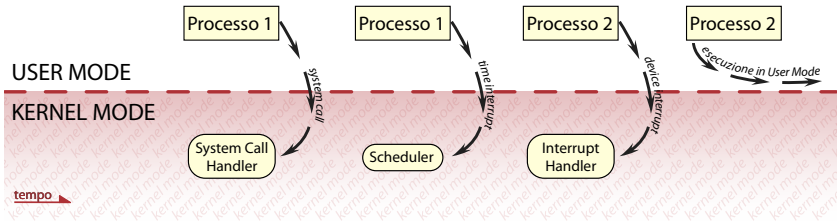

Figura 1: esempio di esecuzione di due processi

controllo del processore deve essere passato al kernel che gestirà la situazione secondo la routine specifica;

- un dispositivo hardware della macchina solleva un *interrupt* atto a segnalare alla CPU che è avvenuto un particolare evento (per esempio la disponibilità di dati provenienti da una periferica a seguito di una operazione di I/O). Il kernel dovrà dunque gestire l'interrupt con il relativo *interrupt handler*;
- un thread del kernel è eseguito, poichè questo viene eseguito in `Kernel Mode`, il corrispettivo programma va considerato parte del kernel.

In figura 1 è raffigurato il principale ruolo svolto dallo *scheduler*; come si evince, a seguito dell'interruzione di un timer, questo programma viene invocato per gestire tale interruzione e, se necessario, per sospendere il processo che sta attualmente eseguendo a favore di uno pronto per eseguire.

2.2 SCHEDULER

Lo `scheduler` è un componete del kernel che, quando un processo abbandona volutamente l'uso della CPU oppure viene sospeso, decide quale processo deve eseguire.

Da ora in poi chiamerò il processo che esegue processo running.

Nel caso di un sistema che prevede *preemption* della CPU, il kernel può togliere la CPU al processo *running* a favore di un altro processo, in questo caso si parla di `preemptive scheduling`. Uno scheduling `cooperativo` invece non permette al sistema operativo di togliere l'uso della CPU al processo *running* il quale lascia la CPU solo se si interrompe autonomamente oppure quando termina di eseguire. Per essere precisi, anche in un kernel coopertativo è possibile togliere l'uso della CPU ad un processo, ma non quando questo è in kernel mode; ciò favorisce la risposta del sistema ai processi interattivi. Si può notare che la definizione di scheduling *preemptive* trova corrispondenza con la definizione di kernel *preemptive* enunciata nel capitolo precedente, è proprio la tipologia dello scheduler che differenzia i kernels dotati di preemption da quelli non dotati di questa funzionalità.
I sistemi operativi moderni eseguono concorrenzialmente varie tipologie di processi: alcuni richiedono un gran uso di CPU, altri sono computazionalmente più leggeri ma devono essere eseguiti in tempi brevissimi; vedremo come, fin dalle prime versioni del kernel, lo scheduler "scheduli" processi differenti con metodologie differenti. La natura che differenzia i processi verrà presentata nel capitolo successivo.

3 PROCESSI

Prima di proseguire in un'analisi approfondita, per la trattazione successiva è fondamentale definire cos'è un processo, argomento che è fondamentale per le trattazioni successive.

3.1 PROCESSI

Il processo identifica un'astrazione e, purtroppo, non risulta di facile interpretazione. Nella letteratura informatica con processo ci si riferisce sia all'istanza di un programma in esecuzione sia al contesto in cui un programma esegue.
E' importante distinguere i programmi dai processi: molti processi possono eseguire lo stesso programma contemporaneamente; uno stesso processo, invece, può eseguire molti programmi sequenzialmente. Più semplicemente: un programma, una volta avviato, risulta al sistema operativo come un processo in quanto, come detto precedentemente, è l'istanza di un programma che esegue; il sistema operativo vede processi eseguire, non programmi. A sua volta un programma che esegue (e quindi un processo) può avviare altri programmi di cui necessita il programma "madre". Dal punto di vista del kernel, la finalità del processo non è altro rappresentare un'entità per cui sono state allocate delle risorse di sistema, le quali possono essere: CPU time, memoria centrale e memoria secondaria. Una volta creato, un processo riceve una copia logica dell' *address space* del processo genitore ed esegue lo stesso codice di quest'ultimo. Nonostante ci sia questa condivisione, le loro strutture dati sono differenti: ognuno possiede la sua, indipendente da quella del padre/figlio.
Attualmente Linux usa i cosiddetti *lightweight processes* che garantiscono un supporto migliore per le applicazioni multithreading. Due processi *lightweight* possono condividere le stesse risorse, come gli *address space*, i file aperti e cosi via; quando uno dei due processi modifica una risorsa condivisa, il cambiamento a sua volta è condiviso e risulta immediatamente al secondo processo. E' opportuno che i *lightweight process* si sincronizzino a vicenda qualora dovessero accedere a delle risorse condivise, onde evitare l'accesso simultaneo agli stessi files.

3.1.1 *thread*

Strettamente associato a processo è il concetto di THREAD (abbreviazione di *thread of execution*). Un thread è la componente atomica di un processo, nonchè quella sezione di codice che viene eseguita senza interruzioni, parallelamente ed indipendentemente ad altri thread dello stesso processo. Va sottolineato che un processo contiene come minimo un thread: il processo stesso. Le vecchie versioni di Linux non supportavano le applicazioni con più thread; dagli occhi del kernel, un applicazione *multithreaded* risultava come un unico processo e questo a scapito di sicurezza e prestazioni. In questi casi l'applicazione era creata, gestita e schedulata in *user mode* da una libreria apposita.
Un esempio può chiarire il motivo per cui un'implementazione di que-

sto tipo era davvero poco efficiente: supponiamo che un'applicazione abbia due thread principali: uno attende un input da tastiera e lo scrive su un file, l'altro elabora i dati presenti nel file. Sarebbe opportuno che, mentre il primo thread attende l'input dell'utente, l'atro continui a lavorare, cosi da sfruttare i tempi morti dovuti dalla relativa lentezza delle operazioni di input. Nel caso in cui il programma fosse un unico processo, il primo thread non può evocare una chiamata di sistema bloccante, in quanto bloccherebbe anche il secondo thread; per evitare questo, il thread dovrà essere progettato con sofisticati artefizi per evitare che il processo si blocchi, complicando lo sviluppo software. Un processore con un singolo core può eseguire solo un flusso di codice, quindi un solo processo (o meglio, un solo thread); più cresce il numero di core logici, maggiore è la possibilità del sistema operativo di eseguire più porzioni di codice contemporaneamente. In generale un calcolatore ha poche CPU, per cui i processi che effettivamente eseguono contemporaneamente sono altrettanto pochi.

3.1.1.1

Un modo semplice per realizzare il *multithreading* in Linux è quello di associare ad ogni *lightweight process* un *thread*, cosi i threads di un processo condividono aree di memoria ed il kernel può schedularli indipendentemente, sarà cosi possibile che un un thread dello stesso programma sia *running* ed uno invece stia "dormendo".

3.2 PROCESS DESCRIPTOR

Per schedulare al meglio i processi, il kernel deve essere in grado di poter accedere a delle informazioni particolari riguardanti i processi. E' per questo che è stata prevista una struttura dati di tipo `task_struct` chiamata process descriptor. Ogni processo ha la sua struttura, la quale contiene puntatori a delle altre strutture dati che contengono a loro volta informazioni sul processo.

Una fondamentale informazione che risiede in questa struttura dati è lo stato del processo, lo `state`. Questo consiste in un vettore di *flags* ognuno dei quali descrive uno stato possibile del processo; nelle ultime versioni del kernel questi stati sono mutualmente esclusivi. I possibili stati sono:

- `TASK_RUNNING`: il processo sta eseguendo (*running*) oppure è pronto per essere eseguito;
- `TASK_INTERRUPTIBLE`: il processo è sospeso (*sleeping*); un processo in questo stato può riprendere l'eseguibilità solo attraverso il verificarsi di una opportuna condizione: la risorsa per cui sta aspettando è nuovamente disponibile oppure viene eseguito un signal da parte di un altro processo;
- `TASK_UNINTERRUPTIBLE`: raramente usato, è come lo stato precedente, con la differenza che un segnale non lo "risveglia";
- `TASK_STOPPED`: l'esecuzione del processo si è fermata dopo che questo ha ricevuto un segnale del tipo SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU;
- `TASK_TRACED`: l'esecuzione del processo si è fermata per l'intervento di un debugger;

- `EXIT_ZOMBIE`: l'esecuzione del processo è terminata, ma il processo padre deve ancora verificare l'effettiva terminazione del processo tramite un segnale della famiglia `wait()`; tali informazioni possono servire al processo padre ed il kernel non può rimuoverle dalle sue aree di memoria;
- `EXIT_DEAD`: questo è lo stato finale dove il processo sta per essere rimosso dal sistema.

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

3.3 IDENTIFICATORI DI PROCESSO

L'identificazione di un processo è fondamentale sia per lo scheduler, sia per l'intero sistema operativo; di fatto, se non ci fosse un modo comodo e facilmente interpretabile per identificare univocamente i processi, l'efficacia di molti programmi in Linux e dello stesso kernel verrebbe a mancare. Ci sono vari attributi che possono identificare univocamente un processo, quale, ad esempio, l'indirizzo in memoria del Process Descriptor. Questa scelta sebbene corretta, non sarebbe ottimale: utilizzare numeri a 32 bit in esadecimale infatti non è ciò che un utente vorrebbe vedere come identificativo dei processi che ha avviato e che magari vuole terminare. Lo stesso si potrebbe dire per un programmatore: complicherebbe solo le cose (già esse stesse complesse). La soluzione è stata semplice: associare ad ogni processo un numero progressivo partendo dall'identificatore "1" e successivamente ad ogni figlio (o nuovo processo) associare il numero dell'ultimo processo creato incrementato di uno. Tale numero è denominato Process ID (PID).
Il PID è tenuto in memoria della struttura dati del processo, nel campo denominato `pid`.

4 SCHEDULING DEI PROCESSI

4.1 L'ALGORITMO DI SCHEDULING

L'algoritmo di scheduling utilizzato nelle prime versioni di Linux era molto semplice ed intuitivo; ogni volta che avveniva un switch di processo, il kernel controllava la lista dei processi running, aggiornava le loro priorità e selezionava, eseguendolo, il processo migliore. Il concetto su cui si basava era piuttosto elementare, ma senza dubbio corretto; tuttavia un'implementazione di quel tipo mostrava una pecca fondamentale: le esigue risorse computazionali. Lo scheduler infatti, ogni volta che veniva chiamato, porgeva delle prestazioni O(n), quindi, più task stavano eseguendo nel sistema, più tempo era richiesto dallo scheduler per effettuare la scelta del processo migliore. É chiaro quindi che un'operazione O(n) (eseguita ogni cent di secondo o anche meno) ha un impatto significativo sulle prestazioni del sistema. Se inoltre consideriamo le limitate risorse hardware disponibili fino a pochi anni fa, una soluzione così realizzata era ancor più svantaggiosa.

