Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA SPECIALISTICA

Analisi e sviluppo di un framework integrato per il tracciamento di task Linux ed RTAI

Relatore
Professore Sergio Congiu
Correlatore
Ingegnere Antonio Barbalace

Laureando
Enrico Marini

ANNO ACCADEMICO 2009/2010

Dedicata alla famiglia, agli amici, a me

Indice generale

Indice delle illustrazioni

1 Introduzione e obiettivi

Nel corso dell'esecuzione di un sistema operativo, da parte di un elaboratore, si alternano moltissime attività che svolgono determinati compiti: controllo dello stato del computer, lettura o scrittura di file, programmi di divertimento, gestione delle risorse critiche, ecc. L'attività è meglio nota come processo, task, ovvero una sequenza di passi (più a basso livello, istruzioni) lanciata ed eseguita dalla CPU (Central Processing Unit), il cuore di un computer. I task hanno un ciclo di vita: nascono, vengono eseguiti e terminano. Durante la loro vita possono assumere diversi stati, tra i quali pronto, sospeso, terminato (ready, suspended, terminated).
Poiché l'esecuzione dei processi può essere soggetta a bug (errori logici, deadlock, ...) bisogna essere in grado di controllarla, monitorarla in qualche modo. È, quindi, utile poter ottenere un “tracciamento” (“tracing”) dei task, attraverso i loro stati, per avere un resoconto di come le attività vivono, istante per istante, nel sistema (seguire cosa accade).
È possibile costruire questo report, come sarà spiegato, manualmente, oppure, con gli strumenti di tracing (oggetto di studio). Il “tracciato”, così prodotto, può servire anche per verificare il tempo di esecuzione stesso dei processi.
Qualche tempo fa nelle applicazioni industriali si usavano piccoli sistemi dedicati che costavano molto ed erano completamente deterministici; quindi, la verifica del tempo di esecuzione dei task con il tracing non era necessaria, perché era sufficiente contare i tempi di esecuzione delle loro istruzioni.
Ora ci sono applicazioni industriali con sistemi general purpose, meno costosi, che sono resi non deterministici dall'uso di cache/pipeline e, allora, il tracing diventa essenziale per capire i tempi di esecuzione.
Finora si è parlato di processi, senza particolarità precise: task normali che vengono elaborati, senza scadenze, in un sistema.
Oggigiorno, nelle applicazioni lavorative ed industriali, i task in esecuzione non sono solo task normali. Esistono, infatti, processi sostanzialmente diversi da quelli normali, ovvero processi con caratteristiche aggiuntive: le attività svolte sono scandite dal tempo e devono terminare entro e non oltre un certo limite (deadline), pena il non corretto funzionamento del sistema in cui sono inserite. Questo distingue i cosiddetti task real time. Real time, infatti, sta ad indicare che i processi (attività da eseguire) devono rispettare certi vincoli temporali, ovvero devono terminare entro un tempo predeterminato e, quindi, è fondamentale poter calcolare il loro tempo di esecuzione.
Con questa aggiunta sulla struttura dei processi, l'analisi di essi diventa ancora più interessante.
Si comincia, quindi, ad intuire l'importanza del tracing a livello normale e real time: avere un quadro che mostra, in punti precisi, la situazione dei processi del sistema.
La base di lavoro si appoggia sul mondo “open source”:

- distribuzione Linux, come sistema base;
- RTAI (Real Time Application Interface), come estensione real time di Linux.

Venendo, dunque, agli obiettivi, quello che ci si propone con questa tesi è di studiare alcuni strumenti di tracing esistenti per Linux: nello specifico sarà approfondito ftrace, che sembra essersi affermato nella comunità open source.
Dopo aver studiato la parte di ftrace relativa al tracciamento dei task (nel dettaglio gli switch tra essi e le loro sveglie) si valuta se è necessario integrarla in RTAI per poter tracciare i task di questo sistema, sia quelli in kernel space, sia quelli in user space. Il tracing viene prodotto in formato testuale e tramite l'utilizzo di un altro programma può essere visualizzato anche graficamente.

2 Ambiente di lavoro

Il lavoro è svolto su una macchina monoprocessore single core. La scelta di Linux è dettata dal fatto che l'estensione RTAI è scritta per questo sistema: sono stati installati Linux Debian Lenny 5.0 e RTAI versione 3.7.1.
Le prove per poter verificare la stabilità di quanto realizzato sono state svolte su questa macchina.
Ora segue una breve descrizione di Linux e RTAI.

2.1 Linux

Linux (GNU/Linux) è un termine generico che si riferisce ai sistemi operativi per computer Unix-like basati sul kernel Linux. Il loro sviluppo è uno degli esempi di rilievo della collaborazione del software libero e open source; tipicamente tutto il codice sorgente può essere usato, liberamente modificato e ridistribuito da qualsiasi persona sotto i termini della GNU GPL (GNU General Public License) e altre licenze per software libero.
Linux è un sistema multitasking (più programmi possono essere eseguiti contemporaneamente) e multiutente (più utenti in simultanea). Ha un'architettura divisa fra il kernel, che è il nucleo del sistema che gestisce le risorse hardware (processore, memoria, periferiche) ed i processi (comandi base, applicazioni, interfacce grafiche). Seguono, quindi, i concetti di kernel space, dove viene eseguito il kernel e user space, che è l'ambiente dove girano i processi. Solo il kernel ha la possibilità di accedere direttamente alle risorse hardware della macchina, i processi devono, invece, usare le “system call” (chiamate di sistema).
Il kernel è scritto, prevalentemente, nel linguaggio C, a parte in certi punti dove è usato l'assembly.

2.2 RTAI

RTAI o Real Time Application Interface è una modifica del codice sorgente del kernel di Linux soprattutto per quanto riguarda le politiche di scheduling e di interrupt.
Questa estensione hard real time realizzata dal Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano (DIAPM), come variante di RTLinux, è una suite di aggiornamenti e librerie che consente comunque le normali funzionalità di un normale kernel Linux. In quanto real time, il sistema deve rispettare scadenze temporali in modo incondizionato nell’esecuzione dei processi soprattutto in termini di latenza e predicibilità. Un processo hard real time dovrà, perciò, avere una priorità più alta rispetto a tutti gli altri processi, i quali vengono accodati dopo di esso.
Esso utilizza un approccio a microkernel, sfruttando la modularità di Linux per

ottenere un sistema real time poco invasivo e di grande flessibilità. A partire da Linux per ottenere un comportamento preciso e sincronizzato, adatto ad un ambiente hard real time, è necessario agire sulla politica di scheduling, sulla possibilità di preemption di porzioni del kernel e sulla gestione delle interruzioni.
RTAI, quindi, crea un'interfaccia tra l'hardware della macchina e il sistema operativo, subentrando a quest'ultimo nella gestione delle interruzioni e nello scheduling dei processi in tempo reale e relegando Linux come task in background da eseguirsi quando non ci sono attività real time incombenti.
Essendo RTAI disponibile come modulo kernel di Linux, è caricabile liberamente a sistema avviato. Permette, quindi, di estendere dinamicamente le funzionalità del kernel senza dover essere caricato al boot di sistema.
RTAI permette una gestione dei processi con prelazione (preemption), in funzione delle loro priorità. Lo scheduler è configurabile a seconda della specifica combinazione di hardware e requisiti software da soddisfare attraverso l'API messa a disposizione dello sviluppatore.
I componenti di RTAI, moduli del kernel caricabili, con rispetto delle dipendenze, sono:

- RTHAL (Real Time Hardware Abstraction Layer);
- dispatcher;
- scheduler;
- IPC (Inter Process Communication).

I task sono organizzati secondo una gerarchia di priorità. Quando lo scheduler viene invocato viene messo in esecuzione il task READY a priorità più elevata. Se nessun task di questo tipo viene trovato, allora il controllo del processore passa al kernel di Linux.
RTHAL maschera tutte le strutture e le routine di gestione delle interruzioni del kernel di Linux. Quando un'interruzione arriva, il dispatcher viene invocato (preemption totale). Nel caso esista una ISR (Interrupt Service Routine - routine di servizio dell'interruzione) real time l'interruzione viene passata a questa, altrimenti viene messa in lista come interruzione pendente e il controllo passa allo scheduler. Quando non ci sono più processi real time READY, viene eseguito il kernel di Linux e l'interruzione viene gestita da quest'ultimo.

RTHAL

RTHAL è un nuovo layer tra il kernel Linux e il sistema hardware.

RTHAL racchiude tutti i dati e le funzioni temporalmente critiche del kernel in un’unica struttura:

- permette di catturare facilmente le funzionalità del kernel (interrupt, system call, timer) per poterle gestire in accordo a politiche real time;
- sostituisce le operazioni su strutture originali con operazioni su puntatori RTHAL;
- i puntatori RTHAL sono modificabili dinamicamente: se RTAI non è attivo puntano alle strutture originali di Linux, se RTAI è attivo puntano alle strutture del kernel real time.

RTHAL non fornisce servizi real time: ha la sola funzione di intercettare le chiamate al kernel Linux.
Si possono distinguere due modalità di funzionamento:

- RTAI non attivo, funzionamento normale di Linux;
- RTAI attivo, solo funzionalità real time possono accedere direttamente all'hardware. Linux è gestito come un processo a bassa priorità.

Dispatcher

Il dispatcher viene invocato ad ogni interrupt: preemption totale.

Scheduler

Ad ogni evento temporale o quando un task invoca una primitiva di sistema bloccante:

- i task coinvolti dall'evento o nella chiamata possono cambiare stato;
- il task di priorità più alta tra quelli READY viene eletto.

Il task eletto esegue fino al completamento a meno che:

- un task con priorità più alta non venga eletto;
- non venga terminato;
- non richiami una primitiva di sistema bloccante.

IPC

RTAI fornisce i seguenti meccanismi di comunicazione interprocesso:

- memoria condivisa;
- semafori;
- messaggi;
- RPC stile QNX;
- mailbox;
- real time FIFO tra task real time o fra task real time e Linux.

Sono, inoltre, disponibili le code di messaggi come meccanismi POSIX.

3 Panoramica sugli strumenti di tracing

Finora la parola “tracing” è solo stata nominata e mai spiegata, solo accennata negli obiettivi. Le domande “Cosa vuol dire?” e “Come si esegue?” sono legittime: bisogna, quindi, avere chiaro questo concetto su cui la tesi si focalizza.
Prima di parlare degli strumenti si vuole dare una risposta esauriente ad entrambe queste domande, costituendo, così, la base di partenza.
Gli strumenti sono visti come parte della risposta alla seconda domanda e se ne descriveranno le caratteristiche. Poi seguirà la descrizione di due dei tool di tracing più diffusi per Linux (LTTng e DTrace); mentre ad ftrace sarà dedicato il capitolo successivo.

3.1 Tracing: cos'è?

Si chiama “tracing” l'attività volta a raccogliere informazioni riguardanti l'esecuzione di un programma: valori di variabili, stampe di controllo, passaggi per punti critici, ecc. Serve a dare una visuale particolarizzata di quello che accade nel programma. È un tipo di debugging: infatti, queste informazioni vengono usate dagli sviluppatori per diagnosticare problemi, anomalie, malfunzionamenti nel software da loro prodotto. Il debug è un'attività fondamentale, che va sempre eseguita per ogni software che si vuole creare: un software “non debuggato” è un rischio per gli utenti che devono usarlo.
Il “tracing” coinvolto in questa tesi è leggermente diverso, come accennato precedentemente: riguarda il controllo dei task che vengono eseguiti all'interno di un sistema operativo (normale e real time). In particolare le informazioni che, in questo caso, si vogliono ottenere hanno a che fare con:

- **commutazioni di contesto** (**context switches**): ad un processo viene tolta la CPU e viene data ad un altro, cambia l'attività che si stava facendo (in termini tecnici, il task corrente subisce preemption a favore di un altro task più prioritario, vengono salvati lo stack e i registri, contesto, di modo che possano essere ripristinati quando quel task tornerà in esecuzione);
- **sveglie**: un processo smette di “dormire”, esce dalla sospensione o dal blocco in cui era finito e diventa pronto per riavere la CPU;
- **pid**, **priorità** e **stato**: questi tre dati sono specifici di ogni task, il pid (Process IDentificator) è un numero univoco che, appunto, lo identifica, la priorità segnala la sua importanza e lo stato dice, rispetto alla sua esecuzione, come si trova (pronto, sospeso, bloccato).

Il set di informazioni appena enunciato si rende necessario per i programmatori, al fine di capire come gli oggetti task del sistema vivono e si relazionano tra di loro.
L'obiettivo del lavoro non riguarda solo i semplici processi, ma anche quelli real time. Fornire un monitor sui task in tempo reale, al giorno d'oggi, è un'operazione interessante, ma che richiede particolari attenzioni sulla loro

vita. Il lavoro vuole, quindi, fornire un tracciamento dei task, sia real time, sia normali, che girano nel sistema.
Il tracing, in generale, si applica per qualsiasi dato si voglia analizzare e tracciare nel tempo: da una semplice variabile ad, addirittura, un processo. Ftrace, come si vedrà, permette il tracing delle funzioni che un kernel Linux esegue, della potenza del processore, chiaramente, degli switch tra task e altro.

3.2 Tracing: come?

Chiarito cosa intendiamo, in questa tesi, per tracing, si deve rispondere alla seconda domanda di prima: “Come si esegue?”.
In questo paragrafo si vuole, quindi, esporre le possibili vie per creare un tracing.
Nella pratica esistono due modi per fare dei tracciamenti.

3.2.1 Tracing manuale

Il primo è quello manuale ed è adatto per molti programmi applicativi: lo sviluppatore dell'applicazione sceglie i dati che devono essere analizzati, nel codice sorgente decide i punti critici per quei dati e inserisce dei controlli (di solito stampe, ma possono essere più complessi) per tenere sott'occhio l'evoluzione temporale di questi dati. Il programmatore deve, quindi, assumersi del compito di far stampare all'applicazione le informazioni che vuole sapere.
I vantaggi di questa scelta sono: lo sviluppatore ha la piena supervisione sia sul programma sia sui controlli da eseguire, visto che è lui stesso che li inserisce, quindi, può capire immediatamente quale è o quali sono i problemi che si verificano.
Gli svantaggi sono: i dati da monitorare non possono essere più di variabili o strutture, perché, altrimenti, l'azione stessa di debug si trasformerebbe nella scrittura di un programma di debug e porterebbe via tempo allo sviluppatore (ad esempio, il controllo di task diventa difficoltoso con questo approccio) e, inoltre, una volta terminato il debugging, i controlli, se non necessari all'applicativo, devono essere rimossi, perché l'appesantirebbero e lo farebbero eseguire più lentamente, soprattutto se il debug era oneroso.

3.2.2 Strumenti di tracing

L'altra soluzione è quella di usare qualcosa di già pronto, ovvero degli strumenti costruiti appositamente per fare tracing. Questi tool, che sono rivolti per gli oggetti del sistema operativo, sono molto più complessi di banali stampe e mettono a disposizione diverse possibilità di analisi.
Possono tracciare funzioni, processi, potenza, latenze, abilitazione/disabilitazione di interruzioni, con visualizzazione testuale e, per alcuni, anche grafica dei risultati ottenuti.
In sostanza, specificata l'analisi che l'utente vuole, questi seguono l'evoluzione

dei dati richiesti memorizzandola e alla fine del debug la presentano.
A favore di ciò c'è il fatto che lo strumento è indipendente, non deve essere gestito dal programmatore, è sufficiente avviarlo e fermarlo e permette di controllare i dati di interesse (task, ecc.).
Di negativo c'è, invece, che questi strumenti non sono adatti a produrre semplici tracing, come quello di variabili.

3.2.3 Confronto

Il primo metodo è utile se si desidera fare un debugging particolarizzato, ma semplice di un applicativo che si sta scrivendo, il secondo serve per esplorare delle parti del sistema operativo e, quindi, quella che è svolta, è un'attività più complicata. La scelta va effettuata in base a cosa bisogna controllare: per le variabili va bene quello manuale, con la consapevolezza di individuare correttamente i punti critici; per oggetti più complessi è meglio optare per un tool che supporti il loro tracciamento.

3.3 LTTng

LTTng (Linux Trace Toolkit Next Generation) è un tracer per Linux. È una patch del kernel accompagnata da un insieme di strumenti (ltt-control) per controllare il tracing come anche da un programma di analisi e di visualizzazione di trace (LTTV). LTTng include un insieme di punti di verifica del kernel utile per debuggare un ampio range di bug, che in altro modo sarebbe estremamente impegnativo. Questi includono, per esempio, problemi di prestazioni su sistemi paralleli e su sistemi real time. Strumentazione personalizzata può essere aggiunta con facilità. LTTng è progettato per avere un minimo impatto sulle prestazioni e un impatto quasi nullo quando sta tracciando. Ha un supporto base per almeno tutte le architetture Linux.
LTTV (LTTng Viewer) è un visualizzatore ed analizzatore di trace sia grafico sia testuale. Ha un'architettura basata sui plug-in.
LTTng/LTTV sono sviluppati da una comunità open source. Sono disponibili sotto la GNU General Public License.

3.4 DTrace

DTrace è un framework comprensivo di tracing dinamico creato da Sun Microsystems per risolvere problemi del kernel e applicazioni su sistemi di produzione in real time. Originariamente sviluppato per Solaris, è stato rilasciato sotto la libera Common Development and Distribution License (CDDL) ed è stato realizzato per molti altri sistemi Unix-like.
DTrace può essere usato per ottenere una visione globale di un sistema in esecuzione, come la quantità di memoria, il tempo di CPU, risorse di filesystem e di rete usati dai processi attivi. Può anche fornire molta più informazione a grana sottile, come il log degli argomenti con cui una specifica funzione è

chiamata, o una lista dei processi che accedono ad un file specifico.
DTrace è progettato per dare viste operazionali che permettono agli utenti di mettere a punto e risolvere problemi con applicazioni e il sistema operativo stesso.
I programmi di tracing (anche riferiti come script) sono scritti usando il linguaggio di programmazione D. Il linguaggio è un sottoinsieme del C con funzioni e variabili aggiunte specifiche per il tracing.
Considerazioni particolari sono state prese per rendere DTrace sicuro in modo da usarlo in un ambiente di produzione.

3.5 Commenti

LTTng si presenta come un insieme di molte patch da dover applicare al kernel. Quando si è tentato di installarlo si sono verificati i seguenti problemi:

- il patching è stato lungo e alcune patch non sempre si applicavano correttamente subito, bisognava risolvere la questione a livello di codice della patch stessa o del kernel;
- mancata risoluzione di dipendenze per aggiornare la libreria GTK (pacchetto LTTV non installato).

DTrace, invece, è solo stato oggetto di studio, non è stato provato perché la sua installazione non era possibile farla sull'ambiente di lavoro disponibile.
Per quando riguarda la scelta del tracing è chiaro che si opta per uno strumento (ftrace, appunto), visto che si dovranno trattare commutazioni e sveglie di task.

4 Ftrace

Il capitolo che segue vuole spiegare, con il giusto grado di dettaglio, questo strumento, di cui si è, già, accennato nei capitoli precedenti. Ftrace è stato uno dei software necessari per il lavoro di tesi, quindi, quella che si vuole fornire è una visione che comprende i seguenti punti:

- definizione dello strumento e come è strutturato;
- installazione;
- comandi di utilizzo ed esempi;
- file che compongono ftrace e altri file di interesse;
- analisi a basso livello di una parte di ftrace.

4.1 Che cos'è?

Ftrace è una soluzione di diagnostica contenuta all'interno del kernel (a partire dalla versione 2.6.27), che non richiede l'utilizzo di strumenti user space o particolare supporto. È utile per tenere traccia dei problemi, non solo nel kernel, ma anche nelle sue interazioni in spazio utente.
Il nome proviene da “function tracer”, che era il suo scopo originale, ma può fare molto di più. Un tracer consente, appunto, di tracciare eventi specifici in base al nome che lo descrive. Infatti, allo strumento sono stati aggiunti vari altri tracer per permettere di controllare le commutazioni di contesto, quanto a lungo gli interrupt rimangono disabilitati, quanto ci mettono i task ad alta priorità ad andare in esecuzione dopo che sono stati svegliati e così via. Ftrace, inoltre, include un framework che consente a nuovi tracer di essere aggiunti facilmente.

4.2 Installazione

Ftrace va configurato nel kernel in fase di setup del kernel stesso. Essendo uno strumento integrato appare tra le voci di configurazione: selezionando e salvando quelle opportune ftrace sarà presente nella macchina una volta compilato e installato il kernel. La configurazione del kernel è un'operazione basilare e delicata che serve a selezionare componenti hardware e software per il computer su cui si vuole installare il kernel: è necessario dedicarci tutto il tempo necessario perché un blocco in fase di avvio del sistema è molto spesso dovuto ad una non attenta configurazione.
Una volta sul sito www.kernel.org, si scarichi un vanilla Linux kernel a partire dalla versione 2.6.27. Il file che si scarica, di solito, è un file compresso, lo si decomprima all'interno di `/usr/src` e apparirà una cartella di nome `linux-2.6.xxx` a seconda della versione scaricata. Si apra un terminale e si scriva `cd /usr/src/linux-2.6.xxx`, si prema <Invio>, <Enter> e così si è all'interno della directory dei sorgenti. Poi si digiti `make menuconfig`. Sulla

“shell” saranno visualizzate le voci di configurazione del kernel in questione. Si entri su quella nominata `Kernel hacking`. Si deve selezionare (premere <y>) la voce `Kernel debugging`. Si scenda e si entri sul sottomenù denominato `Tracers`, al suo interno sono presenti i tracer. Qui a seconda dei test che si vogliono eseguire si selezioneranno alcuni piuttosto che altri. Ad esempio il tracer function corrisponde alla voce `Kernel Function Tracer`, sched_switch corrisponde a `Trace process context switches`. Si scelgono i tracer di interesse (sempre premendo <y> nella corrispondente voce). Una volta finito con ftrace si deve configurare il resto del kernel, attivando le voci di interesse. Fatto ciò si salva la configurazione. Adesso si è pronti per compilare il kernel. Digitare il comando `make` (sempre all'interno della cartella dei sorgenti). Attendere il tempo della compilazione. Quando ha finito, se non ci sono stati errori, si devono installare i moduli (voci configurate con <m> al posto di <y>), `make modules_install`. Dopodiché, se lo si ritiene necessario, si crea la “initrd” (immagine di ramdisk, per caricare il filesystem al boot del sistema) con il comando `mkinitramfs` (questa immagine va posizionata nella cartella `/boot`). Il comando `make install` provvederà ad installare il kernel. Ultimo passo, ma fondamentale se non è automatico: bisogna informare il boot loader (LILO o GRUB) della presenza del nuovo kernel, altrimenti al riavvio del computer non sarà possibile farlo partire perché non compare nelle voci rappresentanti i kernel avviabili. Con GRUB è sufficiente scrivere su terminale `update-grub`. Ora si può riavviare la macchina e scegliere il nuovo kernel compilato. Una volta partito, si può usare ftrace.