Lo scheduler introdotto dal kernel 2.6 è più sofisticato ed esegue con ottime prestazioni anche a fronte di un numero elevato di processi; il suo costo in termini prestazionali infatti è costante, quindi O(1). È stato sviluppato per sfruttare una possibile pluralità di CPU; ogni core infatti dispone della sua lista di processi *runnable*. É anche stata migliorata la distinzione tra processi *batch* e processi *interattivi*.
Per qualsiasi tipo di implementazione, lo scheduler adempie sempre al suo scopo e non è ammesso che esso non abbia un processo da scegliere come "migliore"; questo obiettivo è stato raggiunto introducendo un processo chiamato *swapper*, che per antonomasia ha PID 0; è il processo che esegue quando la CPU non ha altri processi pronti per essere eseguiti. Ogni CPU, pertanto, ha il suo *swapper* process, in quanto ogni CPU può trovarsi in "idle".

Ogni processo in Linux è schedulato secondo una di queste classi di schedulazione:

- `SCHED_FIFO`: è una classe per processi real-time con alta priorità, con gestione First-in, First-out. Se lo scheduler assegna la CPU ad un processo appartenente a questa classe, viene lascia il rispettivo descrittore di processo nella runqueue list. Se non ci sono altri processi real-time con priorità maggiore, il task esegue fino a quando non si sospende, anche se sono disponibili processi con la stessa priorità;

- `SCHED_RR`: è una classe per processi real-time con gestione Round Robin. Quando viene assegnata la CPU ad uno di questi processi, lo scheduler mette il rispettivo descrittore di processo in coda alla runqueue list così da assicurare una assegnazione della CPU equa ai processi di questa classe. Esempio: se un processo con priorità 10 viene schedulato con questa classe, viene messo in una coda circolare (la runqueue list della priorità 10) contenente tutti

i processi sempre di priorità 10 e schedulati in round robin; ogni priorità ha dunque la sua coda circolare;

- `SCHED_NORMAL`: è una classe per processi convenzionali *time-shared*.

Quindi la distinzione tra processi real-time e convenzionali è marcata, tanto da rendere necessarie classi di scheduling profondamente differenti. A seconda del caso lo scheduler si comporta diversamente.

4.2 SCHEDULING DI PROCESSI CONVENZIONALI

Ogni processo convenzionale ha la propria *priorità statica*, un parametro che usa lo scheduler per trattare ogni processo adeguatamente, nel rispetto degli altri processi nel sistema. Il kernel riserva 40 valori per rappresentare questa priorità, la quale va da 100 (priorità maggiore) a 139 (priorità minore): all'aumentare del valore, la priorità del processo diminuisce.

Un nuovo processo eredita sempre la priorità statica del padre; tuttavia un utente può configurare la priorità di un processo modificando il `nice` del processo attraverso le chiamate a sistema `nice()` e `setpriority()`.

4.2.1 *Quanto temporale "base"*

Dalla priorità statica lo scheduler calcola il *base time quantum*, nonchè la durata del quanto temporale di esecuzione di un processo. Nella realizzazione del kernel, si è pensato che ogni processo dovesse eseguire all'interno di quanti temporali di lunghezza variabile a seconda delle caratteristiche del processo. Terminato il quanto temporale, il processo viene fermato a favore dei processi concorrenti che possono eseguire; a meno di particolari condizioni, il processo fermato dovrà aspettare il prossimo turno, il quale sarà caratterizzato da un quanto temporale generalmente diverso. Il calcolo del nuovo quanto viene effettuato con la seguente equazione:

$$\text{quanto temporale}_{[\text{ms}]} = \begin{cases} (140 - \text{priorità statica}) \times 20 & \text{se priorità statica} < 120 \\ (140 - \text{priorità statica}) \times 5 & \text{se priorità statica} \geqslant 120 \end{cases} \tag{4.1}$$

È evidente che maggiore è la priorità statica (e quindi minore è il valore), maggiore è il quanto temporale: processi con maggiore priorità ottengono un quanto temporale maggiore. Nella tabella 1 sono riportati i valori che si riferiscono al nice minore (massima priorità), quello di default e quello maggiore; inoltre sono compresi anche due valori intermedi.

Descrizione	Priorità statica	Valore Nice	Quanto temporale	Interactive Delta	soglia sleep time
priorità statica più alta	100	-20	800 ms	-3	299 ms
priorità statica alta	110	-10	600 ms	-1	499 ms
priorità statica di default	120	0	100 ms	+2	799 ms
priorità statica bassa	130	10	50 ms	+4	999 ms
priorità statica più bassa	139	20	5 ms	+6	1199 ms

Tabella 1: Valori tipici per un processo convenzionale

4.2.2 *Priorità dinamica e sleep time medio*

Ogni processo è caratterizzato non solo da una priorità statica, ma anche da una *priorità dinamica*, che, come la statica, va da un massimo di 100 (priorità massima) ad un minimo di 139 (priorità minima). La priorità dinamica è quell'attributo considerato dallo scheduler nel momento in cui deve trovare il processo più adatto da eseguire. C'è una formula empirica che detta il valore della priorità dinamica:

$$\texttt{priorità dinamica} = \max(100, \min(\texttt{priorità statica} - \texttt{bonus} + 5, 139)) \tag{4.2}$$

Il *bonus* è una variabile compresa tra 0 e 10; confrontando priorità statica e dinamica, un valore di bonus inferiore a 5 rappresenta una penalità che diminuirà la priorità del task; un valore maggiore, invece, la aumenterà. Il valore che lo scheduler attribuisce a questa variabile dipende dal passato del processo e più precisamente dal valore assunto da *average sleep time*, letteralmente "tempo medio di sonno"; la tabella 2 presenta i possibili valori di bonus.

AVERAGE SLEEP TIME può essere considerato indicativamente il tempo medio, in nanosecondi, trascorso dal processo durante lo stato di *sleep*, questa definizione però non è del tutto esatta.

Un processo è nello stato sleep quando si è sospeso oppure è stato sospeso; in questo stato i processi non sono nella runqueue ma in una differente struttura dati.

Ad esempio, dormire nello stato `TASK_INTERRUPTIBLE` contribuisce in modo diverso rispetto al dormire nello stato `TASK_UNINTERRUPTIBLE`. In aggiunta, quando il processo sta effettivamente eseguendo, average sleep time diminuisce. Nella seguente tabella sono mostrati esempi di valori significativi con i corrispettivi bonus.

Il valore *average time sleep* è utilizzato dallo scheduler per valutare se il processo è da considerarsi di *batch* o *interattivo*; è interattivo se soddisfa la seguente formula:

$$\texttt{priorità dinamica} \leqslant 3 \times \texttt{priorità statica} \diagup 4 + 28 \tag{4.3}$$

che è equivalente alla formula:

$$\texttt{bonus} - 5 \geqslant \texttt{priorità statica} \diagup 4 - 28 \tag{4.4}$$

Average sleep time	**Bonus**	**Granularità**
Maggiore uguale a 0 e minore di 100 ms	0	5120
Maggiore uguale a 100 ms e minore di 200 ms	1	2560
Maggiore uguale a 200 ms e minore di 300 ms	2	1280
Maggiore uguale a 300 ms e minore di 400 ms	3	640
Maggiore uguale a 400 ms e minore di 500 ms	4	320
Maggiore uguale a 500 ms e minore di 600 ms	5	160
Maggiore uguale a 600 ms e minore di 700 ms	6	80
Maggiore uguale a 700 ms e minore di 800 ms	7	40
Maggiore uguale a 800 ms e minore di 900 ms	8	20
Maggiore uguale a 900 ms e minore di 1000 ms	9	10
1 secondo	10	10

Tabella 2: average sleep time, bonus e granularità di time slice

Ricapitolando, la priorità dinamica di un processo, in quanto *dinamica*, varia nel tempo; per un processo (che nasce con una sua priorità statica), l'unica variabile che interviene nella 4.2 è il bonus, che dipende direttamente (e linearmente) dal valore di sleep time che ha il processo. Entrando nel dettaglio, analizzando la funzione 4.2 (rappresentata in 3 esempi nel grafico 2), si nota come il bonus influisca nella quantificazione della priorità dinamica: più lo sleep time aumenta (e quindi più aumenta il bonus), più la priorità dinamica scende e quindi dal punto di vista di un processo, più questo "dorme". Oltre a questo avrà anche la possibilità di essere considerato interattivo, meritandosi un valore minore di priorità dinamica. Quindi un processo interattivo è quello che aspetta di più (magari un input di qualche periferica). Questa considerazione è in accordo con la condizione 4.3. Il grafico mostra che il processo è interattivo quando ha priorità dinamimca inferiore ad una certa soglia, calcolabile con la priorità statica del processo stesso.

L'espressione `priorità statica / 4 - 28` è chiamata *interactive delta*; tipici valori di questo parametro sono presenti nella tabella 1; si nota che è più facile per un processo ad altà priorità divenire interattivo rispetto ad uno di bassa priorità.

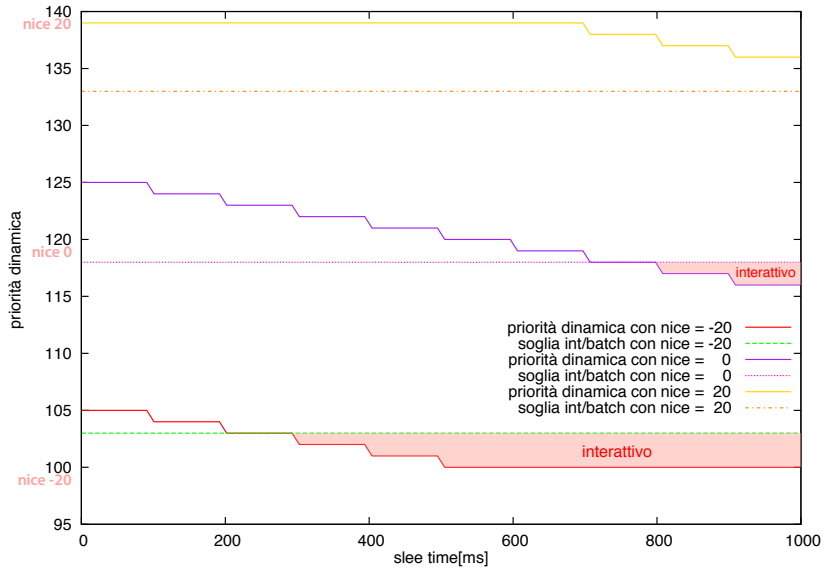

Figura 2: soglia interattivo / batch

Ad esempio: un processo che ha la massima priorità dinamica (100) è considerato interattivo se il suo bonus è maggiore di 2 con una media di sleep time pari a 200 ms. Significativo è il fatto che un processo con priorità statica uguale a 139 non può essere mai considerato un processo interattivo, mentre un processo con priorità statica di default (120) diventa interattivo se dorme per più di 700 ms.

L'immagine 3 può aiutare a distinguere i processi interattivi da quelli di batch. Sono presenti due funzioni: quella di colore rosso rappresenta la priorità dinamica al variare di sleep time e priorità statica, quella blu tratteggiata delimita la soglia secondo la quale lo scheduler considera il processi di batch oppure interattivi. Sono del primo tipo se la priorità dinamica è sopra alla soglia e, invece, del secondo tipo se sta sotto.

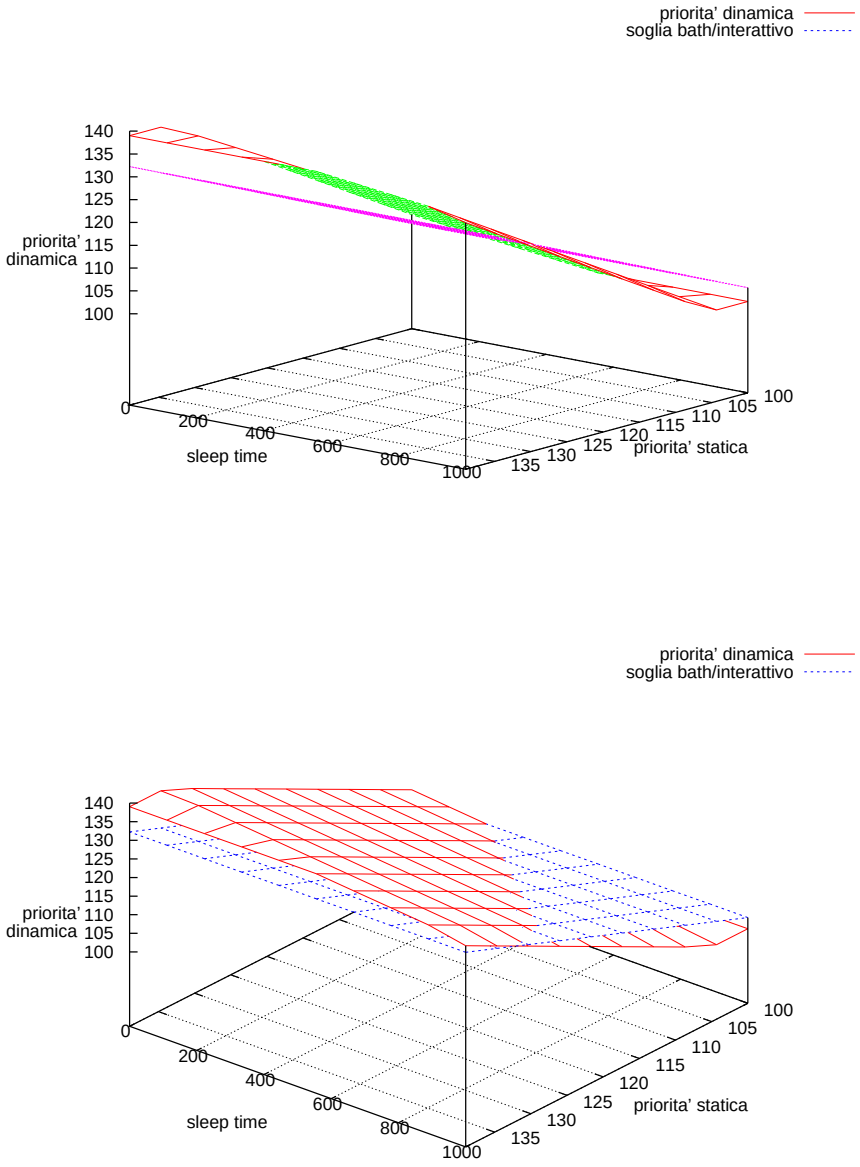

Figura 3: distinzione processi interattivi e di batch

4.2.3 *Processi attivi e scaduti*

Con questa implementazione delle politiche dello scheduler, non sempre capita che un processo acquisisca una priorità di molto inferiore o superiore a quella con cui è stato creato: la natura di un processo rimane piuttosto invariata.

Se cosi fosse e se effettivamente lo scheduler scegliesse sempre il processo migliore, si introdurrebbe una grande problematica che impedirebbe l'esecuzione (da parte dello scheduler) dei processi a bassa priorità, una volta che quelli con alta priorità avessero finito il loro quanto temporale.
Ovviamente questa disfunzione (detta anche *starvation* dei processi) deve essere evitata. Il kernel risolve tale problema utilizzando due liste di processi running che individuano due tipi di processi:

- PROCESSI ATTIVI (*active processes*): questi processi non hanno ancora completato il loro quanto temporale, quindi, finchè non lo terminano, hanno il diritto di eseguire;
- PROCESSI SCADUTI (*expired processes*): questi processi hanno appena finito il loro quanto temporale e finchè non si esaurisco i processi attivi, questi continueranno a non eseguire.

In realtà le cose sono un pò più complicate: in generale un processo di batch se ha finito il suo quanto temporale viene sempre aggiunto alla lista di processi scaduti, ma questo non è vero per un processo interattivo. In questo caso infatti lo scheduler ricalcola il suo quanto temporale e lo lascia nella lista dei processi attivi ammesso che non sia vera una di queste condizioni:

- il primo dei processi (quello più prioritario) della lista expired ha aspettato oltre una determinata soglia;
- un processo scaduto ha priorità *statica* maggiore del processo interattivo: in questo caso lo scheduler è programmato per aggiungerlo alla lista dei processi scaduti.

4.3 SCHEDULING DEI PROCESSI REAL-TIME

Ogni processo real-time ha associata una priorità real-time che va da 1 (priorità più bassa) a 99 (priorità più alta).

A differenza dei processi convenzionali, quelli real-time sono sempre attivi. Se più processi real-time hanno la stessa prorità, lo scheduler sceglie quello che risulta primo della lista di priorità a cui appartiene (di una data CPU).
Un processo real-time viene rimpiazzato da un altro quando si avvera una delle seguenti condizioni:

- il processo subisce prelazione da un altro processo con maggiore priorità real-time. È ovvio che un processo real-time avrà sempre priorità dinamica maggiore di un processo convenzionale;
- il processo esegue un'operazione bloccante ed è messo a "dormire" (nello stato `TASK_INTERRUPTIBLE` o `TASK_INTERRUPTIBLE`);
- il processo è stato fermato (dunque o nello stato `TASK_STOPPED` o `TASK_TRACED`), oppure se "ucciso" (tramite comando `kill`, quindi in stato `EXIT_ZOMBIE` o `EXIT_DEAD`);

- il processo rilascia volontariamente la CPU lanciando la chiamata di sistema `sched_yield()`.
- il processo è in Round Robin real-time (`SCHED_RR`) e, una volta ultimato il suo quanto temporale, subisce prelazione

Una precisazione deve essere fatta per le chiamate di sistema `nice()` e `setpriority()`, quando queste sono applicate ad un processo real-time in Round Robin, non cambiano la sua priorità real-time, ma la durata del suo quanto temporale. Infatti la durata del quanto temporale non dipende dalla priorità real-time del processo, ma dalla sua priorità statica, secondo la formula 4.1.

4.4 STRUTTURE DATI USATE DALLO SCHEDULER

Ricapitolando ciò che è stato detto nei capitoli precedenti, ogni processo è riferito dalla lista dei processi (*processes list*), mentre la *runqueue list* indica tutti i processi che sono in uno stato `TASK_RUNNING`, eccetto il processo *swapper*.

4.4.1 *La struttura dati runqueue*

Questa è la struttura dati più importante utilizzata dallo scheduler Linux. Ogni CPU presente nel sistema ha la propria runqueue. La macro `this_rq()` tiene l'indirizzo della runqueue della CPU "locale", mentre la macro `cpu_rq(n)` fornisce l'indirizzo della runqueue della CPU con indirizzo *n*.

La struttura `rq` è l'implementazione della *runqueue list*, nello specifico, nel kernel Linux 2.6 la sua definizione è presente nel file "*linux.2.6.XX/kernel/sched.c*" (con "linux.2.6.XX" si intende la cartella contenente il sorgente del kernel alla versione 2.6.XX). Come il resto del codice del kernel Linux, la struttura `rq` è scritta in linguaggio C.

Variabili di notevole importanza sono:

- `unsigned long nr_running`: tiene il conto dei processi running presenti nella ruqueue;
- `struct mm_struct *prev_mm`: usato durante uno *switch* per indirizzare il descrittore di memoria del processo che sta per essere rimpiazzato;
- `struct task_struct *curr`: puntatore al descrittore del processo che sta eseguendo;
- `struct task_struct *idle`: puntatore al desctittore del processo *swapper* per la CPU corrente;
- `struct prio_array *active, *expired`: puntatori alle liste di processi attivi e a quella dei processi "scaduti";
- `struct prio_array arrays[2]`: array contenente i set di processi attivi e scaduti;
- `struct sched_domain *sd`: punta al dominio di schedulazione della CPU corrente;
- `struct task_struct *migration_thread`: puntatore al descrittore del kernel thread "*migration*";

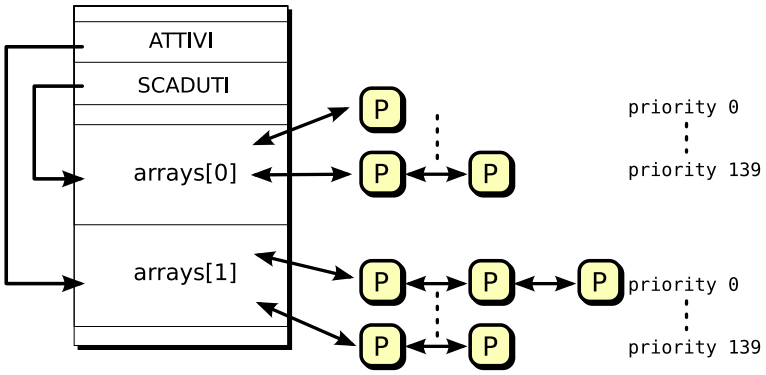

Figura 4: Struttura runqueue e le due liste di processi eseguibili

- `struct list_head migration_queue`: lista dei processi che devono essere rimossi dalla runqueue list.

Il campo più importante della struttura runqueue è quello legato alla lista di processi eseguibili (runqueue). Ogni processo *runnable* appartiene ad una, ed una sola, runqueue; finchè il processo rimarrà in quella lista, solo la CPU corrispondente a quella runqueue potrà eseguirlo. La runqueue prevede inoltre due puntatori a delle strutture di tipo `prio_array` che detengono la lista dei processi attivi e scaduti. Esse includono 140 nodi iniziali (*heads*) di doubly linked list (una per priorità), una mappa di priorità (*priority bitmap*) ed un contatore che conta i processi della struttura. Periodicamente accade che il puntato dei due puntatori viene scambiato; la lista di processi attivi diviene quindi quella dei processi scaduti e viceversa.

4.4.2 *Descrittore di processo*

Ogni processo, nel proprio descrittore, include vari campi finalizzati al processo di schedulazione. I più importanti, utili nella seguente trattazione, sono:

- `thread_info->flags`: tiene il flag `TIF_NEED_RESCHED`, che è settato se il task deve essere rischedulato;
- `thread_info->cpu`: numero della CPU della runqueue a cui appartiene il processo corrente;
- `state`: lo stato corrente del processo;
- `prio`: priorità dinamica del processo;
- `static_prio`: priorità statica del processo;
- `array`: puntatore alla runqueue (`prio_array_t`) che contiene il processo;
- `sleep_avg`: sleep time medio del processo;
- `timestamp`: tempo dell'ultima interazione con il processo, di inserzione nella lista o di contest switch;
- `activated`: condizione di risveglio del processo;
- `policy`: classe di schedulazione del processo;

- `time_slice`: ticks di esecuzione rimanenti al time slice del processo;
- `first_time_slice`: flag posto a 1 se il processo è nel suo primo time slice;
- `rt_priority`: priorità real time del processo.