4.3 Tracer

Rappresentano i tipi di controlli che sono possibili con ftrace. Ogni tracciatore ha un suo compito e, in base alla volontà dell'utente, viene scelto uno piuttosto che un altro.
Sul kernel 2.6.29.4 (versione compatibile per l'installazione di RTAI) i vari tracer che si hanno a disposizione sono i seguenti:

- power: traccia le transizioni di stato della potenza del processore;
- function_graph: come il tracer function (vedere sotto), ma traccia anche le entrate e le uscite dalle funzioni, generando un grafo testuale;
- wakeup: traccia la massima latenza del task a priorità più alta da quando viene “svegliato” a quando viene schedulato
- irqsoff: analizza le funzioni mentre gli interrupt sono disabilitati;
- function: traccia ogni funzione chiamata dal kernel;
- sysprof: genera periodicamente dei trace dello stack per il processo corrente o per il thread del kernel;
- sched_switch: considera i context switches;
- initcall: traccia l'ingresso e l'uscita delle chiamate alla funzione di inizializzazione durante l'avvio del sistema;
- nop: non è un tracer, annulla tutti i tracing possibili.

4.4 Aspetti pratici

Per attivare ftrace da un terminale (shell) digitare:

```
# mkdir /debug
# mount -t debugfs nodev /debug
```

Si crea la cartella `debug` all'interno di `/` (root); questo comando si usa solo la prima volta che si avvia ftrace. Dopo si “monta” (abilita) debugfs; questo comando, invece, va sempre digitato, altrimenti non sarà possibile usare lo strumento di analisi.
Ftrace viene usato, quindi, tramite il file system “debugfs”. Appena eseguiti questi due comandi all'interno della cartella (directory) `debug` saranno presenti delle nuove cartelle. In particolare si dovrà entrare sulla directory `tracing`; qui sono disponibili i file di controllo e di output di ftrace.
Seguono i file più importanti:

Nome del file	Descrizione
`current_tracer`	contiene il nome del tracer corrente che deve essere usato per fare un certo tipo di analisi
`available_tracers`	contiene i vari tracer che sono stati configurati nel kernel (la lista elencata sopra)
`tracing_enabled`	questo file mostra se il tracciatore è attivato per il tracing (1) oppure è disabilitato (0)
`trace`	visualizza l'uscita del tracing in formato leggibile
`latency_trace`	come il file `trace`, ma le informazioni sono organizzate in modo da mostrare le possibili latenze nel sistema
`trace_pipe`	l'output è lo stesso del file `trace`, ma questo file si intende consumarlo con "tracing vivo". Le letture da questo file si bloccheranno finché nuovi dati non sono recuperati
`trace_options`	permette all'utente di controllare la quantità di dati visualizzati sui file di output sopra citati
`trace_max_latency`	alcuni tracer memorizzano la massima latenza
`buffer_size_kb`	questo file imposta la dimensione (in kilobyte) del buffer di ogni processore. I buffer di ogni processore hanno la stessa dimensione
`tracing_cpumask`	questa è una maschera che permette all'utente di fare il trace su specifici processori (il formato è una stringa esadecimale rappresentante le CPU)
`set_ftrace_filter`	viene limitato il tracing alle sole funzioni elencate in questo file
`set_ftrace_notrace`	le funzioni qui inserite non saranno tracciate
`set_ftrace_pid`	Il tracer function traccia un singolo thread
`set_graph_function`	il file imposta una funzione "trigger" dove il tracing dovrebbe iniziare con il tracer function_graph
`available_filter_functions`	il file contiene la lista di funzioni che ftrace ha elaborato e può tracciare.

	Queste sono le funzioni che possono essere scritte su `set_ftrace_filter` o su `set_ftrace_notrace`

4.5 Esempi di utilizzo di tracer

L'utilizzo di ftrace avviene tramite terminale. Per semplificare il lavoro si può creare uno “script” che poi sarà lanciato eseguendo il lavoro richiesto. Uno “script” è un file contenente comandi che vengono scritti sulla shell, ha estensione “.sh” e si esegue, dopo eventualmente avergli cambiato i permessi, con `/percorso_del_file/file_script.sh` e premendo <Invio>.
Viene adesso discusso l'uso di un determinato tracer su processi Linux. I comandi per l'attivazione sono:

```
# echo function > /debug/tracing/current_tracer
# echo 1 > /debug/tracing/tracing_enabled
```

il primo comando seleziona il tracer function, il secondo lo abilita al tracciamento.
Poi si eseguirà:

```
# echo 0 > /debug/tracing/tracing_enabled
# cat /debug/tracing/trace
```

si disabilita il tracciamento e poi si stampa il file risultato del tracing a terminale.
Viene illustrato un pezzo del file `trace` ottenuto eseguendo i comandi sopra citati:

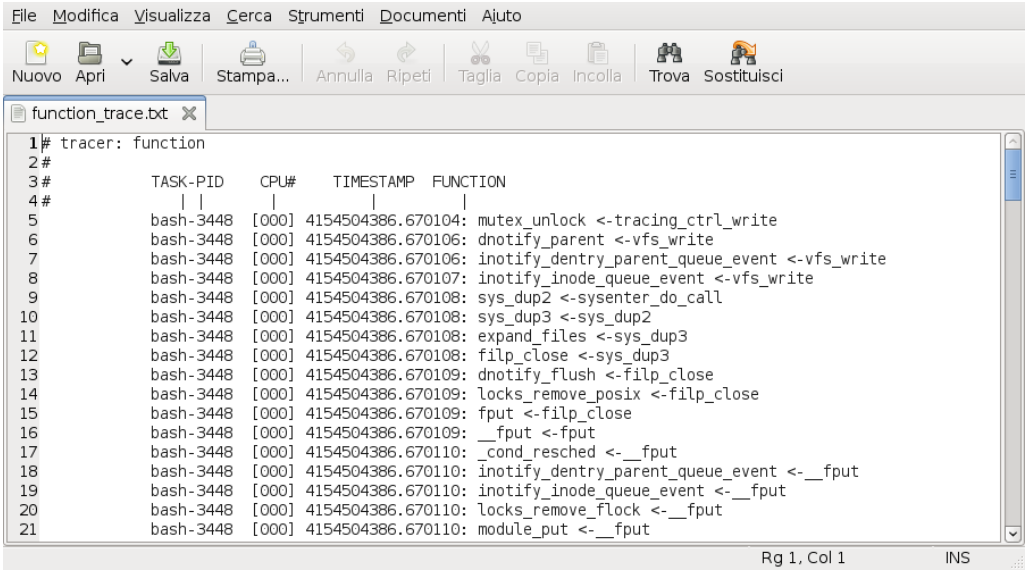

Illustrazione 1: Tracciato del tracer function (file `trace`)

Si nota il nome del tracer (function) e sotto segue una sorta di tabella con indicato il task con il relativo pid (process identifier, identificativo di processo) che sta eseguendo, il processore, il timestamp (formato <secondi>.<microsecondi>), la funzione che sta venendo tracciata e la sua funzione chiamante. Il timestamp indica l'istante di tempo in cui la funzione ha iniziato l'esecuzione.
Un altro tracer, di cui si farà uso nella tesi, è sched_switch. Si attiva con gli stessi comandi visti appena sopra, sostituendo `function` con `sched_switch`. Un estratto del suo operato è il seguente:

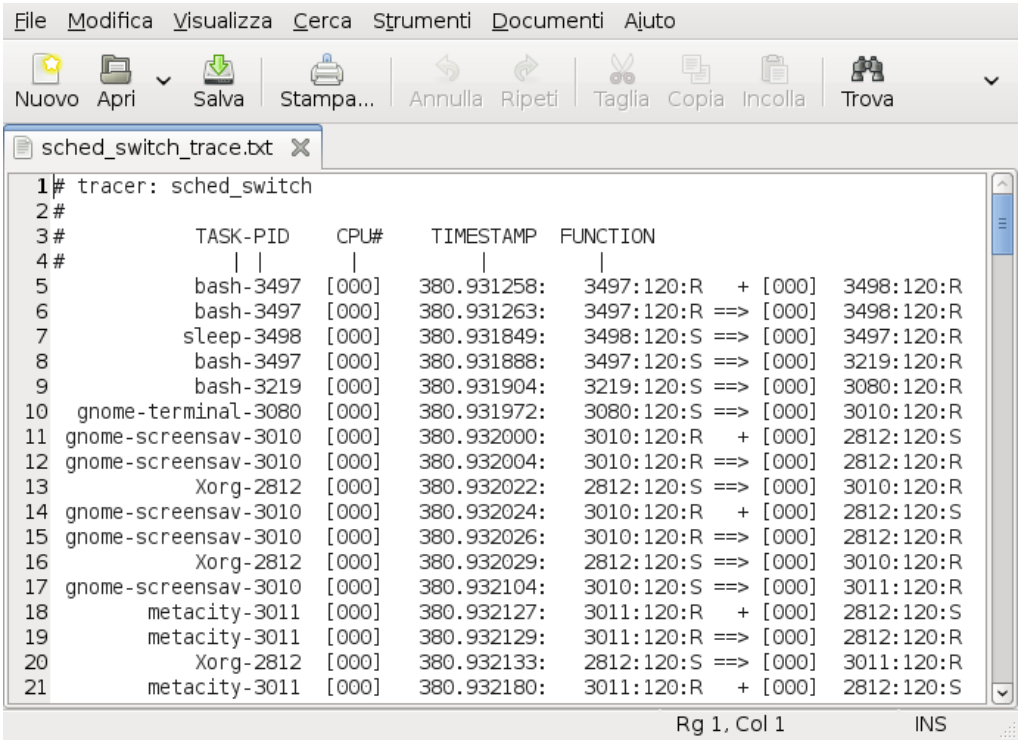

Illustrazione 2: Tracciato del tracer sched_switch (file `trace`)

Il tracer sched_switch, anche, include il tracciamento delle sveglie dei task, oltre alle commutazioni di contesto.
Le sveglie sono rappresentate da un + e le commutazioni di contesto sono mostrate da ==>. Il formato è:

Operazione	**Descrizione**
context switch	task precedente task successivo <pid>:<prio>:<state> ==> <pid>:<prio>:<state>
sveglie	task corrente task che si è svegliato <pid>:<prio>:<state> + <pid>:<prio>:<state>

prio è la priorità del kernel interna, inversa della priorità che è, di solito, visualizzata da strumenti user space.
La testata è uguale a quella del tracer precedente solo che `FUNCTION` è un termine improprio, poiché esso rappresenta le sveglie e le commutazioni.

Le cifre che appaiono tra parentesi quadre (`[000]`) dopo il simbolo `+` o `==>` indicano la CPU dove si è svegliato o dove andrà in esecuzione il processo scritto subito dopo. La prima riga del file nell'immagine significa che mentre è in esecuzione il task `bash` (processo corrente) nella CPU 000, si sveglia il processo con pid 3498, sempre, sulla CPU 000. La seconda riga, invece, è la commutazione da `bash` al task `sleep`, pid 3498 (il nuovo task corrente) nel processore 000.
Gli stati dei task sono:

- `R` – running: vuole eseguirsi, oppure può non essere, realmente, in esecuzione
- `S` – sleep: il processo sta aspettando di essere svegliato
- `D` – disk sleep: il processo deve essere svegliato
- `T` – stopped: processo sospeso
- `t` – traced: il processo sta venendo tracciato (con qualcosa simile al programma gdb)
- `Z` – zombie: il processo è in attesa di essere ripulito
- `X` – sconosciuto

Per visualizzare, invece, dei tempi di latenza si deve far ricorso al file `latency_trace`. Le informazioni in esso racchiuse servono a spiegare perché la latenza accade. Un esempio (sempre processi Linux):

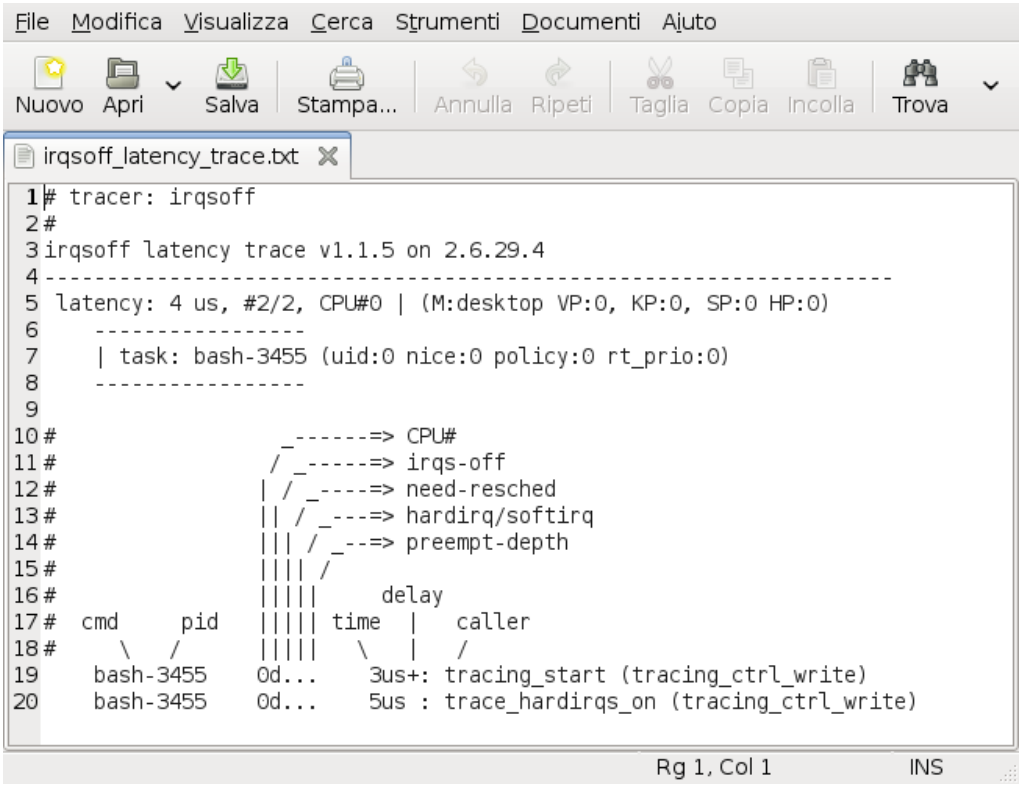

Illustrazione 3: Tracciato del tracer irqsoff (file `latency_trace`)

Come si nota il tracer sotto esame è irqsoff. Mostra:

- la latenza massima in microsecondi (4 µs) da quando gli interrupt sono disattivati a quando sono riattivati;

- il tipo di preemption usato (`DESKTOP`, quello che è stato scelto in fase di configurazione del kernel);
- `VP`, `KP`, `SP` e `HP` sono sempre a zero.

La voce `task` corrisponde al processo che stava venendo eseguito quando la latenza è accaduta (`bash`, ovvero il terminale). La funzione che ha causato sia la disabilitazione degli interrupt che la successiva riabilitazione è la stessa, ossia `tracing_ctrl_write`. Segue la spiegazione del resto del file:

- `cmd`, il nome del processo coinvolto nel trace;
- `pid`, l'identificativo di quel processo;
- `CPU#`, il processore nel quale stava venendo eseguito il processo;
- `irqs-off`, se è presente `d` gli interrupt sono disabilitati, se invece c'è `.` sono abilitati;
- `need-resched`, se `N` il task `need_resched` è impostato, se `.` non impostato;
- `hardirq/softirq`, `H` un'hard irq è accaduta all'interno di una soft irq, `h`, un'hard irq sta venendo eseguita, `s` una soft irq sta venendo eseguita, `.` contesto normale;
- `preempt-depth`, il livello di preempt_disabled (se preemption è stata disabilitata);
- `time`, timestamp relativo all'inizio del trace in microsecondi;
- `delay`, `!` più grande di preempt_mark_thresh (il default è 100), `+` più grande di un microsecondo, “ ” non più di un microsecondo.

4.6 Tracer fondamentali

Come già accennato, ftrace mette a disposizione diversi tracer ognuno con uno specifico campo di utilizzo: dal tracciamento delle commutazioni di contesto, al tracciamento delle funzioni del kernel, alla misura della massima latenza (dopo aver disabilitato gli interrupt) ecc.
Vediamo, quindi, i principali tracer presenti:

Function

Il suo utilizzo si rende necessario quando c'è bisogno di debuggare un particolare comportamento del kernel e si vuole analizzare passo passo ogni singola funzione eseguita. Importante è avere abilitato l'opzione “ftrace_enabled” in modo da, effettivamente, registrare le funzioni attraversate, altrimenti il tracer risulta un “nop”, ovvero non scrive nessun tracing.

Sched_switch

Memorizzare gli switch di contesto e le riattivazioni dei processi nelle situazioni dove è opportuno sapere, primo, quali task passano per la CPU e, secondo, istante per istante, conoscere il loro stato. Questo è l'obiettivo di questo importante tracciatore.

Irqsoff

Quando le interruzioni sono disabilitate il processore è isolato dagli eventi esterni e non può reagire; quello che succede, quindi, è un ritardo di reazione, o meglio una latenza. Questo tracer interviene nei casi in cui si vuole conoscere la massima latenza da quando gli interrupt sono disabilitati a quando ritornano abilitati.

Wakeup

In un ambiente real time è bene conoscere il tempo di risveglio ("wakeup") per il task a priorità più alta da quando viene, appunto, svegliato a quando va in esecuzione. Questo tempo viene, anche, nominato "latenza di schedulazione". In particolare si è interessati alla latenza nel caso peggiore (ritardo più lungo prima che qualcosa accada). Se si vuole sapere qual'è il tempo di wakeup bisogna far agire questo tracer.

4.7 File del kernel di interesse

Questo paragrafo vuole elencare alcuni file del kernel ritenuti importanti per questa tesi. Sono presenti due sezioni: nella prima ci sono i file che definiscono ftrace e nella seconda alcuni file contenenti funzioni di cui si è fatto uso più avanti.

4.7.1 File che compongono ftrace

Sono elencati i file sorgenti del kernel che costituiscono ftrace (anche qui certe funzioni all'interno di alcuni di questi file saranno utilizzate successivamente).
Viene presentata una tabella con due colonne: nella prima c'è il nome del file e nell'altra c'è il percorso di dove risiede il file (con `TREE` si intende la cartella contenente l'albero del kernel, ad esempio `/usr/src/linux-2.x.xx`).

Nome del file	**Percorso del file**
`ftrace.c`	`TREE/kernel/trace`
`ring_buffer.c`	`TREE/kernel/trace`
`trace.c`	`TREE/kernel/trace`
`trace.h`	`TREE/kernel/trace`
`trace_boot.c`	`TREE/kernel/trace`
`trace_branch.c`	`TREE/kernel/trace`
`trace_functions.c`	`TREE/kernel/trace`
`trace_functions_graph.c`	`TREE/kernel/trace`
`trace_hw_branches.c`	`TREE/kernel/trace`
`trace_irqsoff.c`	`TREE/kernel/trace`

trace_mmiotrace.c	TREE/kernel/trace
trace_nop.c	TREE/kernel/trace
trace_power.c	TREE/kernel/trace
trace_sched_switch.c	TREE/kernel/trace
trace_sched_wakeup.c	TREE/kernel/trace
trace_selftest.c	TREE/kernel/trace
trace_selftest_dynamic.c	TREE/kernel/trace
trace_stack.c	TREE/kernel/trace
trace_sysprof.c	TREE/kernel/trace
block.h	TREE/include/trace
boot.h	TREE/include/trace
sched.h	TREE/include/trace
ftrace.h	TREE/include/linux
ftrace_irq.h	TREE/include/linux
ipipe_trace.h	TREE/include/linux
mmiotrace.h	TREE/include/linux
stacktrace.h	TREE/include/linux
tracepoint.h	TREE/include/linux
tracepoint.c	TREE/kernel
tracer.c	TREE/kernel/ipipe

4.7.2 Altri file

Ora sono riportati i file contenenti funzioni e strutture dati spiegate in seguito. La tabella sotto è organizzata come la precedente.

Nome del file	**Percorso del file**
inode.c	TREE/fs/debugfs
fs.h	TREE/include/linux
ring_buffer.h	TREE/include/linux
seq_file.c	TREE/fs
seq_file.h	TREE/include/linux
sched.c	TREE/kernel

4.8 Analisi a basso livello

La tesi ha come obiettivo la creazione di un applicativo per il tracing di thread e processi Linux e task RTAI; ciò comporta l'estensione di ftrace al domino di RTAI (co-scheduler di Linux). Si vuole tener traccia nel tempo di tutte le commutazioni di contesto che avvengono tra i diversi tipi di task/thread/processi hard, soft e non real time per avere un rapporto completo di cosa accade nel sistema.
Per fare ciò occorre capire come funziona ftrace, o meglio il tracer `sched_switch` a basso livello. Partendo dal file sorgente `trace_sched_switch.c` sono state analizzate le funzioni principali, estendendosi, anche, ad altri file, cercando di acquisire i concetti su cui si fondavano. Inserendo, in alcune funzioni, in ingresso e in uscita, delle stringhe di debug è stato possibile individuare quando queste vengono chiamate, a che punto dell'esecuzione di ftrace con attivo `sched_switch`; in altri casi si è reso necessario far stampare dei valori di variabili interessandosi, così, ai dati che passavano per lo strumento. Il kernel, quindi, è stato modificato e per visualizzare quelle stringhe si è dovuto ricompilarlo e reinstallarlo.
L'analisi ha richiesto un'attenta e ripetuta navigazione dei sorgenti in linguaggio C per il continuo uso di macro e l'esportazione di molti simboli e funzioni: sono passato da `sched_switch` al mondo dei filesystem, alle scritture e letture del ring buffer per arrivare alle chiamate del tracer da parte dello scheduler.
Nel seguito verranno analizzati i seguenti punti:

1. funzioni principali per il tracer `sched_switch`: registrazione di un context switch e registrazione di una sveglia;
2. filesystem debugfs: come viene creata la directory `tracing` e i suoi file all'interno;
3. ring buffer, oggetto di memorizzazione di ftrace: scrittura e lettura.

Saranno, anche, mostrati e commentati segmenti di codice ritenuti d'interesse.