Quando un processo viene creato con il comando `sched_fork`, il numero di ticks rimanenti del processo sono divisi a metà, una per il padre ed una per il figlio. Questo meccanismo elimina la possibilità di guadagnare un potenziale tempo infinito di esecuzione; un esempio banale può essere quello di un'interfaccia grafica ed i processi da essa dipendenti (o figli). Ogni volta che si avvia un programma, questo comincia l'esecuzione con un quanto temporale completo ed è evidente come alla lunga questo possa portare ad un abuso di risorse da parte di questi processi.

4.5 FUNZIONI USATE DALLO SCHEDULER

Le più importanti funzioni utilizzate dallo scheduler sono:

- `scheduler_tick()`: utilizzata per aggiornare lo `time_slice` del processo *current*;
- `try_to_wake_up()`: risveglia un processo dormiente;
- `recalc_task_prio()`: aggiorna la priorità del processo;
- `schedule()`: seleziona il processo che deve eseguire;
- `load_balance()`: tiene le runqueues dei vari processori bilanciate.

Il file "sched.c" è presente in ogni versione del kernel Linux; ad ogni rilascio di una nuova versione è possibile vedere le modifiche apportate nel file di changelog.

Per capire a fondo il funzionamento dello scheduler (ricordando che non si tratta dello scheduler attualmente implementato) è stato necessario analizzare riga per riga il codice presente, prevalentemente nel file sched.c. Per non complicare troppo l'argomentazione, non è stato possibile riportare tutte le considerazioni e le nozioni utili che interessano questa parte, per cui nelle prossime righe saranno trattati solo i passaggi peculiari, riportando, dove significativo, il codice relativo.

4.5.1 *Funzione `scheduler_tick()`*

Il nome di questa funzione suggerisce il suo ruolo di fondamentale importanza nella schedulazione in Linux, la quale è eseguita ad ogni tick del sistema, scandendo, in un certo modo, l'esecuzione delle funzioni dello scheduler concorrentemente all'esecuzione dei processi. Si ricorda che lo scheduler è, di fatto, un processo.

4.5.1.1 *circuiti di timer*

É opportuno aprire una parentesi: qualsiasi calcolatore, nelle sue componenti hardware, presenta vari dispositivi che definiscono il passare del tempo attraverso dei "*tick*". Questi dispositivi (*timer circuits*) possono essere di varia natura; il più famoso è certamente il *Real Time Clock* (RTC) che, sollevando interruzioni hardware ad un frequenza stabilita,

permette di tenere traccia del trascorrere del tempo e di fornire data ed ora. Un altro timer di fondamentale importanza è il "Time Stamp Counter" (TSC), il quale riceve un segnale di clock da un oscillatore esterno ed aggiorna il numero di oscillazioni in una variabile, accessibile con un'istruzione assembly. Per esempio, se il clock oscilla a 1MHz, avremo un tick ogni microsecondo. Questa struttura permette al sistema operativo di eseguire calcoli temporali ad alta precisione. In un computer casalingo con un kernel Linux standard, la questione dei timer non è di fondamentale importanza: lo scheduler, o meglio, i processi, non devono rispettare delle "deadline" oltre le quali vengono compromesse le funzionalità del sistema. Nei sistemi Real-Time invece, avere una scansione del tempo precisa è di fondamentale importanza, per questo motivo, sono disponibili versioni "adattate" del kernel che rivisitano sensibilmente il concetto di scheduling e di gestione delle interruzioni hardware.

4.5.1.2 *Scheduler, scansione temporale*

Il sistema operativo, ad ogni tick, esegue varie funzioni al fine di garantire funzionalità e stabilità al sistema. Queste funzioni sono eseguite a seguito della gestione dell'interrupt sollevato dal timer del sistema; come anticipato in precedenza, una di queste è `scheduler_tick()`, chiamata da `update_process_times()`. I passi fondamentali della sua funzione sono:

il valore del flag `TIF_NEED_RESCHED` viene controllato ad ogni ritorno da interrupt o exceptions attraverso istruzioni assembly eseguite in lock, soprattutto al ritorno dagli interrupt del timer

A. Controllare se il processo *current* è lo *swapper* della CPU locale, in tal caso:
   a) se la runqueue locale contiene un altro processo oltre allo *swapper*, viene settato il flag `TIF_NEED_RESCHED` per forzare la rischedulazione del processo;
   b) salta al punto F in quanto allo swapper non viene aggiornato il *time slice*.

B. Controllare se `current->array` punta alla lista dei processi attivi, in caso negativo significa che il processo ha terminato il suo quanto temporale e deve ancora essere rimpiazzato. Viene dunque settato il flag `TIF_NEED_RESCHED` e salta al punto F.

C. Acquisire il lock della lista con `this_rq()->lock`.

D. Decrementare il time slice e controlla se questo è ultimato.

E. Rilasciare lo spin lock della coda.

F. Eseguire `rebalance_tick()` che assicura che i processi siano equamente distribuiti tra le varie CPU del sistema.

4.5.1.3 *Aggiornamento per processi real-time*

Per i processi real-time le istruzioni di aggiornamento sono differenti, in particolare se il processo *current* è un processo real-time FIFO, lo scheduler non fa nulla. In questo caso il processo non può subire preemption da un processo con priorità uguale o minore ed è per questo motivo che non avrebbe senso tenere il suo contatore aggiornato. Differente è la sorte dei processi real-time RR (gestiti con politica Round Robin). Per questi la funzione è programmata per controllare se il quanto temporale è scaduto dopo averlo decrementato; in caso positivo (`!--p->time_slice == 1`) verrà ricalcolato il time slice del processo (secondo formula 4.1). Verrà azzerata la condizione di esecuzione nel

primo time slice e verrà anche posto il flag `TIF_NEED_RESCHED` ad 1. Il campo `first_time_slice` è proprio di ogni task e inizializzato ad 1 nel momento in cui il processo viene creato tramire chiamata `fork()`. Prima di uscire, la funzione porrà in coda ai processi dello stesso - anello il task current e rilascerà il lock, cosi da forzare l'invocazione dello scheduler quando l'esecuzione rientrerà dall'interrupt del timer. Il codice responsabile di quanto appena descritto è riportato qui di seguito:

```
if (rt_task(p)) {
                if ((p->policy == SCHED_RR) && !--p->time_slice) {
                        p->time_slice = task_timeslice(p);
                        p->first_time_slice = 0;
                        set_tsk_need_resched(p);
                        requeue_task(p, rq->active);
                }
                goto out_unlock;
        }
```

4.5.1.4 *Aggiornamento per processi convenzionali*

Se invece *current* è un processo convenzionale, `scheduler_tick` esegue come segue:

A. Decrementa il time slice del task;

B. Qualora fosse scaduto il quanto temporale:

a) rimuove il processo dalla coda degli attivi (active runqueue);

b) setta il task come da rischedulare (flag `TIF_NEED _RESCHED`);

c) aggiorna la priorità dinamica del processo secondo 4.2;

d) ricalcola il quanto temporale del processo:

```
p->time_slice = task_timeslice(p);
p->first_time_slice = 0;
```

e) se il valore di `expired_timestamp` è zero (quindi se la coda di processi scaduti è vuota), lo aggiorna con il tick corrente:

```
if (!rq->expired_timestamp)
        rq->expired_timestamp = jiffies;
```

f) inserisce il processo sulla coda degli *expired* o degli *active*, a seconda di particolari condizioni. Se il processo non è interattivo o c'è *starvation* tra gli expired, il processo viene accodato nel set degli *expired*, altrimenti in quello degli attivi. In altre parole, il processo, qualora ci fosse starvation viene *sempre* messo tra gli *expired*, altrimenti, dipendentemente dalla condizione che sia o meno interattivo, viene posto rispettivamente tra gli *active* o tra gli *expired*:

```
if (!TASK_INTERACTIVE(p) || expired_starving(rq)) {
        enqueue_task(p, rq->expired);
        if (p->static_prio < rq->best_expired_prio)
                rq->best_expired_prio = p->static_prio;
        } else
                enqueue_task(p, rq->active);
```

La condizione di starvation non è facile da definire sinteticamente; con *starvation* si vuole indicare quella particolare condizione di un processo che, a causa di task con priorità più alta, non riesce ad eseguire. Si tratta di casi poco frequenti poichè è raro (in ambito desktop o embedded) che ci siano tanti processi `RUNNABLE` da poter provocare starvation. Tuttavia il kernel Linux, utilizzato in

una vastissima gamma di sistemi, deve saper gestire ogni tipo di scenario. Si riporta qui di seguito il metodo `expired_starving()`, responsabile della valutazione della condizione di starvation:

```
static inline int expired_starving(struct rq *rq)
{
        if(rq->curr->static_prio > rq->best_expired_prio)
                return 1;
        if(!STARVATION_LIMIT || !rq->expired_timestamp)
                return 0;
        if(jiffies - rq->expired_timestamp >
                        STARVATION_LIMIT * rq->nr_running
                            )
                return 1;
        return 0;
}
```

Nel codice, `STARVATION_LIMIT` è pari al massimo tempo medio di *sleep* tra i processi; la condizione `!rq->expired_timestamp` verifica che non ci siano processi tra gli scaduti, mentre l'ultima condizione restituisce 1 se il tempo passato dal primo dei task scaduti è maggiore di una data quantità atta a garantire equità tra i processi.

c. Se il task a questo punto non è stato inserito in qualche coda, significa che non ha ancora ultimato il suo quanto temporale e pertanto deve continuare la sua esecuzione se il proprio quanto temporale non è troppo lungo. La macro `TIMESLICE_GRANULARITY` è stata definita per evitare che un task si trovi nelle condizioni di monopolizzare la CPU:

```
if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
        p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
        (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
        (p->array == rq->active))
{
        requeue_task(p, rq->active);
        set_tsk_need_resched(p);
}
```

4.5.2 *Funzione `try_to_wake_up()`*

Un altro aspetto che deve ancora essere considerato è il seguente: "come è possibile rendere *running* un processo che si è sospeso?" Fino ad ora si è supposto che i processi, anche se non eseguono sempre, siano comunque in uno stato `TASK_RUNNING` e per questo in una runqueue. I processi però, come abbiamo visto, possono attendere la realizzazione di un certo evento, e dunque si sospendono in uno stato `TASK_INTERRUPTIBLE` o in uno stato `TASK_UNINTERRUPTIBLE`. Qualora un processo in uno di questi stati volesse risvegliarsi, la routine del kernel adibita a questa funzione dovrà utilizzare il metodo `try_to_wake_up()`. Il metodo riceve i seguenti parametri:

- il descrittore del processo da risvegliare;
- una maschera dello stato del processo che può essere risvegliato;
- un flag che proibisce al processo risvegliato di *prelazionare* il processo correntemente *running*.

Senza scendere troppo nel dettaglio, la funzione esegue le seguenti operazioni:

A. Controlla se il task passsato come parametro è congruo con la maschera specificata, quindi termina.

B. Se il puntatore `p->array` non è `NULL`, il processo appartiene già ad una runqueue; in questo caso si salta alla fine del metodo.

C. Seguono varie istruzioni per gestire correttamente il carico computazionale tra le varie CPU che nel nostro caso non sono utili.

D. Se il task è in uno stato `TASK_UNINTERRUPTIBLE`, il contatore `nr_uninterruptible` verrà decrementato di un'unità.

E. Viene attivata la funzione `activate_task`, che esegue le seguenti funzioni:

   a) Invoca la funzione per ottenere il *timestamp* corrente.

   b) Invoca il metodo `recalc_task_prio()` per aggiornare la priorità del task.

   c) Aggiorna la variabile `p->activated` in accordo con la tabella 3.

   d) Aggiorna il valore della variabile `p->timestamp` con il valore calcolato al passo "a".

   e) Inserisce il task (il descrittore del task) nel set degli attivi:

```
1 enqueue_task(p,rq->active);
2 rq->nr_running++;
```

F. Valorizza il campo `p->state` con il valore `TASK_RUNNING`.

G. Ritorna "1" o "0" a seconda che il risveglio sia andato o no a buon fine.

4.5.3 *Funzione `recalc_task_prio()`*

`recalc_task_prio` aggiorna lo sleep time medio del task e la priorità dinamica del processo, riceve come parametro il puntatore al descrittore del processo da aggiornare e il timestamp corrente. Questa funzione è di fondamentale importanza in quanto sviluppa le proprietà che lo scheduler si prepone:

- schedulazione dei processi coerentemente con le priorità dei task;
- perfezionamento della priorità del processo schedulato secondo l'indole di questo.

La funzione esegue le seguenti operazioni:

A. Assegna alla variabile locale `sleep_time` il valore (in nanosecondi) di sospensione del task; se questo valore è maggionre di 1 secondo, viene assegnato il valore di 1 secondo.

B. Controlla che il processo non sia un thread del kernel, che non si sia risvegliato da uno stato `TASK_UNINTERRUPTIBLE` (ossia `p->activated uguale a -1`) e che non stesse dormendo oltre una determinata soglia. Se queste tre condizioni sono soddisfatte, la funzione valorizza il campo `p->sleep_avg` a 900 ticks e quindi salta al punto finale. Questo piccolo espediente da la possibilità ai task che si erano sospesi in attesta di operazioni di I/O (o per ragioni simili) di avere un'adeguata priorità statica per essere eseguiti velocemente, senza causare starvation per gli altri processi.

C. Calcola il `bonus` del task, e lo aggiorna.

D. Se il processo è in uno stato `TASK_UNINTERRUPTIBLE` e non è un thread del kernel, vanno eseguite le seguenti operazioni:

   a) controlla se `p->sleep_avg` è maggiore o uguale alla soglia prestabilita (vedi tabella 1); se è così, azzera il valore dello sleep time medio e salta al punto E;

   b) se `sleep_avg + p->sleep_avg` è maggiore uguale alla soglia, pone il campo `p->sleep_avg` al valore della soglia e a o `sleep_avg`.

E. Aggiunge `sleep_time` al valore di sleep time medio del task (`p->sleep_avg`).

F. Controlla se `p->sleep_avg` eccede i 1000 ticks, in tal caso fa un "cutoff" a 1000 ticks.

G. Aggiorna la priorità dinamica del processo con:

```
p->prio = effective_prio(p);
```

4.5.4 *La funzione `schedule()`*

La funzione `schedule()` è quella che *de facto* implementa lo scheduler e che ha il compito di scegliere il processo più adatto all'esecuzione. Essa è molto importante ed è invocata in due modi, quello diretto e quello pigro, dall' inglese *lazy*.

4.5.4.1 *Invocazione diretta*

Questo tipo di esecuzione si presenta quando un task deve sospendersi per mancanza di una risorsa; in questo caso la routine del kernel (a seguito di una chiamata di sistema) dovrà:

A. Inserire *current* nella coda di attesa della risorsa associata che gli spetta.
B. Cambiare lo stato di *current* in `TASK_INTERRUTIBLE` o `TASK_UNINTERRUTIBLE`, a seconda del processo.
C. Invocare la funzione `schedule()`.
D. Controllare se la risorsa è disponibile, se non lo è tornare al punto B.
E. Quando la risorsa è disponibile, risvegliare il processo rimuovendolo dalla coda di attesa.

Quando il processo è sospeso in una coda, la routine del kernel prenderà il posto di esecuzione del processo; quando eseguirà potrà così controllare ripetutamente che la risorsa sia disponibile. Se cosi non fosse rilascerà la CPU ad un altro processo invocando la funzione `schedule()`.

Si inserisce in questo tipo di invocazione il caso di alcuni driver che, nella gestione degli interrupt, eseguono lunghe iterazioni. Questi task controllano ad ogni iterazione se il valore del flag di current è impostato su `TIF_NEED_RESCHED` (il valore potrebbe essere stato modificato da `scheduler_tick()`); in caso affermativo chiamano la funzione `schedule()` per rilasciare volontariamente l'uso della CPU.

4.5.4.2 *Invocazione "pigra"*

Lo scheduler può essere invocato "pigramente" attivando il flag `TIF_NEED_RESCHED`; esso forzerà l'invocazione dello scheduler al successivo ritorno in *User Mode* (ritornando dalla gestione di un interrupt viene sempre controllato il valore del flag). Ci possono essere vari casi in cui si effettua una chiamata della funzione in questo modo:

A. *current* ha ultimato il suo quanto temporale, `scheduler_tick()` quindi setterà il flag `TIF_NEED_RESCHED`.
B. Un task, che è stato risvegliato dalla routine del kernel, ha priorità maggiore del processo *current*, in questo caso l'azione è compiuta dalla funzione `try_to_wake_up()`.
C. Quado è invocata la chiamata di sistema `sched_setscheduler()`.

4.5.4.3 *Azioni svolte dalla funzione `schedule()` prima del contest switch*

Lo scopo della funzione `schedule()` è piuttosto lineare e semplice; per come è stato realizzato tutto il meccanismo di scheduling, le istruzioni eseguite da questo metodo hanno un costo computazionale costante, indipendente dal numero di processi che stanno eseguendo nel computer: questo va a tutto vantaggio della reattività del sistema.

La funzione chiama `prev` e `next` rispettivamente il task che deve essere sostituito e quello che eseguirà successivamete. Nelle prime istruzioni `schedule()` deve calcolare il tempo che il processo sottrae al proprio time slice; la durata di esecuzione viene associata alla variabile `run_time` che, se dovesse superarare il secondo, subisce un "cut-off" a 1000ms. Successivamente `run_time` verrà pesato secondo il valore di *average sleep time* del task.

```
run_time /= (CURRENT_BONUS(prev) ? : 1);
```

Successivamente dovrà essere verificato se il processo `prev` sia un processo che sta per essere terminato.

```
if (prev->flags & PF_DEAD)
        prev->state = EXIT_DEAD;
```

Come è chiaro dal codice listato, è sufficiente fare un `and` logico tra il valore del flag ed una maschera per verificare se è effettivamente un task "uscente". Ora `schedule()` esamina lo stato di `prev`; se le seguenti condizioni saranno *entrambe* verificate, il task verrà rimosso dalla runqueue. Il processo quindi:

- non deve eseguire; ovvero:
  `prev->state != TASK_RUNNING`
- non non ha subito preemption dalla modalità kernel:
  `!(preempt_count() & PREEMPT_ACTIVE)`

Inoltre, se il task è in uno *sleeping* interrompibile da un segnale e ha un segnale pendente, viene settato *running*; altrimenti, se è in uno stato non interrompibile, viene semplicemente incrementata la variabile `nr_uninterruptible` della runqueue. Seguono delle istruzioni atte alla verifica dell'esistenza di processi *runnable*; qualora non ci fossero, next viene settato sullo *swapper*. Se invece c'è almeno un processo che non sia lo *swapper*, con stato `TASK_RUNNING`, lo scheduler deve verificare a che set di processi appartiene, se agli attivi oppure agli scaduti. Nel secondo caso, i puntatori ai due set di processi sono invertiti. Nel dettaglio: prima si controlla il valore della variabile `rq->nr_running` (totale di processi nelle code attive e scadute); poi se necessario (`!(rq->nr_running == 0)`), il processo *swapper* diventa `next`; se invece `rq->nr_running > 0` il processo (o i processi) possono appartenere sia a `rq->active` che a `rq->expired`: il "dubbio" viene sciolto analizzando il valore di `array->nr_active`. Per i curiosi:

```
if (unlikely(!array->nr_active)) {
        schedstat_inc(rq, sched_switch);
        rq->active = rq->expired;
        rq->expired = array;
        array = rq->active;
        rq->expired_timestamp = 0;
        rq->best_expired_prio = MAX_PRIO;
}
```

A questo punto la funzione `schedule()` deve determinare qual è il processo della runqueue che effettivamente ha priorità maggiore (ovvero che ha il valore di priorità dinamica minore). A tal fine è stata predisposta una bitmask (con tanti campi quante sono le priorità) che, in corrispondenza di una lista di priorità non vuota, ha il bit valorizzato; allo scheduler dunque sarà sufficiente cercare qual è il primo bit non nullo della bitmask. Il codice corrispondente a questi passi è piuttosto semplice ed è attribuito alla funzione `sched_find_first_bit()`.

Valore	Descrizione
0	Il processo era in uno stato `TASK_RUNNING`
1	Il processo era in uno stato `TASK_INTERRUPTIBLE` o in uno stato `TASK_STOPPED` ed è stato risvegliato da una chiamata di sistema o da un thread del kernel
2	Il processo era in uno stato `TASK_INTERRUPTIBLE` o in uno stato `TASK_STOPPED` ed è stato risvegliato da una routine di gestione di interrupt o da una softIRQ.
-1	Il processo era in uno stato `TASK_UNINTERRUPTIBLE` ed è stato risvegliato.

Tabella 3: Valori e significato del campo `activated`.

Sucessivamente si associa alla variabile locale `next` il puntatore al descrittore del processo che sostituirà `prev`; infine si invoca la funzione `context_switch()` che eseguirà lo "scambio" di contesto tra i due processi.

4.5.4.4 *Azioni svolte dalla funzione `schedule()` successivamente al contest switch*

Le funzioni che seguono lo scambio di contesto non saranno eseguite immediatamente dal processo `prev`, perchè la sua esecuzione è stata interrotta e tutte le variabili del processo, comprese quelle in kernel mode, sono state mantenute in una posizione in memoria. È intuitivo capire che problema sorgerà quando il processo `prev` verrà rieseguito: le proprie variabile si riferiranno al contesto in cui era stato lasciato, rendendo problematiche le funzioni dello schedurer successive al *contest switch* (per maggiori dettagli si rimanda al capitolo riguardante i processi ed alla figura 5). Grazie all'uso di un registro per salvare l'indirizzo dell'effettivo processo `prev`, le istruzioni "problematiche" non sono più tali.