4.8.1 Funzioni principali

`trace_sched_switch.c` contiene la base per `sched_switch`, le operazioni necessarie alla sua attivazione/disattivazione e utilizzo.
Si restringe il suo studio a due sole funzioni. In particolare queste riguardano il tracciamento delle attività per cui bisogna inserire questo tracer: commutazioni di contesto e sveglie di processi.
Le altre trattano l'abilitazione e disabilitazione di `sched_switch`. L'albero (sequenza) delle chiamate alle funzioni quando si imposta questo tracer è il seguente:

```
sched_switch_trace_init->start_sched_trace-
>tracing_start_sched_switch_record->tracing_start_sched_switch-
>tracing_sched_register
```

Quest'ultima effettua tre registrazioni, o meglio connessioni: vengono connessi dei “tracepoint”, identificati da nomi, a dei “probe”, identificati da funzioni.
Un tracepoint fornisce un aggancio per chiamare una funzione che può essere decisa runtime, quindi, “chiamando” quel nome viene eseguita la funzione ad esso associata.

Nel nostro caso i tre tracepoint sono collegati alle sveglie e agli switch e sono:

1. *sched_wakeup*;
2. *sched_wakeup_new*;
3. *sched_switch*.

I primi due sono connessi alla stessa funzione (probe) e considerano una sveglia, invece, il terzo segnala una commutazione di contesto. Questi probe sono le due funzioni enunciate all'inizio del paragrafo: `probe_sched_wakeup` e `probe_sched_switch`.
L'albero contrario, quando si disinserisce sched_switch è, così, composto:
`sched_switch_trace_reset->stop_sched_trace->tracing_stop_sched_switch_record->tracing_stop_sched_switch->tracing_sched_unregister`
`tracing_sched_unregister` scollega i tracepoint dai probe.
Veniamo ora a queste due funzioni.
`probe_sched_switch` (connessa a *sched_switch*) registra, appunto, uno switch di task. Segue il codice:

```
1.  static void
2.  probe_sched_switch(struct rq *__rq, struct task_struct
    *prev,
3.                     struct task_struct *next)
4.  {
5.        struct trace_array_cpu *data;
6.        unsigned long flags;
7.        int cpu;
8.        int pc;
9.
10.       if (!sched_ref)
11.             return;
12.
13.       tracing_record_cmdline(prev);
14.       tracing_record_cmdline(next);
15.
16.       if (!tracer_enabled)
17.             return;
18.
19.       pc = preempt_count();
20.       local_irq_save(flags);
21.       cpu = raw_smp_processor_id();
22.       data = ctx_trace->data[cpu];
23.
24.       if (likely(!atomic_read(&data->disabled)))
25.             tracing_sched_switch_trace(ctx_trace,    data,
    prev, next, flags, pc);
26.
27.       local_irq_restore(flags);
28. }
```

Parametri d'ingresso:

- `__rq`: “run queue” (lista di tutti i processi in attesa di essere eseguiti su

un determinato processore) dove sono presenti i task da cambiare;
- `prev`: indirizzo del task corrente che viene sostituito;
- `next`: puntatore al task sostituto.

Dichiarazione variabili dalla riga 5 alla 8: `data` contiene informazioni per tracciare lo switch, `flags` salva i flag degli interrupt, `cpu` serve come indice del processore corrente e `pc` conta le preemption.
Se la variabile `sched_ref` è nulla si esce dalla funzione (righe 10 e 11).
Vengono salvati i nomi e i pid dei task coinvolti nello switch (righe 13 e 14).
Se il tracer sched_switch non è stato abilitato si esce dalla funzione (righe 16 e 17).
Viene contato quante volte il thread corrente ha subito preemption, poi sono disabilitati gli interrupt con salvataggio flag, è calcolato l'indice della CPU corrente e sono salvati i dati di lavoro per quel processore (dalla riga 19 alla 22).
Se su questi dati il flag `disabled` è nullo, allora viene eseguito il vero e proprio tracciamento dello switch (righe 24 e 25).
Chiude la funzione la riabilitazione degli interrupt con ripristino dei flag (riga 27).

La parte di `tracing_sched_switch_trace` di interesse è la seguente:

```
1. tracing_generic_entry_update(&entry->ent, flags, pc);
2. entry->ent.type              = TRACE_CTX;
3. entry->prev_pid              = prev->pid;
4. entry->prev_prio       = prev->prio;
5. entry->prev_state      = prev->state;
6. entry->next_pid              = next->pid;
7. entry->next_prio       = next->prio;
8. entry->next_state      = next->state;
9. entry->next_cpu   = task_cpu(next);
```

`entry` è un puntatore alla struct `ctx_switch_entry` la quale memorizza gli switch e le sveglie.
`tracing_generic_entry_update` inserisce alcune informazioni in `entry->ent` come, ad esempio, il pid del task corrente.
Dopo viene salvato il tipo di operazione (in questo caso commutazione) e, quindi, segue la registrazione sul ring buffer di pid, priorità e stato dei processi corrente e successivo. Il campo `next_cpu` segnala l'indice del processore su cui andrà in esecuzione `next`. Questa è l'essenza dello switch tra processi segnato da ftrace.

La seconda è `probe_sched_wakeup` (connessa a *sched_wakeup* e *sched_wakeup_new*) e traccia una sveglia. Codice:

```
1. static void
2. probe_sched_wakeup(struct rq *__rq, struct task_struct
   *wakee, int success)
3. {
4.       struct trace_array_cpu *data;
5.       unsigned long flags;
```

```
6.        int cpu, pc;
7.
8.        if (!likely(tracer_enabled))
9.              return;
10.
11.       pc = preempt_count();
12.       tracing_record_cmdline(current);
13.
14.       local_irq_save(flags);
15.       cpu = raw_smp_processor_id();
16.       data = ctx_trace->data[cpu];
17.
18.       if (likely(!atomic_read(&data->disabled)))
19.             tracing_sched_wakeup_trace(ctx_trace,    data,
    wakee, current,
20.                                        flags, pc);
21.
22.       local_irq_restore(flags);
23. }
```

Molto simile alla precedente, il segmento che riguarda la sveglia è contenuto in `tracing_sched_wakeup_trace`:

```
1.  tracing_generic_entry_update(&entry->ent, flags, pc);
2.  entry->ent.type              = TRACE_WAKE;
3.  entry->prev_pid              = curr->pid;
4.  entry->prev_prio        = curr->prio;
5.  entry->prev_state       = curr->state;
6.  entry->next_pid              = wakee->pid;
7.  entry->next_prio        = wakee->prio;
8.  entry->next_state       = wakee->state;
9.  entry->next_cpu              = task_cpu(wakee);
```

`tracing_generic_entry_update` funziona esattamente allo stesso modo come in `probe_sched_switch`.
Il tipo di operazione salvato è una sveglia e, qui, vengono memorizzati, sempre sul ring buffer, il pid, la priorità e lo stato del task corrente (quello in esecuzione) e del processo `wakee`, cioè quello che si è appena svegliato. `next_cpu` ha il significato di prima.

4.8.2 Debugfs

A volte, per gli sviluppatori, si rende necessario avere un riscontro di quello che si sta facendo, per vedere se tutto funziona correttamente, o se ci sono degli errori. Questo riscontro va sotto il nome di “informazioni di debug”. L'azione di debug viene, appunto, svolta per far capire allo sviluppatore come il sistema si comporta quando viene eseguita un'applicazione. Le informazioni, così, prodotte non sono indispensabili per il funzionamento del sistema, ma

danno solo un supporto al programmatore.
Il debugging può essere fatto sia in user space (per applicazioni utente) sia in kernel space (per il kernel e i moduli) in modalità tra loro diverse.
In certi casi, per il debugging in kernel space (non sempre così agevole), può essere usata la funzione `printk ()`, che stampa l'informazione di interesse nel log del kernel: ad esempio, se si vuole monitorare una variabile, con la funzione `printk ()` si può stampare, nel file di log, il suo valore in ogni punto dell'esecuzione ritenuto critico per quella variabile, così si vede il suo andamento e si capisce se assume valori corretti o anomali. È importante notare che i messaggi di controllo che si vuole far stampare devono essere costantemente sorvegliati con il comando `dmesg`, ossia se `printk ()` viene inserita in una funzione che il kernel chiama molto spesso, questi messaggi "inonderanno" i file di log facendoli crescere di dimensione. Di conseguenza si riduce lo spazio su disco e continuamente, visto che i messaggi sono stampati sempre. La soluzione di ciò consiste nello "svuotare" i file di log, una volta che si ha capito quello che si doveva controllare, i quali ricominceranno a riempirsi. A parte questa attenzione da tenere, l'uso di `printk ()` è utile per fare debugging, soprattutto nel kernel per capire determinate sue parti, o nei moduli.
In altre situazioni `printk ()` non è sufficiente, soprattutto se si vuole far cambiare dei valori da spazio utente all'applicazione.
Una via per rendere disponibili informazioni di debug (anche con accesso in scrittura) è di creare uno o più file in un file system virtuale. Ci sono un po' di modi per fare ciò:

- creare file all'interno di `/proc`; questo approccio funziona, ma le funzioni del file system `/proc` non sono molto semplici da usare;
- nei kernel 2.6 è presente il file system `/sys` (sysfs); alcune informazioni di debug possono essere inserite lì, ma il suo utilizzo principale è per scopi amministrativi e le sue regole richiedono che ogni file contenga un singolo valore; per questa ragione, non è possibile usare l'interfaccia `seq_file` (spiegata successivamente) con sysfs; il risultato è che sysfs non è molto adatto per esempio se si desidera salvare una struttura dati complicata;
- creare, interamente, un nuovo file system con libfs; soluzione altamente flessibile, ma per un programmatore va oltre il fatto di rendere disponibili delle informazioni di debug.

Questi tre modi possono, quindi, essere usati dagli sviluppatori, ma non sono ottimali.
Greg Kroah-Hartman ha creato debugfs, un file system devoto alle informazioni di debug.
Debugfs è inteso come un sottosistema relativamente facile e leggero che scompare una volta tolto dalla configurazione del kernel. Con questo file system è possibile gestire tutte le informazioni che devono essere prodotte, salvandole in file opportunamente creati.
Esistono delle funzioni particolari che devono essere usate con debugfs per poter creare file e, anche, directory.
La creazione di un file si effettua con:

```
struct dentry *debugfs_create_file(const char *name, mode_t
mode, struct dentry *parent, void *data, const struct
```

```
file_operations *fops)
```

I parametri di ingresso sono:
•`const char *name`: il nome del file rappresentato da un puntatore a caratteri;
•`mode_t mode`: i permessi del file, identificati da un numero a quattro cifre decimali (comprese tra 0 e 7), il primo è zero, gli altri vanno convertiti in una sequenza di tre bit. La prima sequenza di bit rappresenta i permessi del proprietario del file, la seconda, i permessi degli altri utenti appartenenti al gruppo del proprietario e la terza, i permessi di tutti gli altri utenti. Lettura, scrittura ed esecuzione sono i permessi ordinati, 1 in corrispondenza di un permesso rappresenta il suo possesso, 0 altrimenti;
•`struct dentry *parent`: puntatore alla struttura dati che identifica la cartella contenente il file da creare, la cartella genitore; se questo valore è `NULL` significa che il file viene creato all'interno della directory principale (directory root) del file system, ovvero la cartella dove è stato montato debugfs;
•`void *data`: i dati relativi al file;
•`const struct file_operations *fops`: struttura dati che contiene gli indirizzi delle funzioni che devono gestire il file che si sta creando, apertura, lettura, rilascio, ecc.
Il parametro di ritorno di questa funzione è un puntatore a struct `dentry`, struttura che rappresenta, in un file system, un file (stesso tipo di struct del parametro `parent`).
`NULL`, come valore di ritorno, fa capire che sono successi degli errori e il file non è stato creato.
Una cartella, all'interno di debugfs, viene realizzata sfruttando:

```
struct dentry *debugfs_create_dir(const char *name, struct
dentry *parent)
```

Parametri:
•`const char *name`: il nome della cartella che si vuole creare;
•`struct dentry *parent`: rappresenta la cartella genitore, `NULL` per crearla nella root.
Anche per questa funzione, quello che viene ritornato è un puntatore ad una struttura `dentry` che rappresenta, in questo caso, una cartella.
Se debugfs non è presente nel kernel (manca nella configurazione) e si cerca di creare una directory verrà ritornato l'errore `-ENODEV` (No such device). Se, invece, è `NULL` il valore di ritorno significa che nell'esecuzione della funzione si sono verificati altri tipi di errore. In queste situazioni la directory non è creata.
Debugfs, quando viene “smontato”, non cancella automaticamente quello che è stato creato, quindi, per ogni directory e/o file creato, ci deve essere una chiamata alla funzione di rimozione che, per questo file system, è:

```
void debugfs_remove(struct dentry *dentry)
```

Il puntatore a struct `dentry` rappresenta l'elemento da eliminare.
Questa descrizione si è resa necessaria per il semplice fatto che ftrace usa questo file system per tenere i suoi file di controllo e i file che vengono visualizzati in output, come ad esempio il file `trace`. La cartella `tracing`,

già citata in precedenza, viene creata con la funzione spiegata prima, ovvero `debugfs_create_dir` e ogni file al suo interno con `debugfs_create_file`, in cui vengono passati come parametri i permessi, la struttura che contiene gli indirizzi delle funzioni di gestione, ecc.

Creazione della cartella `tracing`

Il codice sotto serve a mostrare come, all'interno del kernel, ftrace usa il filesystem debugfs:

```
1.  static struct dentry *d_tracer;
2.
3.  struct dentry *tracing_init_dentry(void)
4.  {
5.        static int once;
6.
7.        if (d_tracer)
8.              return d_tracer;
9.
10.       d_tracer = debugfs_create_dir("tracing", NULL);
11.
12.       if (!d_tracer && !once) {
13.             once = 1;
14.             pr_warning("Could    not    create    debugfs
    directory 'tracing'\n");
15.             return NULL;
16.       }
17.
18.       return d_tracer;
19. }
```

La funzione ha il compito di creare la directory `tracing`, base per i file di ftrace. Ritorna il puntatore ad una “entry” di debugfs e non vuole nessun parametro in ingresso.
La variabile `d_tracer`, che contiene il riferimento alla cartella, è dichiarata all'esterno (riga 1).
L'altra variabile, `once`, dice se è la prima volta che viene eseguita questa funzione (riga 5).
Se `d_tracer` non è `NULL` viene tornato così (righe 7 e 8).
Creazione directory (riga 10).
Se `d_tracer` ha valore `NULL` ed è la prima volta che si crea la cartella si scrive, nel log del kernel, un messaggio di warning e si ritorna `NULL` (dalla riga 12 alla 16).
Se le condizioni sopra non sono verificate, si ritorna `d_tracer` (riga 18).

Segue ora l'utilizzo di `debugfs_create_file`. È riportata la creazione di un solo file:

```
1.  struct dentry *d_tracer;
```

```
2. struct dentry *entry;
3.
4. d_tracer = tracing_init_dentry();
5.
6. entry = debugfs_create_file("tracing_enabled", 0644,
   d_tracer,
   &global_trace, &tracing_ctrl_fops);
7. if (!entry)
8. pr_warning("Could not create debugfs 'tracing_enabled'
   entry\n");
```

Il codice è preso dalla funzione `tracer_init_debugfs`.
Dichiarazione variabili (righe 1 e 2): `d_tracer` contiene l'indirizzo della dentry della cartella `tracing` ed `entry` serve per la creazione dei file.
Creazione directory `tracing` con la funzione di prima (riga 4).
Creazione del file `tracing_enabled` (riga 6).
Se `entry` è `NULL` si stampa, nei file di log, una warning (righe 7 e 8).

Questi segmenti volevano far notare come viene usato debugfs.

Struct `file_operations`

I file di ftrace, in qualche modo, devono poter essere controllati dal kernel permettendo all'utente di guardare il contenuto (informazioni) di alcuni file e scrivere all'interno di altri (passaggio di parametri). Operando così si ottiene una sorta di collaborazione utente-kernel, o meglio sviluppatore-ftrace, fondamentale per analizzare particolari dinamiche.
Per questo scopo viene in aiuto una struttura dati molto importante, mostrata di seguito, senza la quale le cose non sarebbero altrettanto governabili.

```
struct file_operations {
      struct module *owner;
      loff_t (*llseek) (struct file *, loff_t, int);
      ssize_t (*read) (struct file *, char __user *, size_t,
loff_t *);
      ssize_t (*write) (struct file *, const char __user *,
size_t, loff_t *);
      ssize_t (*aio_read) (struct kiocb *, const struct iovec
*, unsigned long, loff_t);
      ssize_t (*aio_write) (struct kiocb *, const struct iovec
*, unsigned long, loff_t);
      int (*readdir) (struct file *, void *, filldir_t);
      unsigned int (*poll) (struct file *, struct
poll_table_struct *);
      int (*ioctl) (struct inode *, struct file *, unsigned
int, unsigned long);
      long (*unlocked_ioctl) (struct file *, unsigned int,
unsigned long);
      long (*compat_ioctl) (struct file *, unsigned int,
unsigned long);
      int (*mmap) (struct file *, struct vm_area_struct *);
      int (*open) (struct inode *, struct file *);
      int (*flush) (struct file *, fl_owner_t id);
      int (*release) (struct inode *, struct file *);
```

```
      int (*fsync) (struct file *, struct dentry *, int
datasync);
      int (*aio_fsync) (struct kiocb *, int datasync);
      int (*fasync) (int, struct file *, int);
      int (*lock) (struct file *, int, struct file_lock *);
      ssize_t (*sendpage) (struct file *, struct page *, int,
size_t, loff_t *, int);
      unsigned long (*get_unmapped_area)(struct file *,
unsigned long, unsigned long, unsigned long, unsigned long);
      int (*check_flags)(int);
      int (*flock) (struct file *, int, struct file_lock *);
      ssize_t (*splice_write)(struct pipe_inode_info *, struct
file *, loff_t *, size_t, unsigned int);
      ssize_t (*splice_read)(struct file *, loff_t *, struct
pipe_inode_info *, size_t, unsigned int);
      int (*setlease)(struct file *, long, struct file_lock
**);
};
```

Questa struttura dati, definita all'interno del file `fs.h`, permette di gestire un file creato nel file system debugfs.
In generale una struttura (struct) del linguaggio C è caratterizzata da un nome e da un insieme di campi che non sono altro che variabili precedute dal tipo o prototipi di funzioni. Una possibile assegnazione di valori a queste variabili e prototipi rappresenta un'istanza della struct.
Nel codice, quindi, `file_operations` è il nome della struct e quelli che seguono sono i campi.
Si vede che i campi sono diversi e non necessariamente devono essere riempiti tutti, alcuni possono restare vuoti. Di solito i campi classici usati sono:

- `open`: contiene il puntatore alla funzione d'apertura, realizzata ad hoc per quel file;
- `read`: questo indirizzo punta ad una funzione ad hoc oppure alla `seq_read` (funzione usata con i “seq_file”).

Poi, ma non sempre usati, ci sono `llseek`, `write` e `release`. Il primo serve per spostarsi all'interno del file in una posizione indicata, il secondo punta ad una funzione di scrittura e il terzo, invece, rilascia il file, operazione che avviene quando si ha finito di lavorarci.
In debugfs ogni file ha la sua struttura `file_operations`, diversa da quella degli altri, che punta alle corrette funzioni implementate per quel file.
Nella pratica il funzionamento è questo, quando si vuole visualizzare un file all'interno di `tracing`:

1. usando il comando `cat`, viene da prima chiamata la funzione puntata da `open`, la quale predispone il file per poter essere letto;
2. segue la chiamata alla `read` che effettua la lettura vera e propria;
3. infine, prima di uscire, entra in gioco la `release` che rilascia il file.

All'interno di `tracing` conviene usare il comando `cat` per visualizzare un file, perché altri comandi potrebbero causare dei problemi.

Interfaccia `seq_file`

Per ftrace l'utilizzo della struct `seq_file` è fondamentale. Il suo scopo è quello di riuscire a creare un file da sequenze di record. Infatti, come sarà spiegato

dopo, ftrace memorizza le sue registrazioni (record appunto) in una struttura detta ring buffer, quindi, nel processo di visualizzazione vengono letti record e per poterli organizzare in un file bisogna ricorrere a questa interfaccia.
Un record contiene informazioni diverse a seconda del tracer che viene usato. Nel caso `sched_switch` segnala il processo corrente seguito da una commutazione di contesto tra esso ed un altro o una sveglia di un altro task. Una sequenza scritta di questi record è il tracing degli switch e delle sveglie prodotto da ftrace.
Per gestire i record è necessario un iteratore che permette di scorrerci, prendere il successivo, mostrarlo e quando ha finito fermarsi. Esiste una struct particolare che permette proprio questo: mette a disposizione dei campi (prototipi di funzioni) per la vita dell'iteratore.
Le funzioni vanno costruite dall'utente, perché è lui stesso che conosce cosa deve essere gestito e su che file. Fatto questo, gli indirizzi vanno inseriti in una istanza di questa struct particolare che poi sarà associata al seq_file. La struttura si chiama `seq_operations`, presente nel file `seq_file.h` ed è qui riportata:

```
struct seq_operations {
      void * (*start) (struct seq_file *m, loff_t *pos);
      void (*stop) (struct seq_file *m, void *v);
      void * (*next) (struct seq_file *m, void *v, loff_t
*pos);
      int (*show) (struct seq_file *m, void *v);
};
```

- `start`: questo campo serve ad iniziare l'iterazione;
- `next`: trova il record successivo;
- `stop`: ferma l'iteratore;
- `show`: stampa il record.