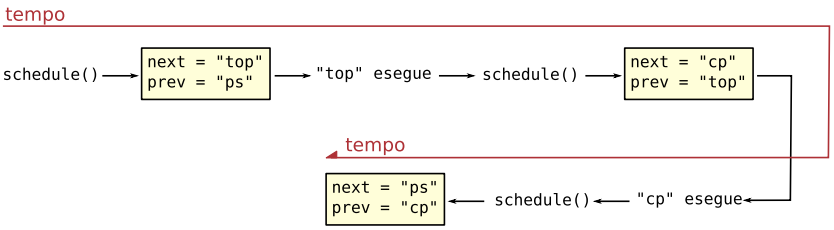

Figura 5: esempio di cambi di contesto

`finish_task_switch(prev)` controlla se il processo passato come parametro è uno zombie, in caso positivo avvia le procedure per eliminarlo. Le ultimissime istruzioni dopo opportune verifiche del caso, controllano se altri processi hanno posto il flag di quello corrente a `TIF_NEED_RESCHED`; in tal caso la funzione viene eseguita da capo, altrimenti termina.

Parte II

SPERIMENTAZIONE

5 SCENARIO DI PARTENZA

Ora sarà analizzata la situazione di partenza, in cui ci si soffermerà sulle motivazioni di questa sperimentazione.

Le macchine (spesso saranno indicate con l'acronimo Customer Premise Equipment (CPE)) su cui si è lavorato sono dotate di un chipset Broadcom che incorpora un processore MIPS a 300Mhz. La RAM a bordo del sistema è di "soli" 64MB. Apparentemente sia processore che memoria RAM sono molto modesti, ma, considerando la finalità di un modem/router, questi numeri si rivelano nella totalità dei casi sufficienti. Le uniche sofferenze che la macchina in analisi mostra sono isolate al caso di saturazione del traffico internet con connessione ottica: infatti, lo stesso software compilato ed eseguito su di una macchina con connessione ADSL convenzionale, non mostra nessun problema. Fare delle chiamate in queste condizioni è stato impossibile; la cornetta, una volta alzata, era completamente "muta".

WWW.BROADCOM.COM *per dettagli sui vari chipset della casa.*

Si sono potute constatare concretamente queste problematiche che erano già state anticipate. Durante i test era possibile usufruire di una macchina con uplink ADSL ed una con uplink ottico, tutte e due con lo stesso kernel, compilato con i rispettivi profili (per conessione convenzionale ed ottica); entrambe le macchine erano provviste di uno switch Gigabit Ethernet. Il computer utilizzato poteva accedere a queste macchine tramite telnet e porta seriale; sfruttando il collegamento ethernet si sono potute caricare le immagini prima compilate e scriverle sulla memoria flash a bordo delle CPE. Di seguito viene riportata una descrizione degli strumenti da utilizzati nel corso degli studi.

SMARTBITS Appena inziata la fase sperimentale si è subito potuto utilizzare uno speciale strumento atto a simulare un traffico di rete entrante ed uscente dal router su cui si sono svolti i test. Questo strumento di test era pilotabile da un altro computer mediante il proprio software applicativo; con questo si poteva decidere che percentuale di banda occupare e la grandezza dei pacchetti del traffico. Inoltre stampava la percentuale di `packet loss` per ogni test, ossia un indice del quantitativo di pacchetti andati perduti. I singoli test in realtà erano programmati per spaziare varie ampiezze di pacchetti e varie occupazioni di banda.

OPALTESTER Un altro strumento che si è rivelato fondamentale per l'analisi è stato un pacchetto applicativo appositamente realizzato da Telsey. Il programma (OpalTest) permette di testare la parte fonica della CPE; il software ripete l'esecuzione di una serie di eventi che simulano una chiamata. Esso è anche in grado di segnalare la possibilità di eseguire una chiamata correttamente; qualora ci siano problemi, (`no dial tone` e `no ringback tone` prevalentemente), il programma segnala `failed` il test. OpalTest permette all'utente di scegliere una vasta gamma di possibili simulazioni; in questo caso è stato utilizzato

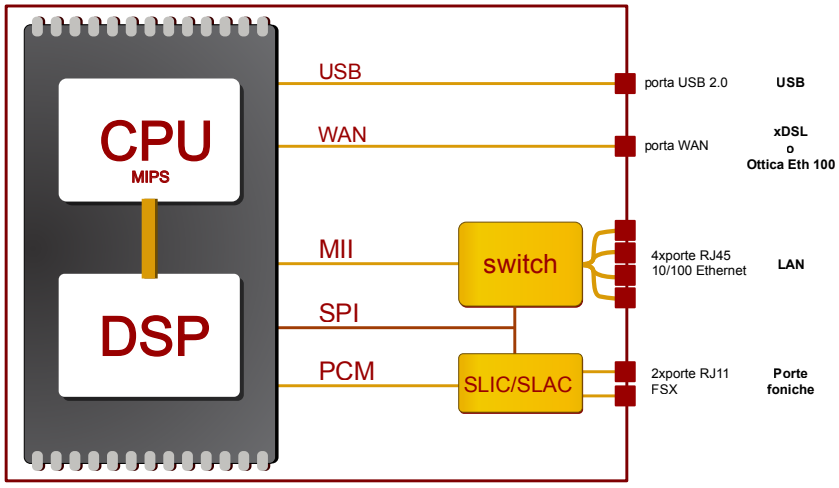

Figura 6: schema dell' architettura della CPE utilizzata

frame size (byte)	64			128		
% banda	20	30	40	40	50	60
packet loss	0.000	0.682	26.135	0.000	0.351	17.544
frame size (byte)	256			512		
% banda	70	80	90	80	90	100
packet loss	0.000	0.000	09.953	0.000	0.000	1.833
frame size (byte)	1518					
% banda	80	90	100			
packet loss	0.000	0.000	0.635			

Tabella 4: Andamento indice packet loss senza stress test

un test piuttosto generale, il cui scenario prevedeva 4 terminali telefonici che si chiamavano l'un altro ripetutamente.

SIMULATORE DI EVENTI TELEFONICI Questo dispositivo che è stato utilizzato nella prima parte dei test; anche questo realizzato da Telsey, permette la generazione di eventi telefonici quali: aggancio e sgancio della cornetta telefonica e generazione di segnali DTMF secondo definite tempistiche e sequenze, allo scopo di simulare un dispositivo telefonico. Questi eventi telefonici erano fittizi, ma possono essere certamente approssimati a quelli reali. Il dispositivo, sebbene funzionasse perfettamente, si è dimostrato marginalmente utile poichè non stressava sufficientemente il router.

5.1 ANALISI DELLA SITUAZIONE INIZIALE

Con "situazione iniziale" si intende il modem connesso con tutti i processi che eseguono, compresi quelli relativi al voip.

Utilizzando tutti gli strumenti descritti precedentemente è stato semplice disegnare un quadro della situazione di partenza. Come anticipato, le difficoltà del router risultano evidenti quando ci si avvicina ad una banda di 100Mbit/s; la tabella 4 da un'idea significativa del problema.

La tabella mostra che, più i pacchetti sono di piccole dimensioni, più il processore fa fatica a gestirli mentre, con pacchetti di grandi

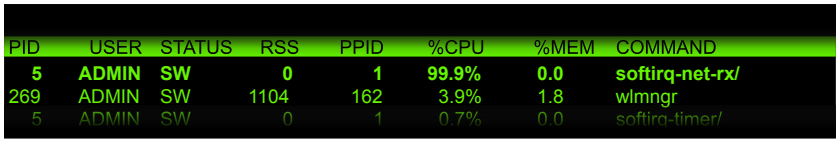

Figura 7: visualizzazione del comando top da una shell remota

dimensioni (1518 `byte`), è addirittura possibile saturare completamente la banda senza averne un'apprezzabile perdita. Durante questi test si era comunque liberi di eseguire il login sul router, anche se l'operazione era piuttosto difficoltosa; infatti l'elevato uso di CPU causato dal traffico di rete rendeva impraticapile la shell del sistema. Nonostante questa difficoltà, il comando `top` confermava il completo utilizzo della CPU.

Nell' immagine 7 si evidenzia il kernel thread `softirq-net-rx/` il quale è responsabile dello scodamento dei pacchetti `TCP`. É da precisare che, in questi test, il traffico generato (entrante nella porta WAN della CPE) era "passante", cioè non destinato alla CPE, di conseguenza non veniva terminato al layer IP ma inoltrato ad una porta LAN da una precisa regola di NAT; il pacchetto dunque tornava allo SmartBits.

i pacchetti una volta ricevuti non vengono terminati al layer IP ma vengono inoltrati dalla porta wan alla porta lan e viceversa.

6 SVILUPPO SOFTWARE, SDK

In questo capitolo verranno trattate tutte le modifiche che sono state attuate per predisporre la macchina ai test.

6.1 IPOTESI DI SOLUZIONE

Successivamente allo studio del kernel ed in maggior dettaglio dello scheduler, è stato possibile definire la strada e gli strumenti che sarebbero stati condotti ed utilizzati per cercare di risolvere le debolezze della CPE nelle particolari condizioni mostrate nel capitolo precedente. Si sono delineate due principali ipotesi di soluzione:

1. modificare lo scheduler del kernel cosi da privilegiare l'esecuzione del processo voce o più in generale apportare modifiche allo scheduler cosi da ottimizzarlo perfettamente per un router;
2. variare la priorità dei processi così da trovare il giusto bilanciamento di esecuzione dei processi coinvolti.

Questa scelta ha aperto la strada a nuovi studi e nuove considerazioni. La più importante riguarda la natura di uno scheduler in un sistema operativo: l'analisi dei numerosi changelog stilati puntualmente ad ogni rilascio di una nuova versione del kernel non può che confermare l'attenzione degli sviluppatori a rendere lo scheduler EQUO. Garantire equità tra i processi è una delle maggiori difficoltà di sviluppo; infatti la complessità del codice dello scheduler è dovuto quasi esclusivamente a questo importante requisito. Un altro obiettivo e quindi un' altra difficoltà è stata quella di rendere lo scheduler FLESSIBILE, adatto all'esecuzione in ogni contesto ed in ogni configurazione hardware e software. Queste due grandi peculiarità del kernel, che sono anche due suoi importanti punti di forza, hanno indotto a scartare la prima ipotesi, anche perchè la possibilità di variare la priorità dei processi nasce appunto per evitare complicate modifiche al file `sched.c`. La scelta e le inadeguate possibilità offerte dal software pre-installato sulle macchine router, hanno evidenziato la necessità di apportare alcune modifiche ed aggiunte.