Esempio preso dal codice di ftrace

Viene, qui, riportato del codice per dare un'idea di come sono usati i seq_file e le due strutture dati enunciate prima. I pezzi sono presi dal kernel e si trovano sul file `trace.c`.
In particolare è mostrata la gestione del file `trace` all'interno della cartella `tracing`.

```
entry = debugfs_create_file("trace", 0444, d_tracer,
&global_trace, &tracing_fops);
```

Questa chiamata crea il file `trace` e gli viene associata l'istanza di struct `file_operations` di nome `tracing_fops`.

```
static struct file_operations tracing_fops = {
      .open       = tracing_open,
      .read       = seq_read,
      .llseek           = seq_lseek,
      .release    = tracing_release,
};
```

Come si può notare, per `tracing_fops` ci sono quattro campi riempiti: apertura, lettura, seek e rilascio. `seq_read` e `seq_lseek` sono funzioni, già, esistenti, necessarie per i seq_file. La lettura non serve personalizzarla, è sufficiente, appunto, usare `seq_read`, che legge ogni record memorizzato. Stesso discorso per il puntamento, o seek, effettuato da `seq_lseek`.
Invece, l'apertura e il rilascio sono creati ad hoc per il file `trace`.

```
static int tracing_open(struct inode *inode, struct file *file)
{
	int ret;

	__tracing_open(inode, file, &ret);

	return ret;
}
```

Il cuore della funzione è all'interno di `__tracing_open`, la quale ritorna un puntatore ad un iteratore (in questo caso è ignorato) per scorrere sul ring buffer e prelevare i record da stampare, oltre a predisporre la lettura del file `trace`. La variabile `ret`, passata per indirizzo, indica se ci sono stati errori nell'esecuzione di queste operazioni. In particolare, `ret` è usata in questo passo:

```
*ret = seq_open(file, &tracer_seq_ops);
```

Se vale zero tutto si è svolto regolarmente, altrimenti ci sono stati problemi nella `seq_open` o in passi precedenti.
`seq_open` è chiamata dalle operazioni di apertura che trattano file creati con sequenze di record, come il file `trace`.
Se non ci sono stati errori, seguono le istruzioni:

```
m = file->private_data;
m->private = iter;
```

Nella prima si passa al seq_file (`m`) l'indirizzo dello spazio di memoria dove saranno salvati i record del file e la seconda memorizza l'iteratore (`iter`) nel campo dati del seq_file.
Queste tre righe di codice sono state inserite per confermare il fatto che `trace` è costruito a partire da record e per osservare come viene inizializzato il seq_file.
`tracing_release` termina di usare l'iteratore, chiama `seq_release`, che libera lo spazio usato dal seq_file e dealloca l'iteratore stesso.

```
struct seq_file *m = (struct seq_file *)file->private_data;
struct trace_iterator *iter = m->private;
```

Queste sono definizioni di variabili, messe allo scopo di indicare il ripristino dei dati da usare nella funzione stessa.
Nella chiamata a `seq_open` sono passati due parametri: il puntatore al file e l'indirizzo della variabile `tracer_seq_ops`. Quest'ultima è l'istanza di una struttura `seq_operations`.

```
static struct seq_operations tracer_seq_ops = {
```

```
        .start              = s_start,
        .next       = s_next,
        .stop       = s_stop,
        .show       = s_show,
};
```

Il puntatore ad essa viene assegnato ad un campo specifico del seq_file, registrando così come sarà usato l'iteratore.

```
1.  static void *s_next(struct seq_file *m, void *v, loff_t
    *pos)
2.  {
3.          struct trace_iterator *iter = m->private;
4.          int i = (int)*pos;
5.          void *ent;
6.
7.          (*pos)++;
8.
9.
10.         if (iter->idx > i)
11.                 return NULL;
12.
13.         if (iter->idx < 0)
14.                 ent = find_next_entry_inc(iter);
15.         else
16.                 ent = iter;
17.
18.         while (ent && iter->idx < i)
19.                 ent = find_next_entry_inc(iter);
20.
21.         iter->pos = *pos;
22.
23.         return ent;
24. }
```

Sopra è riportato il codice della sola `s_next`.
Riceve un puntatore al seq_file, un altro indirizzo non usato e un riferimento ad una posizione.
Dal seq_file si ricava l'iteratore usato per recuperare i record (riga 3).
Si esegue un “cast” (conversione) di variabile e si incrementa la posizione (dalla riga 4 alla 7).
Se l'indice dell'iteratore supera la posizione richiesta, non si può fare niente e si ritorna `NULL` (righe 10 e 11).
Se, invece, questo indice è negativo si estrapola un record dal ring buffer e si incrementa l'iteratore, salvando in `ent` il suo indirizzo, altrimenti si assegna direttamente ad `ent` la variabile `iter` (dalla riga 13 alla 16).
Finché `ent` non vale `NULL` e l'indice non corrisponde alla posizione da raggiungere, si continua a leggere record (righe 18 e 19).
Si salva la posizione e si ritorna il puntatore all'iteratore, il quale contiene dei campi con valori aggiornati (dalla riga 21 alla 23).

Con il precedente esempio si è voluto inquadrare, soprattutto, l'utilizzo di `file_operations` e `seq_operations`, oltre a qualche accenno su come è trattato un seq_file. Poi si è voluto inserire, anche il codice di `s_next`, per far vedere una possibile implementazione del campo `next`. Si ricorda che una istanza di struct `seq_operations` va realizzata completamente dallo sviluppatore, a seconda di come e dove sono registrati i suoi record in memoria che devono essere salvati su file.

4.8.3 Ring buffer

Si è giunti, ora, alla parte più importante di questa analisi. Le informazioni raccolte da ftrace devono essere scritte in qualche zona, per poter essere poi recuperate. Esiste, quindi, un posto, appositamente sviluppato, dove lo strumento inserisce quello che ogni suo tracciatore (non solo sched_switch) raccoglie: ogni tracer scrive su quello che si chiama ring buffer. Questo oggetto può essere pensato come un cerchio ("ring") che viene percorso e riempito. Ha un inizio (testa) e una fine (coda) che possono coincidere in due situazioni:

- il buffer è vuoto;
- il buffer è pieno.

Questa struttura dati, in ftrace, ha un funzionamento complicato basato su paginazione. Una pagina del buffer è una struct con dei campi e, in particolare, contiene un'altra struttura che mette a disposizione uno spazio (vettore) per salvare i dati della pagina stessa.
Viene riportato il codice delle due struct relative alle pagine:

```
struct buffer_data_page {
      u64          time_stamp;      /* page time stamp */
      local_t            commit;    /* write commited index */
      unsigned char      data[];    /* data of buffer page */
};

struct buffer_page {
      local_t            write;           /*   index   for   next
write */
      unsigned     read;            /* index for next read */
      struct list_head list;        /* list of free pages */
      struct buffer_data_page *page;      /* Actual data page */
};
```

Il vettore `data[]` è il contenitore informativo della pagina e ogni casella può registrare un byte.
Una pagina del buffer non è infinita (anche se `data[]` non è un vettore dimensionato), ha una dimensione massima pari a 4084 byte.
Le pagine sono, quindi, le unità di storage del ring buffer. Questo oggetto è, anche, lui caratterizzato da una struttura dati con vari campi, il cui codice è il seguente:

```
struct ring_buffer {
      unsigned                pages;
      unsigned                flags;
```

```
        int                     cpus;
        cpumask_var_t                   cpumask;
        atomic_t                record_disabled;

        struct mutex                    mutex;

        struct ring_buffer_per_cpu    **buffers;
};
```

Oltre a questa struttura, per ftrace vengono creati tanti buffer quanti sono i processori del computer, ovvero ogni CPU ha il suo buffer, il quale è costituito da un'ulteriore struct:

```
struct ring_buffer_per_cpu {
        int                     cpu;
        struct ring_buffer              *buffer;
        spinlock_t              reader_lock; /* serialize readers
*/
        raw_spinlock_t                  lock;
        struct lock_class_key           lock_key;
        struct list_head        pages;
        struct buffer_page              *head_page; /*   read   from
head */
        struct buffer_page              *tail_page; /* write to tail
*/
        struct buffer_page              *commit_page;     /*
commited pages */
        struct buffer_page              *reader_page;
        unsigned long                   overrun;
        unsigned long                   entries;
        u64                     write_stamp;
        u64                     read_stamp;
        atomic_t                record_disabled;
};
```

Ftrace alloca due ring buffer: uno serve per salvare i dati del tracing corrente e l'altro viene usato per fare una fotografia del primo, quando si raggiunge una latenza massima.
La funzione di allocazione si chiama `ring_buffer_alloc`. Come parametri vuole la dimensione in byte per ogni processore (usata per creare il numero di pagine) e i flag di impostazione del buffer. L'unico valore attribuibile ai flag è `RB_FL_OVERWRITE` e sta a significare che quando il buffer è pieno, le pagine vecchie vengono sovrascritte dalle nuove. Se non è impostato questo valore, allora quando la coda tocca la testa (buffer riempito), i dati nuovi in arrivo sono scartati. La funzione ritorna il puntatore al ring buffer.
Vediamo alcune parti della costruzione dei buffer di ftrace.
Si riserva lo spazio fisico per la struttura dati:

```
buffer = kzalloc(ALIGN(sizeof(*buffer), cache_line_size()),
                 GFP_KERNEL);
```

Creazione del numero di pagine (almeno due) e inserimento flag, maschera di CPU e numero di processori del sistema:

```
buffer->pages = DIV_ROUND_UP(size, BUF_PAGE_SIZE);
```

```
buffer->flags = flags;


if (buffer->pages == 1)
      buffer->pages++;

cpumask_copy(buffer->cpumask, cpu_possible_mask);
buffer->cpus = nr_cpu_ids;
```

Spazio per gli indirizzi dei buffer che saranno creati per ogni CPU:

```
bsize = sizeof(void *) * nr_cpu_ids;
buffer->buffers   =   kzalloc(ALIGN(bsize,   cache_line_size()),
GFP_KERNEL);
```

Allocazione delle struct `ring_buffer_per_cpu` (una per processore):

```
for_each_buffer_cpu(buffer, cpu) {
      buffer->buffers[cpu] =
      rb_allocate_cpu_buffer(buffer, cpu);
      if (!buffer->buffers[cpu])
            goto fail_free_buffers;
}
```

Inizializzazione di un meccanismo di mutua esclusione e ritorno del puntatore:

```
mutex_init(&buffer->mutex);

return buffer;
```

Non indicate, ma presenti nella funzione sono le “vie d'uscita” in caso di errore: se necessario, si libera la memoria coinvolta e si ritorna `NULL`.
È stato, così, creato il buffer “globale”. Analizziamo ora come sono allocati i buffer in ogni processore, usati dai tracciatori per salvare il lavoro da loro eseguito. La funzione è `rb_allocate_cpu_buffer`. I parametri sono il puntatore alla struttura `ring_buffer` e l'indice della CPU su cui allocare questo buffer.
Recupero spazio per la creazione della struct per il processore di indice `cpu`:

```
cpu_buffer       =       kzalloc_node(ALIGN(sizeof(*cpu_buffer),
cache_line_size()), GFP_KERNEL, cpu_to_node(cpu));
```

Inizializzazione variabili interne:

```
cpu_buffer->cpu = cpu;
cpu_buffer->buffer = buffer;
spin_lock_init(&cpu_buffer->reader_lock);
cpu_buffer->lock = (raw_spinlock_t)__RAW_SPIN_LOCK_UNLOCKED;
INIT_LIST_HEAD(&cpu_buffer->pages);
```

Costruzione, tramite un supporto, del campo `reader_page`:

```
bpage  =  kzalloc_node(ALIGN(sizeof(*bpage),  cache_line_size()),
GFP_KERNEL, cpu_to_node(cpu));
if (!bpage)
      goto fail_free_buffer;
```

```
cpu_buffer->reader_page = bpage;
addr = __get_free_page(GFP_KERNEL);
if (!addr)
      goto fail_free_reader;
bpage->page = (void *)addr;
rb_init_page(bpage->page);

INIT_LIST_HEAD(&cpu_buffer->reader_page->list);
```

Allocazione pagine del buffer (il numero è sempre lo stesso per ogni `ring_buffer_per_cpu` ed è contenuto in `buffer->pages`):

```
ret = rb_allocate_pages(cpu_buffer, buffer->pages);
```

Creazione dei campi `tail_page`, `commit_page` e `head_page` e ritorno del puntatore al buffer del processore:

```
cpu_buffer->head_page=        list_entry(cpu_buffer->pages.next,
struct buffer_page, list);
cpu_buffer->tail_page = cpu_buffer->commit_page = cpu_buffer-
>head_page;

return cpu_buffer;
```

Sono presenti, anche qui, le “uscite di sicurezza” se si verificano errori.
Lo schema successivo vuole mostrare, graficamente, il legame tra il buffer globale e i buffer di ogni CPU.

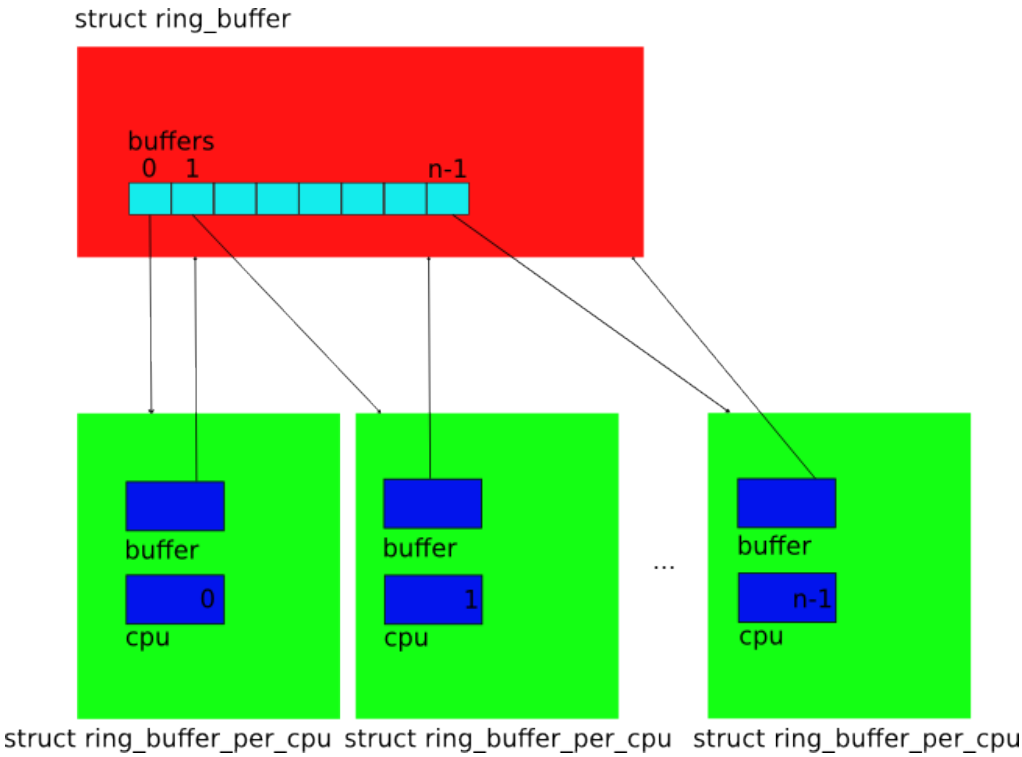

Illustrazione 4: Legame tra la struct `ring_buffer` e le struct `ring_buffer_per_cpu` (una per processore)

Si procede, adesso, con lo studio dell'allocazione delle pagine, elementi dove si scrivono e leggono i dati di ftrace, per concludere la carellata sul setup iniziale del ring buffer. `rb_allocate_pages`, chiamata nella funzione vista prima, richiede due informazioni in ingresso: il puntatore al `ring_buffer_per_cpu` corrente e il numero di pagine da creare.
Variabili della funzione:

```
struct list_head *head = &cpu_buffer->pages;
struct buffer_page *bpage, *tmp;
unsigned long addr;
LIST_HEAD(pages);
unsigned i;
```

Segue un ciclo in cui, in ogni iterazione, viene allocata una pagina, rappresentata da una struttura `buffer_page`. Si riserva, quindi, dello spazio per essa con `kzalloc_node` e l'indirizzo di memoria è scritto in `bpage`. `bpage` con valore `NULL` significa che in memoria non c'è abbastanza posto per la struct, allora si prende una "via di fuga" e si ritorna un errore. In caso contrario, si esegue l'aggancio tra `bpage->list` e la variabile `pages`. Il campo `list` di `bpage` e `pages` sono due strutture di tipo `list_head` e sono connessi fra loro in modo da creare una lista percorribile in tutti e due i sensi (doppiamente concatenata). `list_add`, infatti, aggiunge, ad ogni ciclo, `bpage->list` corrente (il valore di `bpage` cambia ad ogni iterazione) tra `pages` e la `list_head` puntata da `pages.next` (`next` e `prev` sono i due campi di `list_head`). Le pagine del buffer sono collegate, una all'altra, da questo meccanismo a lista. I dati veri e propri sono memorizzati all'interno di `page` (struttura `buffer_data_page`), campo di `buffer_page`. Si deve, allora, trovare uno spazio per essa. `__get_free_page` serve a questo scopo: trova un indirizzo da assegnare a questo campo. Se ritorna `NULL` si è verificato un errore e si deve uscire dal ciclo. Ultima istruzione, `rb_init_page` inizializza a zero il valore di `commit`, variabile interna a `bpage->page`. Codice:

```
for (i = 0; i < nr_pages; i++) {
      bpage         =         kzalloc_node(ALIGN(sizeof(*bpage),
cache_line_size()), GFP_KERNEL, cpu_to_node(cpu_buffer->cpu));
      if (!bpage)
            goto free_pages;
      list_add(&bpage->list, &pages);

      addr = __get_free_page(GFP_KERNEL);
      if (!addr)
            goto free_pages;
      bpage->page = (void *)addr;
      rb_init_page(bpage->page);
}
```

Si prosegue con il collegamento di `pages` a `head`, la quale punta a `cpu_buffer->pages`, cioè le pagine create vengono unite al buffer, si effettua un controllo di integrità e si ritorna zero (nessun errore è accaduto):

```
list_splice(&pages, head);

rb_check_pages(cpu_buffer);
```

```
return 0;
```

La “via di fuga” elimina ogni pagina creata e ritorna un errore:

```
free_pages:
      list_for_each_entry_safe(bpage, tmp, &pages, list) {
            list_del_init(&bpage->list);
            free_buffer_page(bpage);
      }
      return -ENOMEM;
```

Si conclude, così, la spiegazione del setup iniziale del ring buffer. Riassumendo per tre punti, la creazione è la seguente:

1. `ring_buffer_alloc` costruisce il buffer globale (struct `ring_buffer`);
2. `rb_allocate_cpu_buffer` alloca un buffer (struct `ring_buffer_per_cpu`) in un processore;
3. `rb_allocate_pages` crea lo stesso numero di pagine per ogni buffer del punto 2.

Segue, anche in questo caso, uno schema illustrativo della lista che tiene collegata le pagine del buffer.

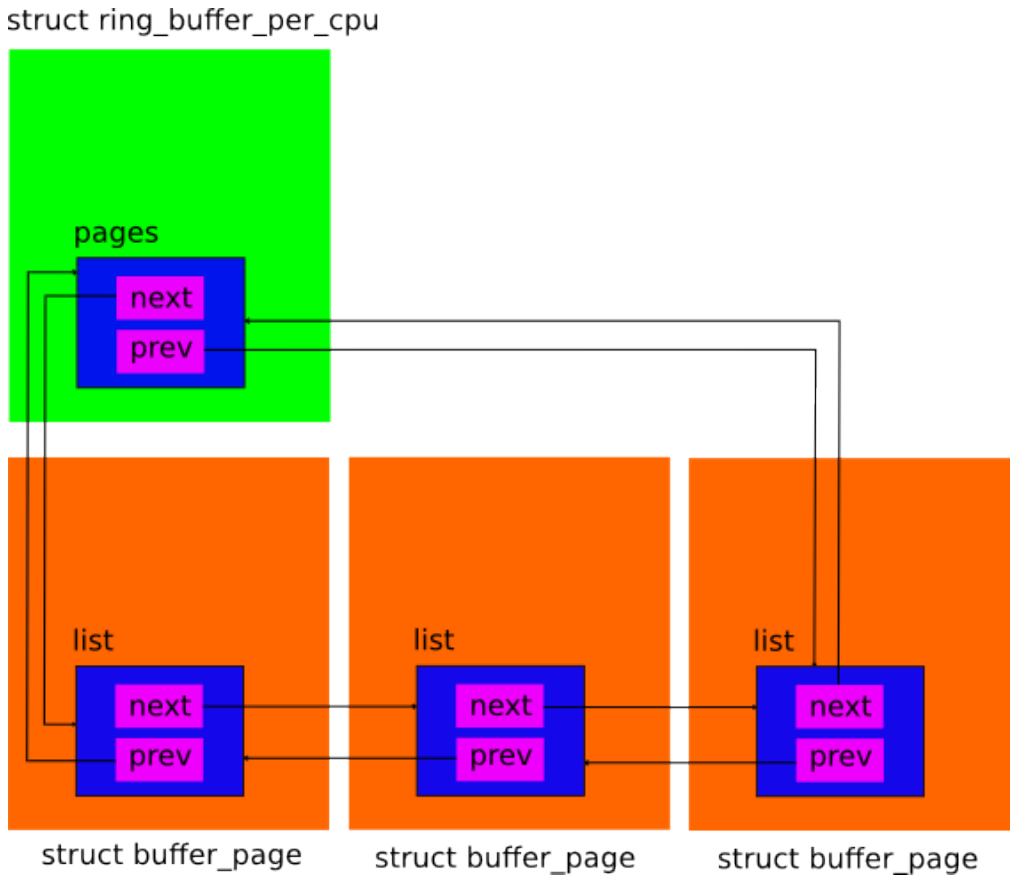

Illustrazione 5: Pagine del ring buffer

Vista l'allocazione, si può iniziare l'analisi della scrittura sul ring buffer. Sostanzialmente è realizzata da due funzioni, `ring_buffer_lock_reserve` e

`ring_buffer_unlock_commit`, che devono essere usate assieme: la prima riserva una zona sul buffer per poter scrivere e la seconda aggiorna gli indici sulle pagine, una volta salvati i dati.
È necessario, però, per poter procedere, introdurre un'altra struttura dati, la struct `ring_buffer_event`:

```
struct ring_buffer_event {
      u32         type:2, len:3, time_delta:27;
      u32         array[];
};
```

Il primo campo è lungo 32 bit ed è suddiviso in tre sezioni: `type` di 2 bit (tipo di informazione), `len` di 3 bit (lunghezza codificata dell'evento) e `time_delta` di 27 bit (utile per calcolare il successivo istante di scrittura a partire dal precedente). Il secondo è un vettore, non dimensionato, di elementi da 4 byte.
Esistono quattro tipi di eventi, ma usati nella pratica sono tre. Il campo `type` può assumere i seguenti valori, espressi in binario:

1. `RINGBUF_TYPE_PADDING`: corrisponde a 00 (il valore dei due bit di `type`) ed è usato per completare il riempimento di una pagina, con informazioni di nessun interesse, quando bisogna cambiare pagina per memorizzare i nuovi dati;
2. `RINGBUF_TYPE_TIME_EXTEND`: corrisponde a 01 ed è usato quando un istante di scrittura è troppo grande (per la sua rappresentazione binaria servono più di 27 bit), ossia nel ring buffer viene lasciato dello spazio per scrivere questo tempo (vengono usati `time_delta` e la prima cella di `array`, `array[0]`);
3. `RINGBUF_TYPE_TIME_STAMP`: corrisponde a 10 ed è il tipo non usato;
4. `RINGBUF_TYPE_DATA`: corrisponde a 11 e rappresenta il tipo per i dati che si vogliono scrivere.

Nell'evento sono, quindi, memorizzate informazioni di un certo tipo e di una certa lunghezza (in byte): il primo campo rappresenta l'intestazione e il vettore `array[]` salva i dati di ftrace (a volte l'intestazione può occupare, anche, la prima cella di `array[]`). `ring_buffer_lock_reserve`, citata in precedenza, rende, quindi, disponibile un puntatore ad un evento che è l'indirizzo della prima cella libera del campo `data[]` di una pagina del ring buffer (`data[]` fa parte di una `buffer_data_page` contenuta in una `buffer_page`). Il nocciolo della questione è che, mentre si sta scrivendo sull'evento contemporaneamente si scrive, anche, sul ring buffer.
Va, inoltre, aggiunto che i campi `tail_page` e `commit_page` di un `ring_buffer_per_cpu` sono molto importanti in fase di scrittura: infatti, gli eventi sono scritti in `tail_page` (nel vettore `data[]`) e la `commit_page`, pagina di lavoro dove sono modificati gli indici dopo che i dati sono stati salvati, viene fatta puntare alla stessa struttura puntata dal campo `tail_page` (cambiare gli indici di `commit_page` equivale a cambiare quelli di `tail_page`).
Vediamo la funzione in dettaglio:

```
1. struct ring_buffer_event *
2. ring_buffer_lock_reserve(struct ring_buffer *buffer,
3.                          unsigned long length,
4.                          unsigned long *flags)
```

```
{
        struct ring_buffer_per_cpu *cpu_buffer;
        struct ring_buffer_event *event;
        int cpu, resched;

        if (ring_buffer_flags != RB_BUFFERS_ON)
                return NULL;

        if (atomic_read(&buffer->record_disabled))
                return NULL;

        /* If we are tracing schedule, we don't want to
recurse */
        resched = ftrace_preempt_disable();

        cpu = raw_smp_processor_id();

        if (!cpumask_test_cpu(cpu, buffer->cpumask))
                goto out;

        cpu_buffer = buffer->buffers[cpu];

        if (atomic_read(&cpu_buffer->record_disabled))
                goto out;

        length = rb_calculate_event_length(length);
        if (length > BUF_PAGE_SIZE)
                goto out;

        event       =       rb_reserve_next_event(cpu_buffer,
RINGBUF_TYPE_DATA, length);
        if (!event)
                goto out;

        /*
         * Need to store resched state on this cpu.
         * Only the first needs to.
         */

        if (preempt_count() == 1)
                per_cpu(rb_need_resched, cpu) = resched;

        return event;

 out:
        ftrace_preempt_enable(resched);
        return NULL;
}
```

In ingresso richiede l'indirizzo di `ring_buffer`, la lunghezza della quantità informativa da memorizzare e un puntatore per salvare i flag degli interrupt. In uscita fornisce lo spazio sotto forma di evento.
Variabili di utilizzo, dalla riga 6 alla 8: `cpu_buffer` memorizza il puntatore al buffer della CPU corrente, `event` contiene l'indirizzo dell'evento riservato, `cpu` l'indice del processore corrente e `resched` salva il ritorno della disabilitazione della preemption.
Se il buffer è disattivato oppure se è attivo, ma è disabilitata la registrazione su di esso si ritorna `NULL` (dalla riga 10 alla 14).
Si disattiva la preemption e si richiede l'indice della CPU corrente (dalla riga 17 alla 19).
Si controlla se è possibile usare il buffer della CPU corrente (righe 21 e 22).
Si prende il riferimento del buffer del processore corrente (riga 24).
Se la registrazione su di esso è disabilitata si "salta" nella zona di codice etichettata con out, "via d'uscita" (righe 26 e 27).
La lunghezza passata come parametro è la dimensione delle sole informazioni di ftrace, ma, come accennato prima, viene salvato l'intero evento compresa l'intestazione e si deve tenere conto anche di questa. Inoltre il campo `array[]` della struttura `ring_buffer_event` contiene celle da 32 bit l'una e se una di queste non è riempita del tutto, viene, comunque, considerata come completa. In altre parole, se ftrace deve inserire *x* byte, il numero di caselle occupate in `array[]` sarà *ceil (x/4)*, dove *con ceil* si indica l'intero superiore del suo argomento, invece le celle di `data[]` occupate sono *ceil (x/4)*4* perché ognuna ha grandezza pari ad un byte. `rb_calculate_event_length` prende in input la lunghezza originale e la modifica in modo che, poi, possa essere riservato lo spazio per tutto l'evento. Se questa nuova lunghezza, però, supera il valore di *BUF_PAGE_SIZE*, ossia 4084 byte, significa che i dati non ci stanno su una pagina e questo è un errore, quindi, bisogna seguire la "via d'uscita" (dalla riga 29 alla 31).
A questo punto della funzione sono presenti gli elementi per riservare lo spazio come evento. `rb_reserve_next_event` creerà un evento sul buffer della CPU corrente, `cpu_buffer`, per memorizzare dati normali di ftrace, `RINGBUF_TYPE_DATA`, che occuperà un numero di byte pari a `length`.
Oltre a questo tipo, saranno, anche, creati eventi di tipo `RINGBUF_TYPE_TIME_EXTEND`, quando è necessario ulteriore spazio per salvare un istante di scrittura e, invece, `RINGBUF_TYPE_PADDING`, viene chiamato in causa per riempire una pagina qualora i successivi dati da scrivere non ci stanno e vengono, quindi, inseriti in una nuova pagina.
`event` `NULL` è un altro errore e si deve andare nella zona `out` (dalla riga 33 alla 35).
Si salva la variabile `resched` e si torna `event` (dalla riga 42 alla 45).
La "via d'uscita" `out` riabilita la preemption e ritorna `NULL` (dalla riga 47 alla 49).

Se la variabile `event` tornata non è `NULL`, si ha accesso per scrivere sul ring buffer, ma prima è necessario usare un'altra funzione. Un evento ha un'intestazione che viene anch'essa scritta sul buffer. `ring_buffer_event_data` tiene conto di questa intestazione e ritorna il puntatore da cui si può iniziare a salvare i dati di interesse.
Un esempio, tratto dal codice di ftrace, di memorizzazione sul ring buffer si è

visto quando si sono spiegate le funzioni principali, `probe_sched_switch` e `probe_sched_wakeup`. In particolare era stata riportata la scrittura di uno switch e di una sveglia. Viene ripreso lo switch:

```
event  =  ring_buffer_lock_reserve(tr->buffer,  sizeof(*entry),
&irq_flags);
if (!event)
      return;
entry = ring_buffer_event_data(event);
tracing_generic_entry_update(&entry->ent, flags, pc);
entry->ent.type               = TRACE_CTX;
entry->prev_pid               = prev->pid;
entry->prev_prio        = prev->prio;
entry->prev_state       = prev->state;
entry->next_pid               = next->pid;
entry->next_prio        = next->prio;
entry->next_state       = next->state;
entry->next_cpu   = task_cpu(next);
```

Si nota la chiamata a `ring_buffer_lock_reserve` che ritorna l'evento e, poi, segue `ring_buffer_event_data`. `entry`, puntatore a struct `ctx_switch_entry`, contiene il riferimento dei dati della commutazione di contesto che stanno per venire scritti.
Questo è un esempio, ma sul ring buffer può essere salvato qualsiasi tipo di dato: è sufficiente che il valore tornato da `ring_buffer_event_data` sia inserito in un puntatore alla variabile o struttura che si vuole scrivere.
Memorizzate le informazioni, per terminare la scrittura, occorre spostare l'indice del vettore `data[]` per farlo puntare alla successiva cella libera. Questo è il compito della seconda funzione, `ring_buffer_unlock_commit`. In realtà, i campi che fungono da indice sono due: `commit` di una `buffer_data_page` e `write` di una `buffer_page`. Loro due, alla fine della scrittura, hanno lo stesso valore: `write` cambia, addirittura, nella fase precedente, durante la “creazione” dell'evento di scrittura e `commit`, invece, viene aggiornato dopo che i dati sono stati scritti.

```
1.  int ring_buffer_unlock_commit(struct ring_buffer *buffer,
2.                            struct ring_buffer_event *event,
3.                            unsigned long flags)
4.  {
5.        struct ring_buffer_per_cpu *cpu_buffer;
6.        int cpu = raw_smp_processor_id();
7.
8.        cpu_buffer = buffer->buffers[cpu];
9.
10.       rb_commit(cpu_buffer, event);
11.
12.       /*
13.         * Only the last preempt count needs to restore
    preemption.
14.         */
15.       if (preempt_count() == 1)
16.
```

```
        ftrace_preempt_enable(per_cpu(rb_need_resched,
cpu));
17.     else
18.         preempt_enable_no_resched_notrace();
19.
20.     return 0;
21. }
```

I parametri che vuole sono tre: il puntatore al ring buffer, l'evento che è stato scritto e i flag di interrupt ricevuti da `ring_buffer_lock_reserve`.
Variabili della funzione dalla riga 5 alla 6: `cpu_buffer`, buffer sotto esame e in `cpu` è messo l'indice del processore corrente.
Si prende il riferimento al CPU buffer corrente, quello di indice `cpu` (riga 8).
`rb_commit` svolge diversi compiti. Richiede il CPU buffer e l'evento. Prima di tutto, viene incrementato di una unità il campo `entries` di `cpu_buffer` segnando che è stato inserito qualcosa sul ring buffer, poi, tramite un'ulteriore funzione, viene verificato se la scrittura può, effettivamente, essere registrata e se è così viene calcolato l'istante in cui la scrittura è avvenuta e salvato in `cpu_buffer->write_stamp`. Chiude la funzione l'allineamento, se necessario, della `commit_page` alla `tail_page` (con aggiornamento di alcuni campi di `cpu_buffer` come il `write_stamp`) e la modifica dell'indice `commit` (riga 10).
Viene riabilitata la preemption, disattivata da `ring_buffer_lock_reserve` e si ritorna zero (dalla riga 15 alla 20).

Con questa funzione termina l'analisi della scrittura sul ring buffer. Può essere riassunta in tre punti:

1. `ring_buffer_lock_reserve` rende disponibile un evento scrittura da riempire con i dati da salvare (scrivere sull'evento equivale a scrivere sul buffer);
2. inserimento informazioni nel buffer;
3. `ring_buffer_unlock_commit` sposta gli indici della pagina per, effettivamente, memorizzare la scrittura.

Con l'immagine che segue si vuole sottolineare il fatto che il puntatore all'evento, ritornato da `ring_buffer_lock_reserve`, è l'indirizzo di una cella del vettore `data[]`.

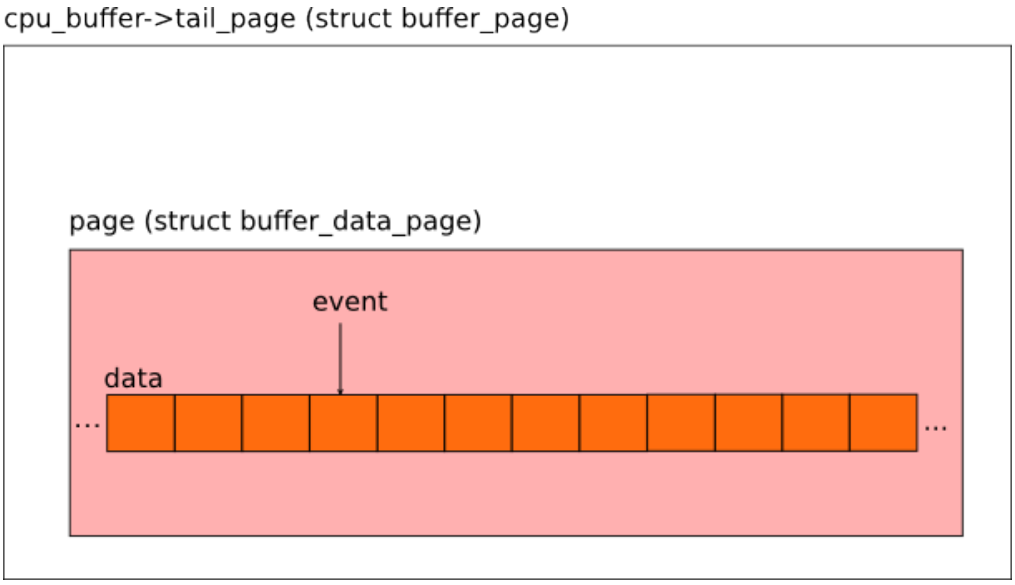

Illustrazione 6: Scrittura su una pagina del buffer della CPU corrente

L'altra operazione fondamentale sul buffer è la lettura delle informazioni scritte che, poi, ftrace visualizzerà con il file `trace`, usando un seq_file di appoggio. Può essere realizzata con due modalità diverse.
Lettura scandita da un iteratore è il primo modo analizzato. Se si ha a disposizione un ambiente multiprocessore, viene creato un iteratore per CPU, ovvero per struct `ring_buffer_per_cpu`.
Va, quindi, accennata la struttura `ring_buffer_iter`, usata come iteratore:

```
struct ring_buffer_iter {
      struct ring_buffer_per_cpu    *cpu_buffer;
      unsigned long                 head;
      struct buffer_page            *head_page;
      u64                     read_stamp;
};
```

La funzione che agisce sul ring buffer è la `ring_buffer_iter_peek`, il cui codice è il seguente:

```
1.  struct ring_buffer_event *
2.  ring_buffer_iter_peek(struct ring_buffer_iter *iter, u64
    *ts)
3.  {
4.        struct  ring_buffer_per_cpu  *cpu_buffer  =  iter-
    >cpu_buffer;
5.        struct ring_buffer_event *event;
6.        unsigned long flags;
7.
8.        spin_lock_irqsave(&cpu_buffer->reader_lock, flags);
9.        event = rb_iter_peek(iter, ts);
10.       spin_unlock_irqrestore(&cpu_buffer->reader_lock,
    flags);
```

```
11.
12.         return event;
13. }
```

Richiede il puntatore all'iteratore e un indirizzo di uno spazio a 64 bit. Torna il riferimento all'evento dati da leggere.
Variabili dalla riga 4 alla 6: `cpu_buffer`, buffer su cui si legge, `event`, evento dati e `flags` per i flag di interrupt.
Si entra in una sezione critica e si salvano i flag di interrupt (riga 8).
Si estrapola l'evento (riga 9).
Si esce dalla sezione critica e si ripristinano i flag di interrupt (riga 10).
Si ritorna il puntatore all'evento (riga 12).
Questa funzione agisce come un guscio, il vero lavoro è realizzato da `rb_iter_peek`:

```
1.  static struct ring_buffer_event *
2.  rb_iter_peek(struct ring_buffer_iter *iter, u64 *ts)
3.  {
4.        struct ring_buffer *buffer;
5.        struct ring_buffer_per_cpu *cpu_buffer;
6.        struct ring_buffer_event *event;
7.        int nr_loops = 0;
8.
9.        if (ring_buffer_iter_empty(iter))
10.             return NULL;
11.
12.       cpu_buffer = iter->cpu_buffer;
13.       buffer = cpu_buffer->buffer;
14.
15.  again:
16.       /*
17.        * We repeat when a timestamp is encountered. It is
    possible
18.        * to get multiple timestamps from an interrupt
    entering just
19.        * as one timestamp is about to be written. The max
    times
20.        * that this can happen is the number of nested
    interrupts we
21.        * can have. Nesting 10 deep of interrupts is
    clearly
22.        * an anomaly.
23.        */
24.       if (RB_WARN_ON(cpu_buffer, ++nr_loops > 10))
25.             return NULL;
26.
27.       if (rb_per_cpu_empty(cpu_buffer))
28.             return NULL;
29.
30.       event = rb_iter_head_event(iter);
```

```
31.
32.       switch (event->type) {
33.       case RINGBUF_TYPE_PADDING:
34.             rb_inc_iter(iter);
35.             goto again;
36.
37.       case RINGBUF_TYPE_TIME_EXTEND:
38.             /* Internal data, OK to advance */
39.             rb_advance_iter(iter);
40.             goto again;
41.
42.       case RINGBUF_TYPE_TIME_STAMP:
43.             /* FIXME: not implemented */
44.             rb_advance_iter(iter);
45.             goto again;
46.
47.       case RINGBUF_TYPE_DATA:
48.             if (ts) {
49.                   *ts    =    iter->read_stamp    +    event-
   >time_delta;
50.
            ring_buffer_normalize_time_stamp(cpu_buffer->cpu,
   ts);
51.             }
52.             return event;
53.
54.       default:
55.             BUG();
56.       }
57.
58.       return NULL;
59. }
```

Vuole gli stessi ingressi e torna lo stesso tipo di uscita di `ring_buffer_iter_peek`.
Variabili della funzione dalla riga 4 alla 7: `buffer` è l'indirizzo del ring buffer, `cpu_buffer` è il buffer di processore, `event` punta alle informazioni recuperate e `nr_loops` serve per ciclare.
Viene ritornato `NULL` se `iter` non ha più nessun evento da estrarre (righe 9 e 10).
Recupero dei riferimenti al CPU buffer e al buffer globale (righe 12 e 13).
Inizio zona `again`: se si passa più di dieci volte per questa zona si ritorna `NULL` (dalla riga 15 alla 25).
Se `cpu_buffer` è vuoto si torna `NULL` (righe 27 e 28).
Si ottiene l'evento dal campo `head_page` di `iter`, indirizzo di una cella del vettore `data[]` di `iter->head_page->page` (riga 30).
Verifica del tipo di evento. Ci sono quattro casi:

1. `RINGBUF_TYPE_PADDING`, `rb_inc_iter` cambia la pagina di lavoro dell'iteratore, aggiorna il suo campo `read_stamp` e azzera l'indice `head`. Si ritorna nella zona `again`;

2. `RINGBUF_TYPE_TIME_EXTEND`, con `rb_advance_iter`, se non ci sono, ancora, dati da leggere in `iter->head_page`, se necessario, viene chiamata `rb_inc_iter` e la funzione termina, altrimenti viene riestratto l'evento e calcolata la sua lunghezza. Se non si è sulla coda di `cpu_buffer`, viene salvato l'istante di scrittura dell'evento e la sua lunghezza aggiunta a `iter->head`. Se, ora, `iter->head` supera la dimensione di `iter->head_page` (non ci sono più dati), però, quest'ultima non punta alla stessa struttura puntata da `cpu_buffer->commit_page` (l'iteratore non sta leggendo dall'ultima pagina scritta), bisogna richiamare ricorsivamente `rb_advance_iter`. Appena questa funzione termina si va nella zona `again`;
3. `RINGBUF_TYPE_TIME_STAMP`, come caso 2;
4. `RINGBUF_TYPE_DATA`, se `ts` non è `NULL` viene memorizzato l'istante di scrittura e tornato `event`.

`NULL` è ritornato se non si entra in nessuno dei quattro casi appena visti (dalla riga 32 alla 58).

La seconda via per leggere i dati non utilizza un iteratore, ma li preleva "manualmente" dal ring buffer. Il codice di interesse è identificato dalla funzione `ring_buffer_peek`:

```
1.  struct ring_buffer_event *
2.  ring_buffer_peek(struct ring_buffer *buffer, int cpu, u64
    *ts)
3.  {
4.        struct ring_buffer_per_cpu *cpu_buffer = buffer-
    >buffers[cpu];
5.        struct ring_buffer_event *event;
6.        unsigned long flags;
7.
8.        spin_lock_irqsave(&cpu_buffer->reader_lock, flags);
9.        event = rb_buffer_peek(buffer, cpu, ts);
10.       spin_unlock_irqrestore(&cpu_buffer->reader_lock,
    flags);
11.
12.       return event;
13. }
```

Ingresso: il buffer globale, `buffer`, la CPU corrente, `cpu` e l'indirizzo di uno spazio a 8 byte per salvare un istante di tempo, `ts`.
Uscita: puntatore all'evento estratto, contenente i dati.
Variabili locali dalla riga 4 alla 6: `cpu_buffer`, buffer su cui si legge, `event`, evento dati e `flags` per i flag di interrupt.
Si entra in una sezione critica e si salvano i flag di interrupt (riga 8).
In `event` viene salvato l'indirizzo dell'evento estrapolato dal buffer (riga 9).
Si esce dalla sezione critica e si ripristinano i flag di interrupt (riga 10).
Ritorno di `event` (riga 12).
Viene spiegato, adesso, come avviene il recupero dell'evento. Codice di `rb_buffer_peek`:

```
1.  static struct ring_buffer_event *
2.  rb_buffer_peek(struct ring_buffer *buffer, int cpu, u64
    *ts)
3.  {
4.        struct ring_buffer_per_cpu *cpu_buffer;
5.        struct ring_buffer_event *event;
6.        struct buffer_page *reader;
7.        int nr_loops = 0;
8.
9.        if (!cpumask_test_cpu(cpu, buffer->cpumask))
10.             return NULL;
11.
12.       cpu_buffer = buffer->buffers[cpu];
13.
14.  again:
15.       /*
16.        * We repeat when a timestamp is encountered. It is
    possible
17.        * to get multiple timestamps from an interrupt
    entering just
18.        * as one timestamp is about to be written. The max
    times
19.        * that this can happen is the number of nested
    interrupts we
20.        * can have.  Nesting 10 deep of interrupts is
    clearly
21.        * an anomaly.
22.        */
23.       if (RB_WARN_ON(cpu_buffer, ++nr_loops > 10))
24.             return NULL;
25.
26.       reader = rb_get_reader_page(cpu_buffer);
27.       if (!reader)
28.             return NULL;
29.
30.       event = rb_reader_event(cpu_buffer);
31.
32.       switch (event->type) {
33.       case RINGBUF_TYPE_PADDING:
34.             RB_WARN_ON(cpu_buffer, 1);
35.             rb_advance_reader(cpu_buffer);
36.             return NULL;
37.
38.       case RINGBUF_TYPE_TIME_EXTEND:
39.             /* Internal data, OK to advance */
40.             rb_advance_reader(cpu_buffer);
41.             goto again;
42.
43.       case RINGBUF_TYPE_TIME_STAMP:
44.             /* FIXME: not implemented */
```

```
45.             rb_advance_reader(cpu_buffer);
46.             goto again;
47.
48.       case RINGBUF_TYPE_DATA:
49.             if (ts) {
50.                   *ts = cpu_buffer->read_stamp + event->time_delta;
51.
      ring_buffer_normalize_time_stamp(cpu_buffer->cpu, ts);
52.             }
53.             return event;
54.
55.       default:
56.             BUG();
57.       }
58.
59.       return NULL;
60. }
```

I parametri di input e output sono gli stessi della funzione precedente.
Variabili dalla riga 4 alla 7: `cpu_buffer` è il buffer di processore, `event` punta alle informazioni recuperate, `reader` punta alla pagina da cui leggere e `nr_loops` serve per ciclare.
Controllo sulla possibilità di utilizzo della struct `ring_buffer_per_cpu` costruita per il processore di indice `cpu` (righe 9 e 10).
Si preleva il buffer di interesse (riga 12).
Inizio zona `again`: se si passa più di dieci volte per questa zona si ritorna `NULL` (dalla riga 14 alla 24).
Viene restituito in `reader` l'indirizzo della pagina da cui leggere le informazioni. Si ritorna `NULL` se `reader` non punta a nessuna pagina (dalla riga 26 alla 28).
Si salva il puntatore all'evento, indirizzo di una cella del vettore `data[]` di `cpu_buffer->reader_page->page` (riga 30).
Ottenuto l'evento, bisogna verificare di che tipo è. Ci sono quattro casi:

1. `RINGBUF_TYPE_PADDING`, viene chiamata `rb_advance_reader`, la quale inserisce in `cpu_buffer->read_stamp`, l'istante di scrittura dell'evento, calcola la sua lunghezza e l'aggiunge all'indice `cpu_buffer->reader_page->read`, usato per la lettura. Poi si torna `NULL`;
2. `RINGBUF_TYPE_TIME_EXTEND`, `rb_advance_reader` esegue le stesse operazioni del caso 1 e poi si torna nella zona `again`;
3. `RINGBUF_TYPE_TIME_STAMP`, come caso 2;
4. `RINGBUF_TYPE_DATA`, viene tornato il puntatore all'evento salvando prima, nello spazio indirizzato da `ts` (se non è `NULL`), l'istante di scrittura.