Sviluppi

La casa produttrice del chipset fornisce un Software Development Kit (SDK), con il quale si può intervenire sul software che si installa sulle macchine. Poichè, come anticipato, le macchine sono provviste di un "cuore" Linux (alla versione 2.6.21.5), il software di supporto che è incluso nel SDK è modificabile. Parte di questo software consiste in una suite di comandi e programmi "base" che, a causa della natura (spiccatamente limitata) della macchina, dovrà essere piuttosto "snella". I progettisti del SDK sono ricorsi quindi alla versione 1.0 di BusyBox, nota suite di programmi (utilizzabili tutti da shell) che ripropone, semplificando, i classici programmi di Linux e delle shell UNIX in generale. Uno dei più utili in questo studio è stato `ps`, comando per

Per maggiori informazioni su BusyBox rimando alla pagina web del progetto: WWW.BUSYBOX.NET

```
renice PRIO/NICE rt/other policy/PID_1 start_pid/PID_2
stop_pid/PID_3 PID_4 PID_5 ...

esempi:

Processi RT      //prio=65 policy=RR ai processi 25,26,27,28,29
renice 65 rt 2 25 29

Processi OTHER  //prio=110 ai processi 25,26,27,28,29
renice -10 other 25 26 27 28 29

Dove:
        SCHED_NORMAL =          0
        SCHED_FIFO   =          1
        SCHED_RR     =          2
        SCHED_BATCH  =          3
```

Figura 8: Sintassi del comando `renice`

visualizzare lo stato dei processi della macchina. Nella versione "snella" `ps` era troppo limitato poichè visualizzava solamente PID, lo stato del programma e poco altro. Questa limitazione che presenta è giustificata dal tipo di uso per cui il comando è predisposto: non è necessaria, in un router, una suite di comandi completa e di elevata scalabilità. Inoltre, l'utenza finale non ha nemmeno l'accesso alla shell del terminale per cui curare questo aspetto non è stato tra le finalità dei teams di sviluppo che hanno curato lo SDK. Tuttavia per questo studio `ps` con le caratteristiche sopra descritte, era poco utile e grazie alle modifiche apportate, è stato possibile stampare a schermo anche priorità statica, NICE, priorità (dinamica) e policy di scheduling di tutti i processi.

Revisione del comando `renice`.

Il codice del comando per quanto riguarda i processi convenzionali non è stata alterata, ad eccezione del flag in modo tale da distinguere i due tipi di argomenti: processi sched other e real-time.

Nel corso dei test si è mostrata la necessità di modificare la priorità di alcuni processi, operazione che in una fresca installazione di qualsiasi distribuzione Linux è piuttosto comune e relativamente semplice. I due comandi che svolgono l'aggiornamento di priorità sono `renice` e `chrt`, il primo per processi convenzionali, il secondo per processi real-time. La macchina su cui si è lavorato (o meglio, il software della macchina) non prevedeva questi due comandi. Ottenere `renice` è stato semplice: essendo incluso nella suite BusyBox è stato sufficiente ricompilare con opportune modifiche l'immagine della macchina per renderlo disponibile su shell. Il programma `chrt` era però indispensabile e mancava: la soluzione più semplice è stata quella di modificare il sorgente di `renice` così da potergli consentire di modificare anche priorità e policy di processi real-time. La sintassi del comando modificato è riportato in figura 8.

Importanza del `proc` file system.

Tutti i suddetti comandi si appoggiano su di uno speciale file system che di default viene montato su `/proc/`. Questo file system si differenzia da tutti gli altri per una peculiarità: non fa riferimento a nessun dispositivo fisico, i dati che contiene danno una rappresentazione dello stato del kernel e di tutte le strutture dati ad esso connesso.

Nel dettaglio, sulla *sub directory* `/proc/PID`, dove con "PID" indico

l'identificativo di un qualsiasi processo, è costantemente aggiornato il file `stat`, contenente una lista piuttosto ampia di numeri, tra i quali spiccano per importanza:

`man proc` per maggiori dettagli.

- stato;
- priorità;
- nice;
- priorità rt.

tutti riferiti al processo "PID". L'analisi del suddetto file è stata obbligatoria, in quanto tutti i programmi in Linux che riportano lo stato di qualche processo, fanno riferimento a questo file. Basti pensare allo stesso comando top, oppure, in un contesto più comune, al programma di Gnome "`gnome-system-monitor`"; tutti questi, accumunati dall'obiettivo di riportare un'*istantanea* dello stato dei processi, traggono le informazioni dal file `/proc/PID/stat`.

Con le modifiche analizzate nei paragrafi precedenti è stato possibile avere il pieno controllo sui processi della macchina.

7 TEST SPERIMENTALI E CONCLUSIONI

7.1 IL SERVIZIO VOCE

Con un elevato stress di rete, la CPU del router viene occupata totalmente; ciò avviene a maggior ragione se il traffico sottoposto consiste principalmente di pacchetti di piccola taglia. Ci sono due thread che gestiscono gli interrupt delle schede di rete della CPE: `softirq-net-rx` e `softirq-net-tx`; quando si avvia il carico di rete, lo scheduler, trovandosi in un contesto con un processo real-time (`softirq-net-rx`) sempre `RUNNING`, è programmato per concedergli l'esecuzione fintanto che questo non sia sorpassato in priorità da altri processi o abbia finito l'esecuzione. Ed è quello che effettivamente è accaduta durante i test. Il servizio voce è composto da un insieme di più thread, tutti con priorità differenti e funzioni differenti. Questa pluralità è stata uno dei maggiori fattori che hanno reso gli interventi software difficoltosi.

L'approccio è stato il seguente: modificare la priorità dei processi voce e dei thread kernel relativi alla rete al fine di migliorare la risposta del software voip.

7.1.1 *Configurazione di test*

La configurazione utilizzata per svolgere il test è illustrata in figura 9; come si vede in totale sono servite due CPE. La cossidetta Device Under Testing (DUT), era dotata di un uplink ottico, di porte lan e due prese foniche (FXS) che erano direttamente connesse al all'OpalTester. La fibra ottica uscente dall'uplink era diretta su di un MEDIA CONVERTER che adattava il segnale al mezzo trasmissivo ethernet mentre una delle porte ethernet era connessa allo Smartbits. La seconda CPE ai capi delle due prese foniche era connessa all'OpalTester, mentre l'uplink ADSL era connesso al router tramite un Digital Subscriber Line Access Multiplexer (DSLAM) e un Broadband Remote Access Server (BRAS), obbligatorio in questo caso per ottenere un segnale utile e per dirigerlo al router. Il server SIP era connesso al Router. Queste macchine dovevano permettere alle due CPE di poter effettuare chiamate. Esso aveva quindi la funzione di pilotare i due "interlocutori" affinchè lo scenario fosse il più completo possibile. L' OpalTester inoltre tracciava un log con tutte le specifiche del test, compresa la riuscita o meno delle chiamate. Lo Smartbits invece, come descritto precedentemente, si limitava a caricare di traffico la periferica DUT e a fare un reso dell'indice di packet loss. Lo scenario, eseguito varie volte, può essere schematizzato come segue:

1. il TELEFONO 1 sgancia;
2. la CPE 1 genera il dialtone verso il TELEFONO 1;
3. il TELEFONO 1 compone il numero;
4. la CPE 1 riceve le cifre;
5. la CPE 1 invia al sip server un `INVITE` per il numero composto;
6. il sip server inoltra l'`INVITE` alla CPE 2
7. la CPE 2 in sequenza:
   a) comincia a fare suonare il TELEFONO 2;
   b) invia una notifica di ringing alla CPE 1 (`180 RINGING`);

8. la CPE 1 in sequenza:
   a) riceve la notifica;
   b) genera un tono di ringback verso il TELEFONO 1;
9. il TELEFONO 2 sgancia;
10. la CPE 2 in sequenza:
    a) invia una notifica di chiamata stabilita alla CPE 1;
    b) inizia a trasmettere il flusso media tramite RTP;
11. la CPE 1 in sequenza:
    a) riceve la notifica;
    b) inizia anch'essa a trasmettere il flusso media tramite RTP;

    [chiamata in corso]
12. il TELEFONO 2 riaggancia;
13. la CPE 2 in sequenza:
    a) interrompe il flusso media RTP;
    b) invia alla CPE 1 la notifica di chiamata terminata;
14. la CPE 1 in sequenza:
    a) riceve la notifica;
    b) interrompe il flusso media RTP;
    c) genera il tono di chiamata disconnessa verso il TELEFONO 1;
15. il TELEFONO 1 riaggancia.

7.1.2 *Conclusioni*

Prima di trarre delle conclusioni, devono essere fatte delle considerazioni. In primo luogo c'è da dire che il test era piuttosto complesso, sia a livello di mezzi e configurazioni utilizzate, sia dal punto di vista delle CPE; essi si trovavano in condizioni piuttosto critiche, con un carico elevato che gravava sulla CPU. Per fronteggiare il problema sono state provate varie combinazioni di priorità; tutti i processi coinvolti, numerosi, sono stati "alterati" dalla loro condizione originaria: vista la complessità del processo voce, creare una configurazione di priorità tale da garantire stabilità permanente nel tempo, era quindi un'impresa complessa.

Il processo voce è in realtà un insieme di processi, ognuno con scopi e complessità differenti che, nel complesso, assolvono allo stesso obiettivo: stabilire una chiamata voce. É possibile definire le principali funzioni del processo con i seguenti punti:

- segnalazione e gestione degli eventi di linea rilevati dallo SLIC ;
- convertire il segnale prelevato dalle porte foniche in segnale digitale;
- convertire il segnale inviato alle porte foniche in segnale analogico;
- controllare il DSP affinchè pacchettizzi il traffico digitale (PCM) sul protocollo RTP;
- trasmettere e ricevere il traffico RTP tra le interfacce DSP e quella di rete;
- gestione della chiamata Voice over IP (VOIP) attraverso la macchina a stati definita dal protocollo Session Initiation Protocol (SIP).

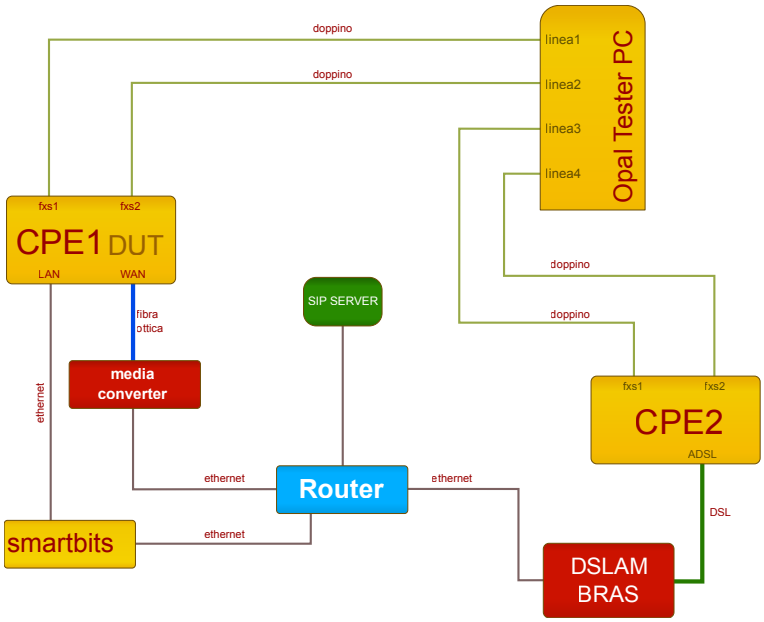

Figura 9: schema dell' architettura utilizzata

Uno dei maggiori errori che si è riscontrato durante i test era il NO DIAL TONE che si riferiva all'impossibilità della CPE chiamante di generare un tono di invito alla digitazione (ovvero il tono che segue lo sganciamento della cornetta) verso il telefono da cui essa si è sganciata; l'OpalTester rilevava questa anomalia, e la segnalava riportando un errore nella chiamata. Il problema trovava spiegazione sull'elevato uso di processore della macchina e comportava che il processo che svolgeva tale funzione (una delle tante della macchina a stati "voce"), non riusciva ad eseguire.
I test e le modifiche della configurazione software sono proseguite fino al raggiungimento di una condizione in cui la chiamata comunque non andava a buon fine, ma il fallimento era imputabile ad un problema differente dal precedente. Il DUT infatti eseguiva correttamente, seppur con ovvii ritardi, tutti i thread coinvolti nel processo voce, ma la congestione di rete era tale da rendere impossibile la comunicazione tra gli apparati connessi alla rete LAN e WAN, rendendo vane le richieste inoltrate dall'OpalTester. Questa particolare situazione generava un secondo errore, il NO RINGBACK TONE che sintetizzava l'assenza del tono in cornetta indicando l'avvenuta risposta della chiamata dal telefono chiamato. In realtà questo errore poteva riferirsi a due scenari simili dove, in entrambi, il TELEFONO 1 chiama il TELEFONO 2:

- il TELEFONO 2 non risponde: il comando di richiesta (`INVITE`) della chiamata non perviene alla CPE chiamata.
- il TELEFONO 2 effettivamente risponde, ma la notifica di avvenuta risposta (`180 RINGING`) non perviene alla CPE chiamante. In questo caso il comando di risposta è perso;

Questa tipologia di errore dipendeva quasi esclusivamente dall'intensità di traffico con cui si appesantiva la DUT e non da una non ottimale configurazione delle priorità dei processi coinvolti nei tentativi di chiamata.

7.2 PERIFERICHE USB

Le problematiche del caso precedente hanno indotto all'analisi di uno studio più contenuto, che avrebbe reagito più prontamente alle modifiche software. Il problema in questione è legato alle performance della lettura/scrittura di dati sulle periferiche USB parallelamente al traffico di rete. Lo scenario previsto per questi test consisteva nel sottoporre la CPU ad uno stress della macchina più limitato; si è partiti infatti da un'occupazione del processore di circa il 50% grazie ad un traffico di 40Mbits con pacchetti da 512 byte. Già in queste condizioni di uso piuttosto comune, il trasferimento di 100MB di dati da una chiavetta ad un altra richiedeva un tempo inaccettabile.

In condizioni di completa disponibilità di CPU, per copiare 100MB da una chiavetta all'altra la CPE impiega circa 20 secondi

I test seguenti si prefiggevano lo scopo di trovare un giusto compromesso tra tempo di trasferimento dei dati e indice di `packet loss`.
Per rendere più veritiere le varie prove, sono stati spaziati più rates di occupazione della rete e più configurazioni di priorità tra i processi coinvolti. Analizzando il comportamento dei vai processi è stato possibile limitare a pochi di quelli l'attività di trasferimento dei dati: nella fattispecie sono stati coinvolti i threads dei driver USB (che indicherò con Kthreads) ed ovviamente il processo di copia stesso; nel nostro caso "`cp`". `cp`, durante la sua esecuzione, attraverso una chiamata di sistema ha interpellato il kernel affinchè questo legga e scriva sulle chiavette USB. Le operazioni che effettivamente scrivono i byte sulla memoria flash sono gestite dai driver USB che figurano al kernel come altri due processi. Per questi motivi (nei grafici successivi sarà evidente) non è bastato variare di priorità solo il processo `cp` (che di default si inizializza con nice=0), bensì si è dovuto intervenire organicamente sui suddetti threads del kernel; questo è stato l'elemento di maggiore difficoltà dell'intervento. La priorità che si è rivelata ideale allo scopo assume il valore 10; il valore infatti doveva garantire una priorità di poco maggiore dei threads di gestione del traffico di rete, e per questo doveva essere real-time. Inizialmente, per sperimentare, si è provata una priorità inferiore alla priorità 5 dei processi del traffico di rete ed effettivamente i cambiamenti non hanno portato ai risultati voluti e la copia dei file nella chiava USB era nettamente rallentata.

7.2.1 *Benchmark periferiche USB, considerazioni*

A conclusione dei vari test, è stato possibile stilare la tabella 5. In essa è possibile distinguere i vari risultati dei test, che dipendono da:

- priorità del processo cp, che di default è pari a 0, non real-time;
- priorità dei threads del kernel, che di default è pari a -10, non real-time;
- presenza del simulatore di eventi telefonici;
- rate stress della rete.

Alla luce dei dati raccolti è bene fare delle considerazioni di natura qualitativa. Indubbiamente il tempo di lettura e scrittura totale è notevolmente migliorato aumentando la priorità del processo `cp` e dei vari kernel threads; tutto però ha un costo. Come si evince dal grafico 11, l'aver alzato considerevolmente la priorità dei due gruppi di processo

ha portato ad un aumento dell'indice di `packet loss`, che inizialmente si attestava su un valore prossimo allo zero. Lo scheduler, inoltre, negli intervalli in cui i processi di copia rilasciavano la CPU, continuava a scodare i numerossisimi pacchetti "iniettati" dalla rete e quando il dispositivo USB, tramite interrupt, segnalava l'avvenuta scrittura del byte al kernel, questo immediatamente risvegliava i processi di copia, provocando inesorabilmente perdita di pacchetti. Il valore di packetloss è accettabile in una condizione di carico della macchina dove l'effetto che provoca sulla maggior parte delle applicazioni di tipo residenziale non è percepibile. Ciò non è però più vero in particolari contesti che prevedono trasmissioni real time. Per rendere il test più veritiero si è anche attivato il simulatore di eventi telefonici, che, anche se minimamente, ha caricato computazionalmente il MIPS. Il grafico 10 ci suggerische che effettivamente c'è stato un minimo aumento della durata del processo di copia. La cosa più interessante è mostrata dal grafico 11: l'indice di `packet loss`, nel rate del 40-50%, è molto simile in tutti i quattro i casi. A percentuali maggiori, l' indice aumenta velocemente, mantenendosi comunque al di sotto di una modesta soglia di sei punti percentuale. Significativo è il valore assunto dal grafico in corrispondenza delle 70Mbs di traffico: l'indice di packetloss con le priorità settate a 10 senza simulatore di eventi telefonici è circa uguale allo scenario con disturbo e priorità a valori di default; rivedendo il concetto, si può quindi affermare che le modifiche di priorità possono essere approssimate ad un piccolo disturbo di rete, evento piuttosto comune.

```
1 /bin/sh
2 PRIO=10
3 cp $1 $2 &
4 pid=‘pidof -o exe‘
5 renice $PRIO rt 2 $pid $pid
```

Workaround necessario per poter eseguire il programma `cp` con la priorità voluta, si tratta di un semplice script shell

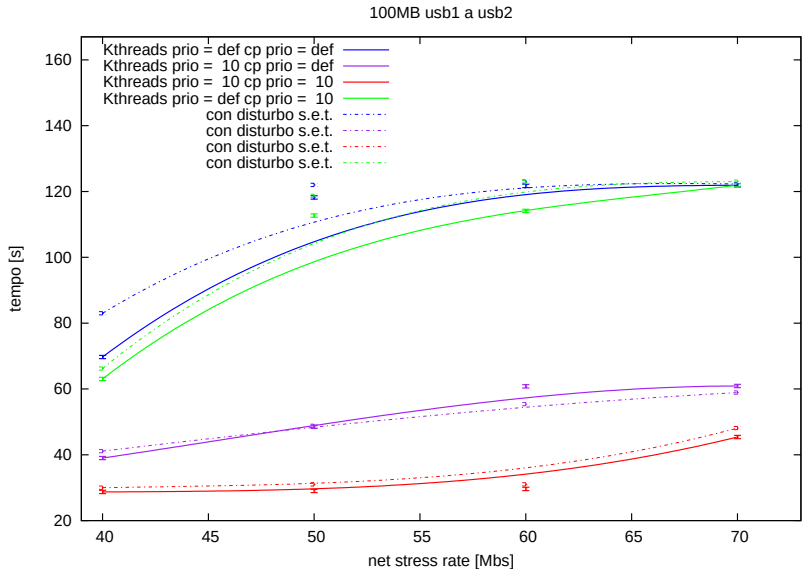

Figura 10: copia 100MB

1) variaizone pacchetti piccoli/grandi

configurazione		s.e.t	rate stress rete			
cp prio	Kthreads prio		40%	50%	60%	70%
DEFAULT		NO	69.7s	118.2s	121.7s	121.9s
			0.0%	0.009%	0.01%	0.015%
DEFAULT		SI	83.0s	122.1s	123.2s	122.3s
			1.35%	1.72%	2.06%	2.12%
DEF	10	NO	63.4s	112.7s	114.1s	121.9s
			0.00%	0.00%	0.2%	0.25%
DEF	10	SI	66.5s	118.7s	122.8s	123.0s
			1.33%	1.66%	2.05%	2.35%
10	DEF	NO	39.0s	48.5s	60.8s	60.9s
			0.01%	0.24%	0.37%	0.45%
10	DEF	SI	41.1s	48.8s	55.4s	58.9s
			1.56%	2.02%	3.17%	3.75%
10	10	NO	28.7s	29.0s	29.6s	45.4s
			0.21%	0.5%	0.89%	2.32%
10	10	SI	30.0s	31.0s	31.2s	48.1s
			1.69%	2.17%	2.79%	5.4%

Tabella 5: Rilevamenti sul trasferimento di 100MB; la colonna s.e.t indica la presenta del simulatore di eventi telefonici

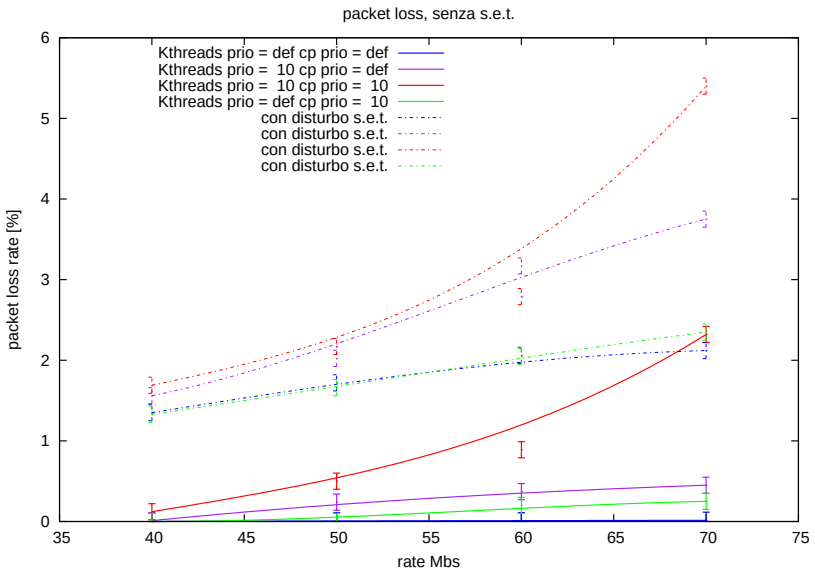

Figura 11: PacketLoss