Se non si rientra in nessuno di questi casi si ritorna `NULL` (dalla riga 32 alla 59).

Si può sintetizzare la lettura in due punti:

1. con iteratore;

2. direttamente dal CPU buffer corrente.

Entrambi i modi visti restituiscono un puntatore ad un evento di tipo dati (`RINGBUF_TYPE_DATA`), da cui possono essere lette le informazioni precedentemente scritte.

Si chiude, così, lo studio di questo importante oggetto, il ring buffer. L'analisi fornita si basa, fondamentalmente, sulle funzioni usate da ftrace per lavorare con esso.

5 GTKWave

Dopo ftrace è utile dare una breve descrizione dello strumento usato per fornire la visualizzazione grafica del tracing testuale prodotto da ftrace stesso.
GTKWave è un visualizzatore di forme d'onda elettroniche basato sulla libreria GTK+ che legge file con estensione FST, LXT, LXT2, VZT e GHW come anche i file dello standard Verilog VCD/EVCD permettendo la loro visualizzazione. GTKWave è sviluppato per Linux e, anche, per altri sistemi operativi inclusi Microsoft Windows e Mac OS X.
GTKWave è progettato per gestire molti segnali alla volta e possiede l'abilità di mostrare i dati in differenti formati: decimale, esadecimale, ottale, ASCII, numeri reali, binario e anche analogico.
Il tracciamento elaborato da ftrace viene salvato in un file con estensione ".txt" (testo). Successivamente, quindi, bisogna convertire questo file in uno supportato da GTKWave: in particolare, il formato utilizzato è il VCD (Value Change Dump). La conversione è stata realizzata con un programma, derivante dalla compilazione di un ".c", che si chiama `sched_switch`: con un terminale si digiti, all'interno della cartella contenente l'eseguibile `sched_switch`:

```
$ ./sched_switch nome_file.txt nome_file.vcd
```

`nome_file.txt` viene trasformato in `nome_file.vcd` cosicché possa essere analizzato da GTKWave (in questo esempio si è supposto che `nome_file.txt` sia nella directory dove è presente `sched_switch` e `nome_file.vcd` venga posto in questa stessa cartella, altrimenti bisogna specificare i percorsi completi dei file).
Un file VCD è un file ASCII che contiene un'intestazione, definizioni di variabili e i cambi di valore per tutte le variabili.
Ottenuto il ".vcd" si è pronti a lanciare GTKWave. Da una shell si scriva:

```
$ gtkwave nome_file.vcd
```

Appare una finestra simile alla seguente:

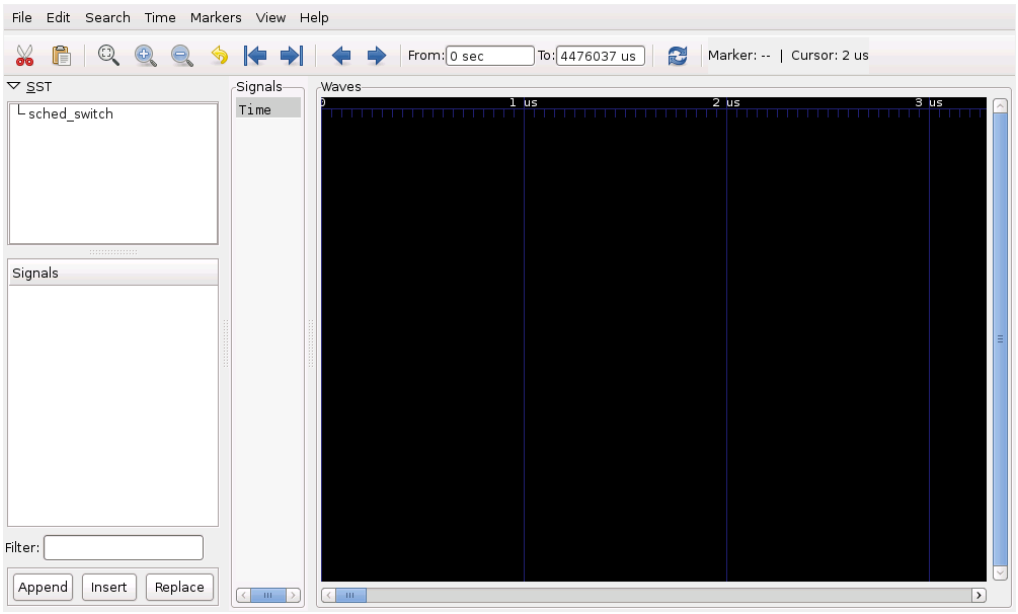

Illustrazione 7: Schermata iniziale di GTKWave

Cliccando su `sched_switch`, sotto la voce `SST`, appariranno tutti i processi, sotto forma di segnale da inserire, tracciati da ftrace e salvati nel ".txt" convertito.

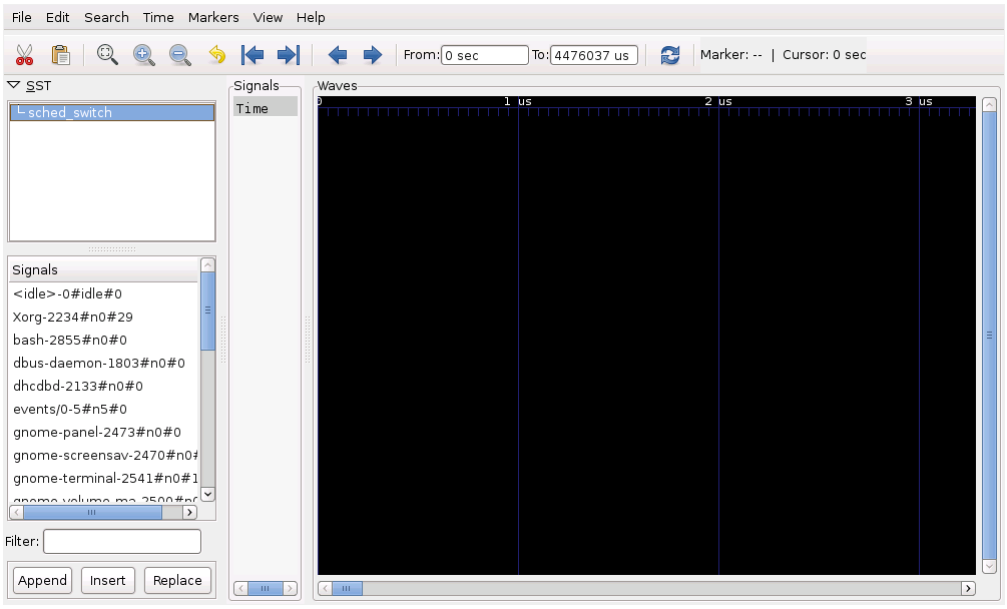

Illustrazione 8: Schermata di GTKWave con i segnali (processi)

Sotto alla lista dei task ci sono tre bottoni: `Append`, `Insert` e `Replace`. Il primo accoda un segnale (processo) sotto a quelli già inseriti nello spazio di visualizzazione, il secondo inserisce un segnale sotto a quello selezionato e il terzo sostituisce un segnale con un altro. Lo spazio dove vengono rappresentate le forme d'onda è quello sulla destra con sfondo nero e in alto ha una barra

temporale.
Le tre icone rappresentanti delle lenti sono pulsanti utili per lavorare con GTKWave:

1. la prima mostra tutto l'arco temporale di analisi in una schermata non ingrandita;
2. la seconda aumenta lo zoom;
3. la terza diminuisce lo zoom.

Avviato GTKWave si inseriscono i processi che si vogliono vedere e ci si può spostare temporalmente nell'asse concentrandosi nei punti di maggiore interesse, usando, oltre alle lenti, le frecce blu. Quelle con la barra verticale davanti mostrano, rispettivamente, l'istante di inizio e quello di fine dell'asse, invece, le altre servono per scorrere sulle commutazioni e sveglie che subisce un task selezionato nell'area `Signals`. Per l'utilizzo delle frecce serve che il campo `Marker` abbia un valore preciso: lo si ottiene cliccando nello spazio nero in corrispondenza dell'istante di tempo considerato di partenza. Da qui si può andare all'inizio o alla fine dell'asse, oppure se è scelto un processo ci si sposta al suo precedente o successivo punto critico (switch o sveglia).
In questa breve descrizione dello strumento si sono fornite delle nozioni base su cosa è e su come si usa relativamente ai file con estensione ".vcd", gli altri tipi di file non sono stati trattati.
Immagini con i processi inseriti saranno fornite nei capitoli successivi, dove si spiegheranno, anche, i significati dei colori usati da questo tool per visualizzare un tracciamento dei task.

6 Necessità dell'integrazione

In questo capitolo viene discusso l'utilizzo di ftrace nel sistema operativo RTAI. RTAI, una volta installato, è un insieme di moduli che possono essere caricati. Un modulo in Linux è un file con estensione ".ko" che deriva da un normale ".c" compilato adeguatamente. Per l'inserimento è usato il comando `insmod` e per la rimozione `rmmod`. Caricato un modulo, questo entra a far parte integrante del kernel. Ulteriore cosa da notare sono le dipendenze: in alcuni casi un modulo necessita di un altro modulo per poter funzionare, cioè non può essere caricato finché non viene caricato quello da cui dipende.
Esempio:
dati due moduli `mod1.ko` e `mod2.ko`, il secondo dipende dal primo, per inserirli bisogna procedere così:

```
# insmod mod1.ko
# insmod mod2.ko
```

la rimozione, invece, deve avvenire in ordine inverso:

```
# rmmod mod2
# rmmod mod1
```

Per usare RTAI il modulo che deve essere caricato per primo è `rtai_hal.ko` e costituisce l'Hardware Abstraction Layer creato da questo sistema. Poi deve seguire lo scheduler real time, perché quello di Linux non va bene per gestire i task RTAI. Ne sono messi a disposizione due a seconda di che tipo di task si vuole far eseguire nel sistema:

- `rtai_sched.ko` per i task che girano in kernel space, ovvero quelli creati con i moduli;
- `rtai_lxrt.ko` per i task user space.

In questo modo la base è inserita e RTAI può essere usato compatibilmente alle esigenze dell'utente: esecuzione di processi real time creati appositamente, IPC (Inter Process Communication), ecc.
Sorge una domanda: inserita la base RTAI e creati dei task real time, è possibile tracciarli con `sched_switch` di ftrace? Per rispondere si è dovuto svolgere delle prove: costruire dei sorgenti che allocassero dei task user space o kernel space, inserire lo scheduler opportuno di RTAI e attivare il tracer `sched_switch`. Nei prossimi due paragrafi saranno descritte le verifiche eseguite per cercare di rispondere al quesito.

6.1 Verifica user space

L'applicativo di test ha come obiettivo verificare se ftrace riesce a tracciare commutazioni e sveglie di processi RTAI user space. La prova è costruita su un meccanismo di IPC. Viene inserita una mailbox (casella di posta) con un modulo del kernel, la quale è usata da due task, `sender` e `receiver`, come

mezzo comunicativo. `sender` invia un messaggio alla volta sulla casella e `receiver` lo preleva.
Nel dettaglio sono stati scritti tre file ".c": quello della mailbox, quello del task inviante e quello del task ricevente. È stato compilato e caricato il modulo, inserito l'indirizzo di memoria della mailbox nei file dei processi (`sender` e `receiver` devono essere a conoscenza di dove si trova, fisicamente, la casella), creati gli eseguibili e lanciati su due terminali distinti. Il codice dei due task prevede un ciclo infinito dove `sender`, ad ogni iterazione, manda un messaggio e, invece, `receiver` lo riceve. È, anche, stato attivato ftrace con `sched_switch`. Lasciati passare alcuni secondi l'esecuzione dei task è stata interrotta e anche ftrace è stato fermato. Visualizzato il file `trace`, con una rapida ricerca, si è notato che i task `sender` e `receiver` comparivano, ossia erano stati visti da ftrace.
Risultato: i task user space sono tracciabili con `sched_switch`.
I moduli di RTAI usati per il test sono:

1. `rtai_hal.ko`, HAL;
2. `rtai_lxrt.ko`, scheduler;
3. `rtai_sem.ko`, necessario per `rtai_mbx.ko`;
4. `rtai_mbx.ko`, senza di questo il modulo della mailbox non poteva essere caricato.

Uno schema illustra la prova eseguita:

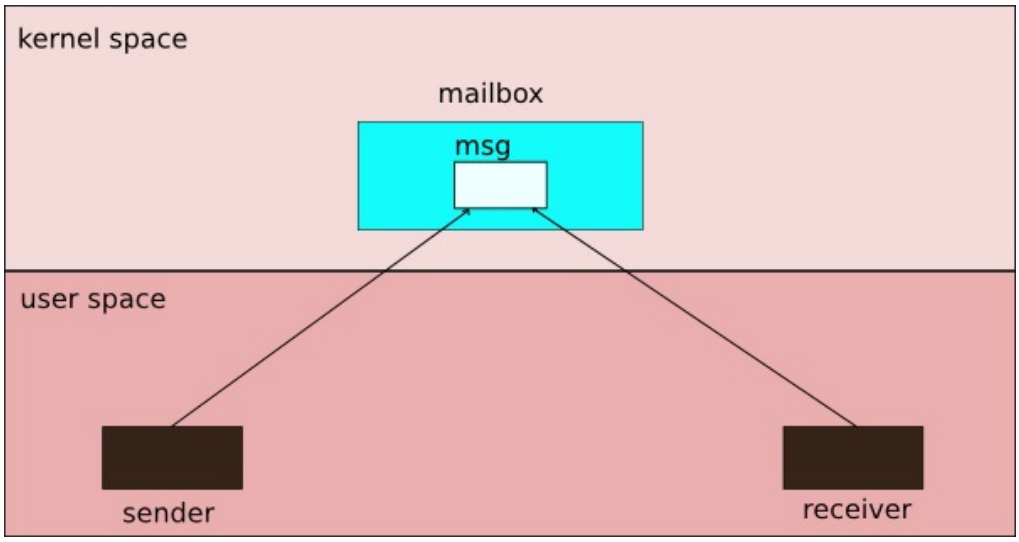

Illustrazione 9: Prova della verifica user space

Si nota l'accesso del `sender` alla mailbox per riporre il messaggio che sarà prelevato dal `receiver`. Il `sender` inserisce un messaggio alla volta e finché il `receiver` non lo legge non può inviare il successivo.
Vengono presentati il tracciato testuale e la visione grafica di questo test:

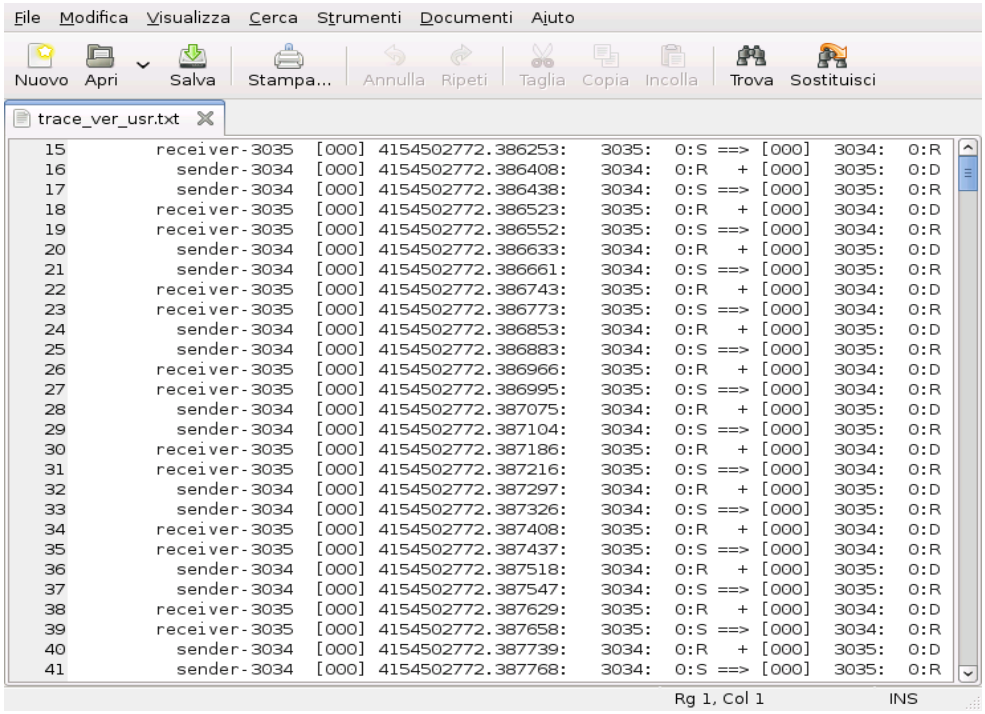

Illustrazione 10: Risultato testuale della verifica user space

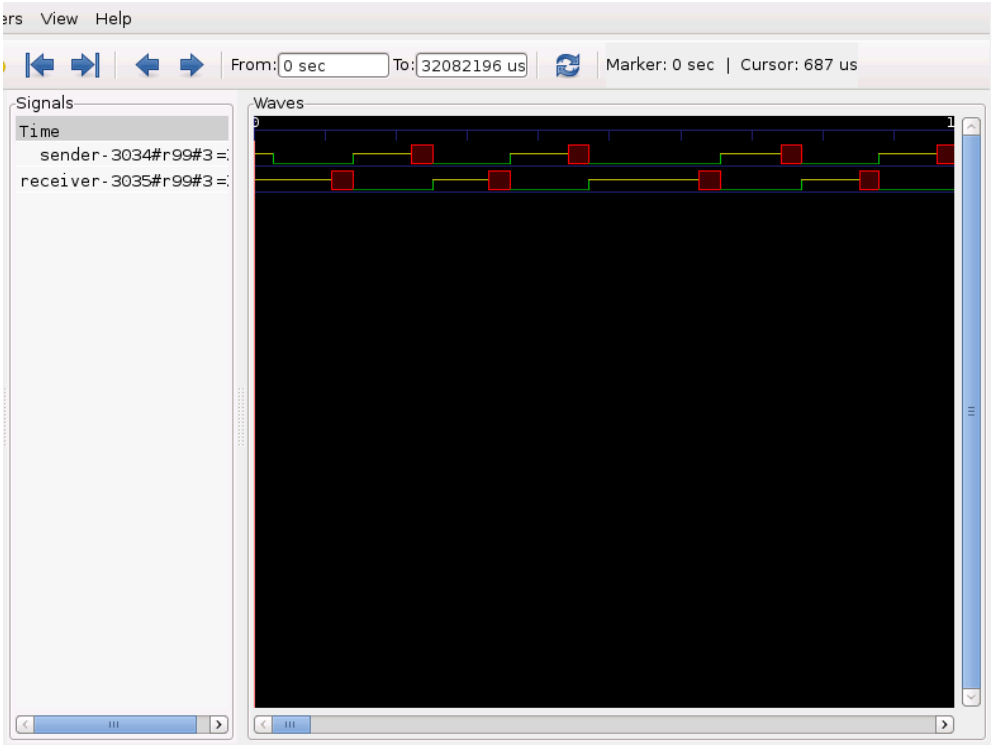

Illustrazione 11: Risultato grafico della verifica user space: solo task `sender` e `receiver`

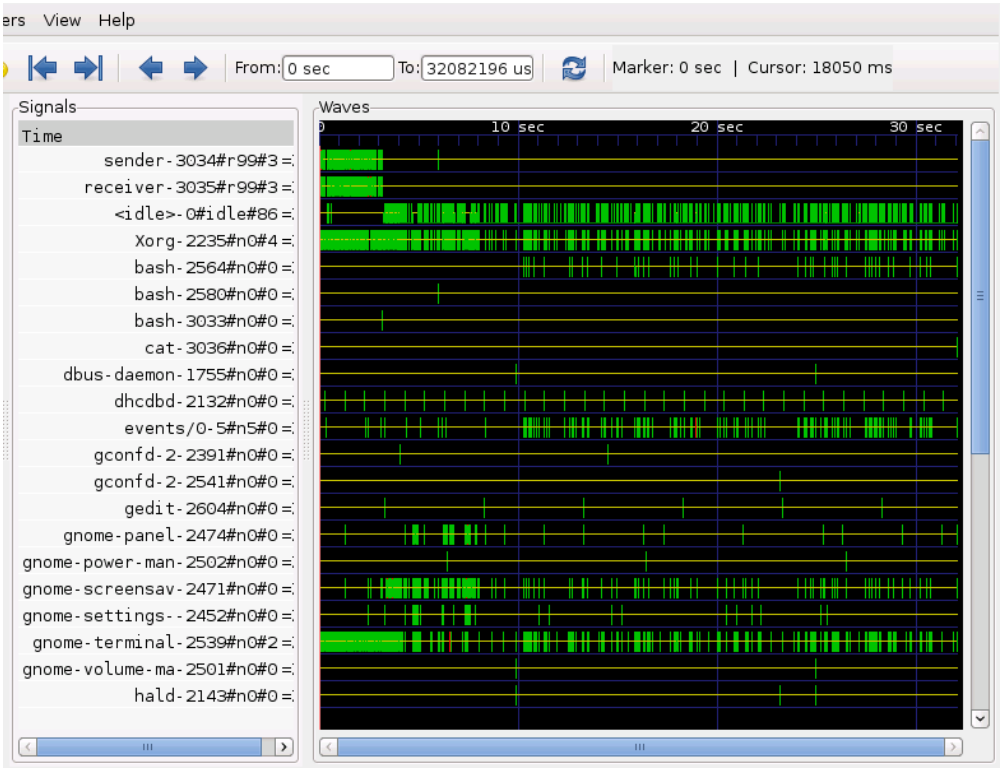

Illustrazione 12: Risultato grafico della verifica user space: tutti i task

Nell'immagine del file ".txt" si notano gli switch tra `sender` e `receiver` e gli istanti di sveglia dopo un periodo di sospensione.
In GTKWave sono stati inseriti i due segnali `sender` e `receiver` che corrispondono ai processi e, nello spazio riservato alle forme d'onda, si vede come si alternano l'uso della CPU. I segnali sono contraddistinti da, sostanzialmente, tre colori: il valore giallo sta ad indicare che il task è sospeso, non sta eseguendo e non ha il controllo del processore, il colore verde indica, invece, l'esecuzione del processo rappresentato dal segnale e il blocco rosso significa che il task si è svegliato da una sospensione, ma non sta ancora eseguendo (l'istante di inizio del blocco indica la sveglia e l'istante di fine segnala la commutazione). Nella prima immagine di GTKWave sono resi evidenti questi colori per il `sender` e il `receiver`. Si nota, ad esempio, che all'inizio il `sender` è verde (sta eseguendo), mentre il `receiver` è giallo (sospeso), finché non si sveglia e attende che il `sender` finisca (blocco rosso). Al termine del `sender` avviene il cambio di contesto, lui diventa sospeso e il `receiver` va in esecuzione e così via. Nella seconda immagine di GTKWave è riportato l'intero arco temporale con tutti i task tracciati (`sender` e `receiver` interagiscono, anche, con altri processi).

6.2 Verifica kernel space

Questo test, invece, si propone di controllare la tracciabilità dei processi RTAI kernel space. La prova è molto semplice: consiste nell'inserire un modulo che crea un task, il quale esegue una semplice attività.

I moduli RTAI sono:

1. `rtai_hal.ko`;
2. `rtai_sched.ko`, scheduler.

Scritto il file di test, è stato compilato come modulo e caricato. Si è attivato ftrace, lasciandolo tracciare per alcuni secondi. Poi, dopo averlo fermato, è stato stampato il file `trace` e, in questo caso, il task kernel space, allocato dal modulo, non appariva.
Risultato: ftrace non traccia i task RTAI kernel space.
Nei risultati testuali e grafici questo processo non compare e, quindi, non sono stati riportati.

6.3 Considerazioni

Le due verifiche precedenti forniscono una risposta alla domanda di inizio capitolo: inserita la base RTAI e creati dei task real time, è possibile tracciarli con `sched_switch` di ftrace?
Lo strumento riesce a fare il tracing dei soli processi user space. Questo risultato non è soddisfacente in quanto si vuole, anche, avere la possibilità di osservare i task che girano in kernel space.
Ftrace deve entrare in contatto con i processi di RTAI, in modo che li possa vedere per poterli segnalare sul ring buffer e renderli, da qui, visibili anche all'utente, quando deciderà di stampare il resoconto del tracciamento (file `trace`).
Si rende necessaria un'integrazione del tracer `sched_switch` in RTAI.

7 Lavoro di integrazione

Come indicato nel capitolo 6, i processi RTAI user space vengono visti da ftrace, ma non vengono tracciati quelli kernel space. La tesi vuole, però, puntare ad un controllo di tutti i tipi di task; quindi, occorre agire a basso livello, sul codice sorgente di RTAI, per completare la funzionalità di tracing realizzata da ftrace.
Il tracer sched_switch, per registrare una commutazione di contesto, usa una funzione particolare, `trace_sched_switch`, la quale chiama la, già citata, `probe_sched_switch` spiegata nell'analisi. I parametri d'ingresso sono una run queue, il task precedente e quello successivo (gli stessi di `probe_sched_switch`). I context switch, in un sistema operativo, sono gestiti dallo scheduler, il quale amministra i task facendoli eseguire nel tempo in base alla priorità. Interviene, quindi, quando scade la “fetta” di tempo riservata per un processo: salva i suoi dati (contesto) e assegna la CPU ad un altro (switch). Può essere chiamato in causa, anche, quando il sistema riceve un interrupt che deve essere gestito dalla sua routine (ISR, Interrupt Service Routine). Si intuisce che, in Linux, ftrace usa `trace_sched_switch` nello scheduler, nel file che lo rappresenta, ovvero `sched.c`. Infatti, questa funzione è invocata all'interno di un'altra che esegue, appunto, la commutazione da un processo al successivo: ftrace salva la commutazione, lo scheduler la esegue.
sched_switch svolge, anche, un altro compito, ossia il tracing delle sveglie dei task: la funzione usata è `trace_sched_wakeup` e, al suo interno, chiama `probe_sched_wakeup` (vuole i suoi stessi parametri: una run queue, il processo svegliato e un intero). La sveglia indica che un task ritorna ad essere pronto per l'esecuzione, dopo una sospensione dovuta, per esempio, ad una attesa su un semaforo (mezzo di sincronizzazione offerto dal sistema operativo) per entrare in una sezione critica, oppure ad una impossibilità a continuare la sua esecuzione per la mancanza di un evento esterno. Se un processo è nello stato ready (pronto), lo scheduler deve tenerlo presente, perché potrebbe essere richiesto, anche in questo caso, il suo intervento. Infatti se il task ha una priorità superiore rispetto a quello corrente, lui ha il diritto di ricevere il processore, a meno che non sia disabilitata la preemption: questo significa che può essere necessario uno switch. Il posto di inserimento di `trace_sched_wakeup` è, ovviamente, il file `sched.c`: è posizionata per permettere ad ftrace di tracciare le sveglie prima che queste avvengono.
In realtà, esiste nel kernel un'altra funzione che si occupa di svegliare processi. Il suo nome è `trace_sched_wakeup_new`: è molto simile a quella appena vista, anche lei chiama `probe_sched_wakeup` e, come il nome fa capire, sveglia task nuovi (appena creati).
Tornando all'ambiente RTAI, l'idea è quella di inserire le chiamate a `trace_sched_switch` e `trace_sched_wakeup` (viene usata solo questa funzione di sveglia) nel suo scheduler. Il file, anche qua, si chiama `sched.c` e da questo vengono creati entrambi i moduli `rtai_sched.ko` e `rtai_lxrt.ko`. Le due questioni da risolvere sono identificare le funzioni che realizzano lo switch e le sveglie in RTAI. All'interno di `rt_schedule`, funzione di

scheduling, è chiamata `switch_rtai_tasks`. Questa tratta gli scambi tra processi kernel space, quindi, è una funzione di interesse per l'integrazione.

```
if (!(new_task = switch_rtai_tasks(rt_current, new_task, cpuid))) {
      goto sched_exit;
}
```

Per le sveglie, sempre in `rt_schedule`, è invocata `wake_up_timed_tasks`: seconda funzione di interesse.

```
wake_up_timed_tasks(cpuid);
```

Individuate queste funzioni, bisogna, quindi, far chiamare `trace_sched_switch` e `trace_sched_wakeup` prima che avvenga il cambio e la sveglia in modo da segnalare l'evento sul ring buffer, così, quando il tracing è concluso, nel file `trace`, insieme agli switch e sveglie di task Linux, appariranno switch e sveglie RTAI.
Ragioniamo sui task. In Linux esiste la struttura dati `task_struct` che rappresenta il descrittore di un processo, contiene campi importanti come il pid, lo stato, la priorità. sched_switch utilizza, nelle sue funzioni, puntatori a questo tipo di struct per tracciare i task. Da qui si capisce che `trace_sched_switch` e `trace_sched_wakeup` vogliono indirizzi di `task_struct` (la cosa era evidente visto che richiedono gli stessi parametri di `probe_sched_switch` e `probe_sched_wakeup`, rispettivamente). In RTAI, però, esiste un'altra struttura per descrivere i processi real time, non viene usata la stessa `task_struct`. Qui è presente `RT_TASK` come descrittore: un task RTAI è identificato da questa struct.
Nasce, quindi, un'incompatibilità, perché non si possono passare puntatori a `RT_TASK` in `trace_sched_switch` e `trace_sched_wakeup`.
Prima di chiudere questa introduzione all'integrazione, vengono spiegate `switch_rtai_tasks` e `wake_up_timed_tasks`.

```
1.  static RT_TASK *switch_rtai_tasks(RT_TASK *rt_current,
    RT_TASK *new_task, int cpuid)
2.  {
3.        if (rt_current->lnxtsk) {
4.              unsigned long sflags;
5.  #ifdef IPIPE_NOSTACK_FLAG
6.              ipipe_set_foreign_stack(&rtai_domain);
7.  #endif
8.              SAVE_LOCK_LINUX(cpuid);
9.              rt_linux_task.prevp = rt_current;
10.             save_fpcr_and_enable_fpu(linux_cr0);
11.             if (new_task->uses_fpu) {
12.                   save_fpenv(rt_linux_task.fpu_reg);
13.                   fpu_task = new_task;
14.                   restore_fpenv(fpu_task->fpu_reg);
15.             }
16.             RST_EXEC_TIME();
```

```
17.             SAVE_PREV_TASK();
18.             rt_exchange_tasks(rt_smp_current[cpuid],
    new_task);
19.             restore_fpcr(linux_cr0);
20.             RESTORE_UNLOCK_LINUX(cpuid);
21. #ifdef IPIPE_NOSTACK_FLAG
22.             ipipe_clear_foreign_stack(&rtai_domain);
23. #endif
24.             if (rt_linux_task.nextp != rt_current) {
25.                   return rt_linux_task.nextp;
26.             }
27.       } else {
28.             if (new_task->lnxtsk) {
29.                   rt_linux_task.nextp = new_task;
30.                   new_task = rt_linux_task.prevp;
31.                   if (fpu_task != &rt_linux_task) {
32.                         save_fpenv(fpu_task->fpu_reg);
33.                         fpu_task = &rt_linux_task;
34.                         restore_fpenv(fpu_task->fpu_reg);
35.                   }
36.             } else if (new_task->uses_fpu && fpu_task !=
    new_task) {
37.                   save_fpenv(fpu_task->fpu_reg);
38.                   fpu_task = new_task;
39.                   restore_fpenv(fpu_task->fpu_reg);
40.             }
41.             SET_EXEC_TIME();
42.             SAVE_PREV_TASK();
43.             rt_exchange_tasks(rt_smp_current[cpuid],
    new_task);
44.       }
45.       RTAI_TASK_SWITCH_SIGNAL();
46.       return NULL;
47. }
```

Per questa funzione possono passare due tipi di processi:

1. task RTAI nativo: creato con un modulo del kernel e schedulato con `rtai_sched.ko`;
2. task di tipo Linux: inglobato all'interno di un processo RTAI.

La routine chiede in ingresso i puntatori (`RT_TASK`) al processo corrente e nuovo (il sostituto) e l'indice della CPU corrente.
Il suo comportamento dipende in base a che tipo di task sono passati per lo switch:

1. nel caso siano entrambi task RTAI nativi si è nella situazione più elementare perché viene, semplicemente, effettuato il cambio di contesto dei registri macchina (viene, quindi, sostituito, anche, il puntatore allo stack); `switch_rtai_tasks` ritornando `NULL` consentirà al task chiamante, una volta che sarà riavviato, di uscire immediatamente da `rt_schedule`;
2. se il primo processo è di tipo Linux e il secondo RTAI nativo viene

salvato nel campo `prevp` del descrittore `RT_TASK` di Linux il puntatore al processo Linux corrente (primo argomento), questo sarà utile non appena si dovrà ritornare a processi Linux (da un task RTAI nativo ad un processo Linux), la routine, in questo caso, ritornerà il puntatore al prossimo processo Linux verso cui fare lo switch, obbligando `rt_schedule` a non uscire, ma effettuare un ulteriore switch tra due task Linux;

3. nel caso, infine, il primo processo sia RTAI e il secondo Linux viene salvato il puntatore a quest'ultimo nel campo `nextp` del descrittore `RT_TASK` di Linux e viene fatto lo switch rispetto al processo salvato nel campo `prevp`; lo switch farà, così, riprendere il task salvato in `prevp` da una situazione del tipo vista in 2 che richiede di eseguire i successivi passi di switch tra due task Linux che richiedono lo scambio delle tabelle di MMU (Memory Management Unit), se i due task sono diversi.

Una volta che il task finisce sulla funzione `rt_exchange_tasks` non viene più eseguito a favore di un altro. Dato che Linux parte prima di RTAI un tipo di switch RTAI-Linux può accadere solo se prima ci sia stato uno switch Linux-RTAI, perciò il punto 3 può accadere solo se prima c'è stato un punto 2.

```
1.  static inline void wake_up_timed_tasks(int cpuid)
2.  {
3.        RT_TASK *taskh, *task;
4.  #ifdef CONFIG_SMP
5.        task = (taskh = &rt_smp_linux_task[cpuid])->tnext;
6.  #else
7.        task = (taskh = &rt_smp_linux_task[0])->tnext;
8.  #endif
9.        if (task->resume_time <= rt_time_h) {
10.             do {
11.                       if ((task->state &=
    ~(RT_SCHED_DELAYED | RT_SCHED_SUSPENDED |
    RT_SCHED_SEMAPHORE | RT_SCHED_RECEIVE | RT_SCHED_SEND |
    RT_SCHED_RPC | RT_SCHED_RETURN | RT_SCHED_MBXSUSP |
    RT_SCHED_POLL)) == RT_SCHED_READY) {
12.                            if (task->policy < 0) {
13.
    enq_ready_edf_task(task);
14.                                } else {
15.                                      enq_ready_task(task);
16.                            }
17. #if defined(CONFIG_RTAI_BUSY_TIME_ALIGN) &&
    CONFIG_RTAI_BUSY_TIME_ALIGN
18.                           task->busy_time_align =
    oneshot_timer;
19. #endif
20.                       }
21.                   rb_erase_task(task, cpuid);
22.                   task = task->tnext;
23.             } while (task->resume_time <= rt_time_h);
```

```
#ifdef CONFIG_SMP
            rt_smp_linux_task[cpuid].tnext = task;
            task->tprev = &rt_smp_linux_task[cpuid];
#else
            rt_smp_linux_task[0].tnext = task;
            task->tprev = &rt_smp_linux_task[0];
#endif
      }
}
```

La routine vuole, come input, l'indice della CPU corrente.
Il suo compito è quello di aggiungere alla “Ready List” (lista dei processi RTAI che possono andare in esecuzione senza particolari proprietà) tutti i task in scadenza entro `rt_time_h` accodati sulla “Timed List” (lista dei processi RTAI che hanno scadenze temporali), rimuovendoli da quest'ultima.
Queste spiegazioni saranno utili per il paragrafo successivo, quando, risolta l'incompatibilità riscontrata prima, si dovranno inserire le azioni di tracing in `switch_rtai_tasks` e `wake_up_timed_tasks`.

7.1 Funzione di conversione

L'incompatibilità è stata risolta costruendo una conversione tra i tipi di processi. Come, già, osservato `trace_sched_switch` e, anche, `trace_sched_wakeup` vogliono `task_struct`, quindi, tramite un'apposita routine, si converte un `RT_TASK` in un `task_struct`. Le informazioni dei processi richieste da sched_switch per una commutazione e una sveglia sono il **nome**, il **pid**, lo **stato** e la **priorità**. È sufficiente prelevarle dal descrittore RTAI e inserirle in uno Linux, per poi passarlo a `trace_sched_switch` o `trace_sched_wakeup`. La funzione inserita in RTAI è stata chiamata `task_conv`:

```
static struct task_struct trace_Linux, trace1_RTAI, trace2_RTAI;
static int process_id=10000;

static void task_conv (RT_TASK *rt_task, struct task_struct *tsk) {
      int i;
      if (rt_task->lnxtsk) {
            tsk->prio=rt_task->lnxtsk->prio;
            tsk->pid=rt_task->lnxtsk->pid;
            for (i=0; i<TASK_COMM_LEN; ++i)
                  tsk->comm[i]=rt_task->lnxtsk->comm[i];
      }
      else {
            tsk->prio=rt_task->base_priority;
            if (!rt_task->tid)
                  tsk->pid=rt_task->tid=++process_id;
```

```
16.             else
17.                   tsk->pid=rt_task->tid;
18.             tsk->comm[0]='T';
19.             tsk->comm[1]='S';
20.             tsk->comm[2]='K';
21.       }
22.       if (rt_task->state==RT_SCHED_READY)
23.             tsk->state=0;
24.       else
25.             tsk->state=1;
26. }
```

Fuori dalla funzione sono dichiarate quattro variabili: tre sono i task di tipo `task_struct` che servono per la conversione, `trace_Linux`, `trace1_RTAI` e `trace2_RTAI` (usate in `switch_rtai_tasks`), invece l'altra è un intero che serve per assegnare progressivamente i pid ai processi RTAI. I task RTAI sono identificati dal puntatore al loro descrittore `RT_TASK`, ma, per facilità di gestione, invece di convertire questo indirizzo in intero senza segno e usarlo come pid, è stato scelto di crearne uno nuovo. Il primo pid sarà 10001, numero volutamente grande in modo che non vada in conflitto con i pid di task Linux (righe 1 e 2).
La funzione richiede in input il puntatore al task RTAI da convertire e l'indirizzo del task Linux che dovrà contenere le informazioni del descrittore `RT_TASK`. Non ritorna nulla.
Variabile locale, riga 5: `i` è usata in un ciclo come indice di un vettore.
Il processo puntato da `rt_task` può essere di tipo Linux o RTAI nativo.
Nel primo caso significa che il campo `lnxtsk` non è `NULL`, quindi, si salva la priorità, il pid e il nome (vettore `comm`) di questo task nella struttura puntata da `tsk` (dalla riga 6 alla 11).
Nella seconda situazione si sta considerando un processo RTAI nativo. Si memorizza la priorità. Il pid viene assegnato sia al task RTAI, usando il campo `tid`, sia a quello Linux indirizzato da `tsk`, usando il campo `pid`. Se `rt_task->tid` è zero vuol dire che questo processo è la prima volta che “passa” per questa funzione (un task RTAI può diventare parametro diverse volte): si incrementa `process_id` e lo si assegna a `tsk->pid` e a `rt_task->tid`. In caso contrario, il task è, già, “passato” per `task_conv`, si copia il suo pid in `tsk->pid`. Serve, poi, dare un nome al task convertito, visto che quello RTAI nativo non ce l'ha. Del vettore `comm` sono usate le prime tre celle. Ad ogni processo è assegnato, sempre, lo stesso nome e la distinzione avviene per mezzo del pid e dell'ordine in cui i moduli che creano i task sono inseriti: il primo modulo avrà un processo con nome *TSK* e pid *10001*, il secondo, nome *TSK* e pid *10002* e avanti così (dalla riga 12 alla 21).
Un processo RTAI è pronto per l'esecuzione se il suo stato è pari a `RT_SCHED_READY`, che è una costante di valore uno, invece un task Linux è pronto se il suo stato è a zero. Questo è il significato dell'`if`. L'`else` sta ad indicare che il processo RTAI non è pronto. In Linux si traduce con uno stato positivo. Per scelta viene posto uguale ad uno (dalla riga 22 alla 25).
La complessità computazionale è O(1), perché il numero di istruzioni eseguite ogni volta che viene chiamata `task_conv` è costante. Si noti che il ciclo `for`

non dipende da una variabile, ma da una costante, quindi, il numero di iterazioni è sempre lo stesso, quando viene eseguito.
Spiegato il codice, ora bisogna trovare i punti di inserimento della funzione di conversione in `switch_rtai_tasks`. Nel suo interno il vero scambio è realizzato da `rt_exchange_tasks` (righe 18 e 43), quindi, `trace_sched_switch` può essere messa prima di essa, preceduta a sua volta da `task_conv`. Lo scambio avviene tra i parametri `rt_current` e `new_task`, dove, almeno, uno dei due è un processo RTAI nativo. Prima di effettuare lo switch, si convertono i due task con `task_conv` (da `RT_TASK` a `task_struct`). I processi ottenuti sono passati come parametri a `trace_sched_switch` (tracciamento della commutazione) e, dopo, avviene il cambio.
In `wake_up_time_tasks` la parte che si occupa di svegliare il task è contenuta nell'`if` dalla riga 12 alla 16. `trace_sched_wakeup` può essere messa prima di questo `if`, dopo l'inserimento di `task_conv`, la quale convertirà il processo che sarà svegliato. Si passerà, anche qui, un puntatore a `task_struct`, per tracciare la sveglia immediatamente prima che avvenga.
Finora si è discusso solo dei task, ma `trace_sched_switch` e `trace_sched_wakeup` richiedono, anche, un altro parametro, l'indirizzo di una run queue. Si nota, esaminando rapidamente il codice, che questo puntatore non è mai usato, quindi, può essere assegnato a `NULL`. Inoltre `trace_sched_wakeup` vuole, come terzo parametro, un intero che, anche questo, non è usato. Viene impostato a 1.
Segue `switch_rtai_tasks` con le aggiunte del tracing:

```
1.  static RT_TASK *switch_rtai_tasks(RT_TASK *rt_current,
    RT_TASK *new_task, int cpuid)
2.  {
3.
4.        //Modifiche allo scheduler di RTAI
5.        struct rq *rq=0;
6.
7.        if (rt_current->lnxtsk) {
8.              unsigned long sflags;
9.  #ifdef IPIPE_NOSTACK_FLAG
10.             ipipe_set_foreign_stack(&rtai_domain);
11. #endif
12.             SAVE_LOCK_LINUX(cpuid);
13.             rt_linux_task.prevp = rt_current;
14.             save_fpcr_and_enable_fpu(linux_cr0);
15.             if (new_task->uses_fpu) {
16.                   save_fpenv(rt_linux_task.fpu_reg);
17.                   fpu_task = new_task;
18.                   restore_fpenv(fpu_task->fpu_reg);
19.             }
20.             RST_EXEC_TIME();
21.             SAVE_PREV_TASK();
22.
23.             task_conv (rt_current, &trace_Linux);
```

```
24.             task_conv (new_task, &trace1_RTAI);
25.             trace_sched_switch      (rq,      &trace_Linux,
    &trace1_RTAI);
26.
27.             rt_exchange_tasks(rt_smp_current[cpuid],
    new_task);
28.             restore_fpcr(linux_cr0);
29.             RESTORE_UNLOCK_LINUX(cpuid);
30. #ifdef IPIPE_NOSTACK_FLAG
31.             ipipe_clear_foreign_stack(&rtai_domain);
32. #endif
33.             if (rt_linux_task.nextp != rt_current) {
34.                   return rt_linux_task.nextp;
35.             }
36.       } else {
37.             if (new_task->lnxtsk) {
38.                   rt_linux_task.nextp = new_task;
39.                   new_task = rt_linux_task.prevp;
40.                   if (fpu_task != &rt_linux_task) {
41.                         save_fpenv(fpu_task->fpu_reg);
42.                         fpu_task = &rt_linux_task;
43.                         restore_fpenv(fpu_task->fpu_reg);
44.                   }
45.             } else if (new_task->uses_fpu && fpu_task !=
    new_task) {
46.                   save_fpenv(fpu_task->fpu_reg);
47.                   fpu_task = new_task;
48.                   restore_fpenv(fpu_task->fpu_reg);
49.             }
50.             SET_EXEC_TIME();
51.             SAVE_PREV_TASK();
52.
53.             task_conv               (rt_smp_current[cpuid],
    &trace1_RTAI);
54.             if (new_task->lnxtsk) {
55.                   task_conv (new_task, &trace_Linux);

56.                   trace_sched_switch   (rq,   &trace1_RTAI,
    &trace_Linux);
57.             }
58.             else {
59.                   task_conv (new_task, &trace2_RTAI);
60.                   trace_sched_switch   (rq,   &trace1_RTAI,
    &trace2_RTAI);
61.             }
62.
63.             rt_exchange_tasks(rt_smp_current[cpuid],
    new_task);
64.       }
65.       RTAI_TASK_SWITCH_SIGNAL();
```

```
66.       return NULL;
67. }
```

Definizione del puntatore alla run queue (riga 5).
`rt_current`, processo di tipo Linux e `new_task`, task RTAI nativo (dalla riga 23 alla 25).
`rt_smp_current[cpuid]`, processo RTAI nativo e `new_task`, task di tipo Linux, oppure due processi RTAI nativi (dalla riga 53 alla 61).
`rt_smp_current[cpuid]` è l'elemento che identifica il task RTAI in esecuzione sul processore di indice `cpuid` e coincide con `rt_current`.
Ora viene riportato il codice con il tracing di `wake_up_timed_tasks`:

```
1.  static struct task_struct wake_task;
2.
3.  static inline void wake_up_timed_tasks(int cpuid)
4.  {
5.        RT_TASK *taskh, *task;
6.
7.        //Modifiche allo scheduler di RTAI
8.        struct rq *rq=0;
9.
10. #ifdef CONFIG_SMP
11.       task = (taskh = &rt_smp_linux_task[cpuid])->tnext;
12. #else
13.       task = (taskh = &rt_smp_linux_task[0])->tnext;
14. #endif
15.       if (task->resume_time <= rt_time_h) {
16.             do {
17.                   if ((task->state &=
    ~(RT_SCHED_DELAYED | RT_SCHED_SUSPENDED |
    RT_SCHED_SEMAPHORE | RT_SCHED_RECEIVE | RT_SCHED_SEND |
    RT_SCHED_RPC | RT_SCHED_RETURN | RT_SCHED_MBXSUSP |
    RT_SCHED_POLL)) == RT_SCHED_READY) {
18.
19.                         task_conv (task, &wake_task);
20.                         trace_sched_wakeup (rq,
    &wake_task, 1);
21.
22.                         if (task->policy < 0) {
23.
    enq_ready_edf_task(task);
24.                               } else {
25.                                     enq_ready_task(task);
26.                           }
27. #if defined(CONFIG_RTAI_BUSY_TIME_ALIGN) &&
    CONFIG_RTAI_BUSY_TIME_ALIGN
28.                         task->busy_time_align =
    oneshot_timer;
29. #endif
30.                   }
```

```
31.                     rb_erase_task(task, cpuid);
32.                     task = task->tnext;
33.             } while (task->resume_time <= rt_time_h);
34. #ifdef CONFIG_SMP
35.             rt_smp_linux_task[cpuid].tnext = task;
36.             task->tprev = &rt_smp_linux_task[cpuid];
37. #else
38.             rt_smp_linux_task[0].tnext = task;
39.             task->tprev = &rt_smp_linux_task[0];
40. #endif
41.         }
42. }
```

Dichiarazione variabile nella riga 1: `wake_task` serve per la trasformazione del processo RTAI che si sveglia.
Puntatore alla run queue (riga 8).
Conversione del task RTAI in uno Linux e successivo tracciamento della sua sveglia (righe 19 e 20).
`task_conv` è la parte fondamentale di questa integrazione, il codice necessario che bisognava aggiungere. Poi, per il tracing, ci si appoggia a qualcosa di, già, sviluppato, ftrace appunto.
A seguire uno schema che riepiloga, graficamente, la conversione.

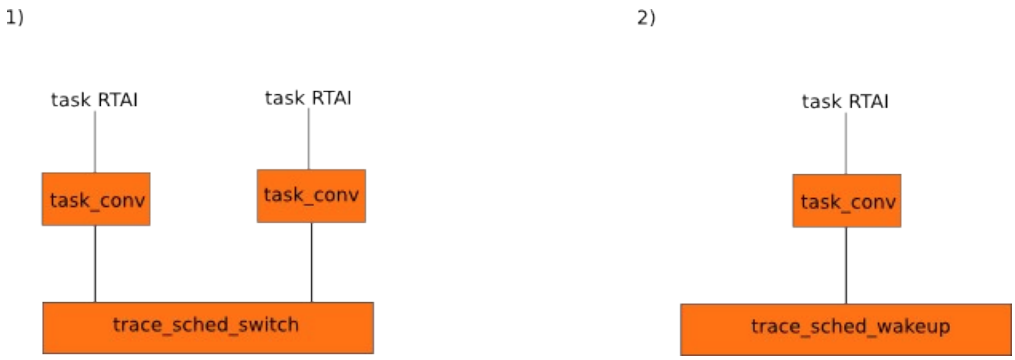

Illustrazione 13: Conversione per il tracciamento di uno switch tra processi (1) e di una sveglia (2)

Le due parti dello schema rappresentano le due conversioni. Nella prima si notano le due chiamate a `task_conv` per trasformare i due processi coinvolti nello switch e il tracciamento dello stesso, subito dopo. Nella seconda è presente una chiamata a `task_conv` (le sveglie sono segnalate una alla volta) seguita dal tracing della sveglia.
Infine, vengono riportati i file di RTAI modificati per questa integrazione: `switch_rtai_tasks` è presente in `sched.c` e il tracciamento delle commutazioni è stato inserito lì dentro, invece, `wake_up_timed_tasks` è implementata in `rtai_schedcore.h` e, quindi, qua si è inserito il tracing delle sveglie. Siccome `sched.c` include `rtai_schedcore.h` (direttiva `#include` del C), la funzione di conversione è stata scritta in quest'ultimo file. Inoltre, sempre qui, deve essere incluso un file ".h" del kernel per poter invocare, da `sched.c`,

`trace_sched_switch` e `trace_sched_wakeup`:

```
#include <trace/sched.h>
```

7.2 Modifica ai sorgenti del kernel

Durante le prove “non ufficiali” del tracing in RTAI (quelle dove i risultati servivano solo come verifica del funzionamento dell'integrazione e non sono stati più considerati), dopo l'inserimento di `task_conv`, si è notato che i processi RTAI, insieme a quelli Linux, venivano, effettivamente, segnalati da ftrace. Il problema stava nel fatto che il nome dei task RTAI riportati sotto la colonna `TASK-PID` della testata del file `trace` era sbagliato. Collegandosi all'esempio fatto nel capitolo 4 sull'utilizzo di sched_switch, in particolare all'immagine, la colonna `TASK-PID` indica il processo correntemente in esecuzione e nella colonna `FUNCTION` <pid>:<prio>:<state> ==> <pid>:<prio>:<state> rappresenta lo switch dal task corrente (quello che, nella stessa riga, appare sotto la colonna `TASK-PID`, identificato dalla terna pid-prio-state posta prima di `==>`) in favore di un nuovo processo (terna dopo `==>`) oppure <pid>:<prio>:<state> + <pid>:<prio>:<state> è una sveglia di un processo (terna pid-prio-state dopo il simbolo +), mentre ne è in esecuzione un altro (quello corrente, terna prima di +). A livello di codice, la commutazione di contesto è tracciata tramite la funzione `tracing_sched_switch_trace`.

```
void tracing_sched_switch_trace(struct trace_array *tr, struct
trace_array_cpu *data, struct task_struct *prev, struct
task_struct *next, unsigned long flags, int pc)
```

La sveglia, invece, viene tracciata da `tracing_sched_wakeup_trace`.

```
void
tracing_sched_wakeup_trace(struct trace_array *tr,
                      struct trace_array_cpu *data,
                      struct task_struct *wakee,
                      struct task_struct *curr,
                      unsigned long flags, int pc)
```

Di queste funzioni se ne è già parlato. Al loro interno c'è una chiamata a `tracing_generic_entry_update` che, tra le varie cose, salva il pid del task corrente, indicato con la variabile `current` di tipo `task_struct`.

```
tracing_generic_entry_update(&entry->ent, flags, pc);
```

La memorizzazione del pid avviene in `entry->ent.pid`, dove si ricorda che `entry` è un puntatore a struct `ctx_switch_entry`.

```
struct ctx_switch_entry *entry;
```

Dopo questa chiamata, segue il tracciamento dello switch tra `prev` e `next` o della sveglia di `wakee` durante l'esecuzione di `current`.
Si deduce, in `tracing_sched_switch_trace`, che `current` e `prev` indirizzano

lo stesso processo (si faccia riferimento al paragrafo 4.8.1). Di conseguenza `current->pid` e `prev->pid` sono uguali (lo stesso vale per priorità e stato), quindi, sono uguali, anche, `entry->ent.pid` e `entry->prev_pid`. Questo vale per il tracing dei soli task Linux. Con i processi RTAI in gioco cambia la situazione: `current` è sempre il task corrente Linux, invece, `prev` (e anche `next`) può essere un processo RTAI convertito con `task_conv`. In questo caso è chiaro che `current` e `prev` sono diversi. Inoltre, il nome che appare in `TASK-PID` viene preso in base al pid del task corrente (più precisamente proprio da `entry->ent.pid`) e da qui si capisce il motivo dello sbaglio: viene preso il nome di `current`, invece di quello di `prev`.
La sveglia è segnalata in maniera simile ad uno switch tra il processo corrente e quello che si è svegliato (c'è + al posto di ==>). In `tracing_sched_wakeup_trace`, vengono salvati i dati (pid, priorità e stato) di `current`: `entry->ent.pid` e `entry->prev_pid` sono sempre uguali, anche se stanno girando task RTAI. Si nota che, anche in questo caso, viene recuperato il nome di `current` e questo è uno sbaglio se `wakee` è un processo RTAI, perché la segnalazione di una sveglia, con questa integrazione, deve avvenire quando c'è in esecuzione un task RTAI e non uno Linux. Inoltre, nella colonna `FUNCTION` risulta sbagliata la prima terna pid-prio-state: è la terna relativa a `current`.
Per le commutazioni, il problema si risolve uguagliando `entry->ent.pid` a `entry->prev_pid` se `prev` è RTAI convertito: in questo modo viene recuperato il suo nome. Si tenga presente che questa modifica è innocua per ftrace (non ne cambia il suo funzionamento) ed è eseguita se ci sono da schedulare processi RTAI.
Per le sveglie la situazione è diversa: devono essere sostituiti i valori di `entry->prev_pid`, `entry->prev_prio` e `entry->prev_state` con quelli del task corrente RTAI convertito (diverso, ovviamente, da `current`). Per fare ciò ci si deve appoggiare a `tracing_sched_switch_trace`: si memorizzano le informazioni di `next` (se è un processo RTAI), il quale sarà il successivo task che andrà in esecuzione e queste rimpiazzeranno quelle di `current`, quando si sveglierà `wakee` (sempre se è un processo RTAI). Deve, anche qui, essere cambiato `entry->ent.pid` per permettere di recuperare il nome del task RTAI corrente.

```
1.  //Modifica al kernel per RTAI
2.  int pid_prev_RTAI=-1, pid_next_RTAI=-1,
    prio_next_RTAI=-1;
3.
4.  void
5.  tracing_sched_switch_trace(struct trace_array *tr,
6.                         struct trace_array_cpu *data,
7.                         struct task_struct *prev,
8.                         struct task_struct *next,
9.                         unsigned long flags, int pc)
10. {
11.         struct ring_buffer_event *event;
12.         struct ctx_switch_entry *entry;
13.         unsigned long irq_flags;
```

```
14.
15.       event = ring_buffer_lock_reserve(tr->buffer,
    sizeof(*entry),
16.                                        &irq_flags);
17.       if (!event)
18.             return;
19.       entry = ring_buffer_event_data(event);
20.       tracing_generic_entry_update(&entry->ent, flags,
    pc);
21.       entry->ent.type                 = TRACE_CTX;
22.       entry->prev_pid                 = prev->pid;
23.       entry->prev_prio        = prev->prio;
24.       entry->prev_state       = prev->state;
25.       entry->next_pid                 = next->pid;
26.       entry->next_prio        = next->prio;
27.       entry->next_state       = next->state;
28.       entry->next_cpu   = task_cpu(next);
29.
30.       //Modifica al kernel per RTAI
31.       if (current!=prev) {
32.             pid_prev_RTAI=entry->ent.pid=entry->prev_pid;
33.             pid_next_RTAI=entry->next_pid;
34.             prio_next_RTAI=entry->next_prio;
35.       }
36.
37.       ring_buffer_unlock_commit(tr->buffer, event,
    irq_flags);
38.       ftrace_trace_stack(tr, data, flags, 5, pc);
39.       ftrace_trace_userstack(tr, data, flags, pc);
40. }
41.
42. void
43. tracing_sched_wakeup_trace(struct trace_array *tr,
44.                       struct trace_array_cpu *data,
45.                       struct task_struct *wakee,
46.                       struct task_struct *curr,
47.                       unsigned long flags, int pc)
48. {
49.       struct ring_buffer_event *event;
50.       struct ctx_switch_entry *entry;
51.       unsigned long irq_flags;
52.
53.       event = ring_buffer_lock_reserve(tr->buffer,
    sizeof(*entry),
54.                                        &irq_flags);
55.       if (!event)
56.             return;
57.       entry = ring_buffer_event_data(event);
58.       tracing_generic_entry_update(&entry->ent, flags,
    pc);
```

```
59.       entry->ent.type                 = TRACE_WAKE;
60.       entry->prev_pid                 = curr->pid;
61.       entry->prev_prio          = curr->prio;
62.       entry->prev_state         = curr->state;
63.       entry->next_pid                 = wakee->pid;
64.       entry->next_prio          = wakee->prio;
65.       entry->next_state         = wakee->state;
66.       entry->next_cpu                 = task_cpu(wakee);
67.
68.       //Modifica al kernel per RTAI
69.       if (wakee->pid==pid_prev_RTAI || wakee->pid>10000)
70.             if (pid_prev_RTAI!=-1 && pid_next_RTAI!=-1) {
71.                   entry->prev_pid=entry-
   >ent.pid=pid_next_RTAI;
72.                   entry->prev_prio=prio_next_RTAI;
73.                   entry->prev_state=0;
74.             }
75.
76.       ring_buffer_unlock_commit(tr->buffer, event,
   irq_flags);
77.       ftrace_trace_stack(tr, data, flags, 6, pc);
78.       ftrace_trace_userstack(tr, data, flags, pc);
79.
80.       trace_wake_up();
81. }
```

Definizione variabili nella riga 2: `pid_prev_RTAI` memorizza il pid di `prev`, `pid_next_RTAI` quello di `next` e `prio_next_RTAI` la priorità di `next` (sono inizializzati a -1).
Dalla riga 31 alla 35: se `prev` è un task RTAI (se è diverso da `current`) si salva il suo pid in `entry->ent.pid` (sovrascrivendo quello di `current`) e in `pid_prev_RTAI`, poi si memorizzano pid e priorità di `next`.
Dalla riga 69 alla 74: si controlla se il pid di `wakee` è uguale a `pid_prev_RTAI` (se è il processo `prev` che ha ceduto il posto a `next`) o se ha un pid superiore a 10000 (se è un task RTAI nativo). Se è così e i valori di `pid_prev_RTAI` e `pid_next_RTAI` sono entrambi diversi da -1, si sostituiscono i dati di `current`: `entry->prev_pid` e `entry->ent.pid` ricevono `pid_next_RTAI`, il pid del task RTAI corrente (convertito con `task_conv`), `entry->prev_prio`, la sua priorità e `entry->prev_state` viene posto pari a zero, visto che è in esecuzione. In questo modo viene preso il nome di `next` e la prima terna pid-prio-state, che appare sotto `FUNCTION` nel file `trace`, sarà corretta.
Si vuole ribadire che queste modifiche non vanno ad intaccare il normale funzionamento di ftrace quando sta tracciando solo processi Linux: infatti vengono chiamate in causa se ci sono, anche, task RTAI. Inoltre il loro tempo di esecuzione è costante (O(1)).

7.3 File modificati

Prima di presentare i risultati vengono riepilogati tutti i file cambiati per lo sviluppo di questa integrazione. Sono tre: due di RTAI e uno del kernel di Linux. Nella tabella sottostante con `PATH` si intende il percorso dove viene posizionata la directory dei sorgenti di RTAI (ad esempio: `/home/nome_utente/kernel/RTAI/rtai-x.x.x`) e `TREE` ha lo stesso significato usato nei paragrafi 4.7.1 e 4.7.2.

Nome del file	**Percorso del file**
`rtai_schedcore.h`	`PATH/base/include`
`sched.c`	`PATH/base/sched`
`trace.c`	`TREE/kernel/trace`

7.4 Risultati ottenuti

Costruita la funzione `task_conv` in RTAI, modificate `switch_rtai_tasks`, `wake_up_timed_tasks`, `tracing_sched_switch_trace` e `tracing_sched_wakeup_trace` come indicato nei paragrafi precedenti, sono stati eseguiti dei test per mettere in gioco questa integrazione. Siccome sono stati cambiati due file di RTAI, il sistema real time va ricompilato, reinstallato e Linux riavviato, per rendere le modifiche presenti ed effettive.
A sistema pronto è stato abilitato RTAI e caricato un modulo che creava un processo. È stato attivato ftrace, impostato `sched_switch` come tracer corrente, lasciato tracciare per qualche istante di tempo, fermato e, poi, visualizzato il file `trace`. Il task inserito veniva visto da ftrace.
Il processo era periodico e il suo compito era quello di scrivere, ad ogni iterazione di un ciclo nel log del kernel, un messaggio, finché non si verificava una condizione di uscita e terminava la sua esecuzione. Viene riportato il codice del modulo che crea questo task:

```
#define TASK_STACK 2048
#define TASK_PRIO 1
RT_TASK task_desc;

void task_body (long cookie) {
int i=1;
      while (1) {
            printk ("%i\n", i);
            if (i>=2000)
                  break;
            ++i;
            rt_task_wait_period ();
      }
}
```

```
static int /*init*/ __init task_init (void) {
      int err;
      err=rt_task_init (&task_desc, task_body, 1, TASK_STACK,
TASK_PRIO, 0, 0);
      if (err!=0) {
            printk (KERN_INFO "Error: can't init RTAI task\n");
            return -1;
      }
      else
            printk (KERN_INFO "RTAI task initialized\n");
      rt_set_oneshot_mode ();
      start_rt_timer (0);
      rt_task_make_periodic_relative_ns     (&task_desc,     0,
100000000);
      return 0;
}

void /*cleanup*/ __exit task_exit (void) {
      stop_rt_timer ();
      rt_task_delete (&task_desc);
      printk (KERN_INFO "RTAI task deleted\n");
}

module_init (task_init);
module_exit (task_exit);
```

Lo stack del processo è pari a 2048 byte, la sua priorità è uno e il descrittore è la variabile `task_desc`. La funzione `task_body` identifica l'attività descritta prima: in pratica, il task scrive il numero dell'iterazione del suo ciclo interno sui messaggi di log del kernel e quando arriva a 2000 stampe termina l'esecuzione. `task_init` con la macro `__init` è la routine chiamata quando il modulo è inserito con il comando `insmod` e `task_exit` (macro `__exit`) è quella invocata quando il modulo è tolto (comando `rmmod`).
Viene presentato il tracciamento, testuale e grafico, ottenuto con questo task, scheduler `rtai_sched.ko` e disattivazione del tracing dei task Linux.

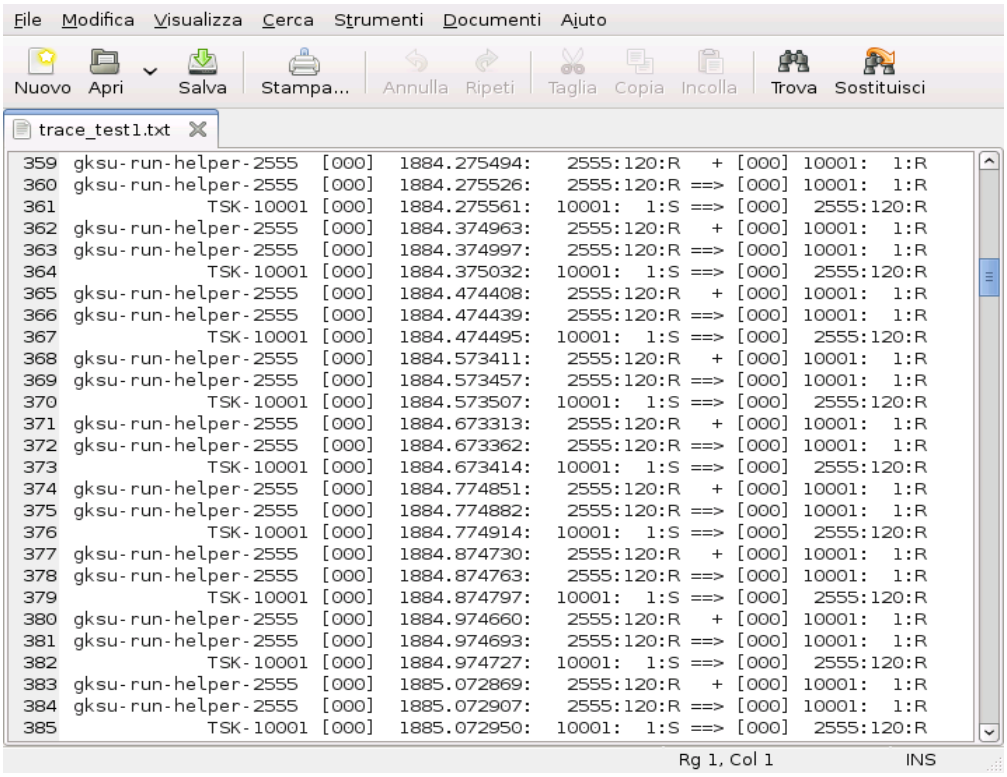

Illustrazione 14: Risultato testuale della prova con un solo processo RTAI (sono tracciati solo task RTAI)

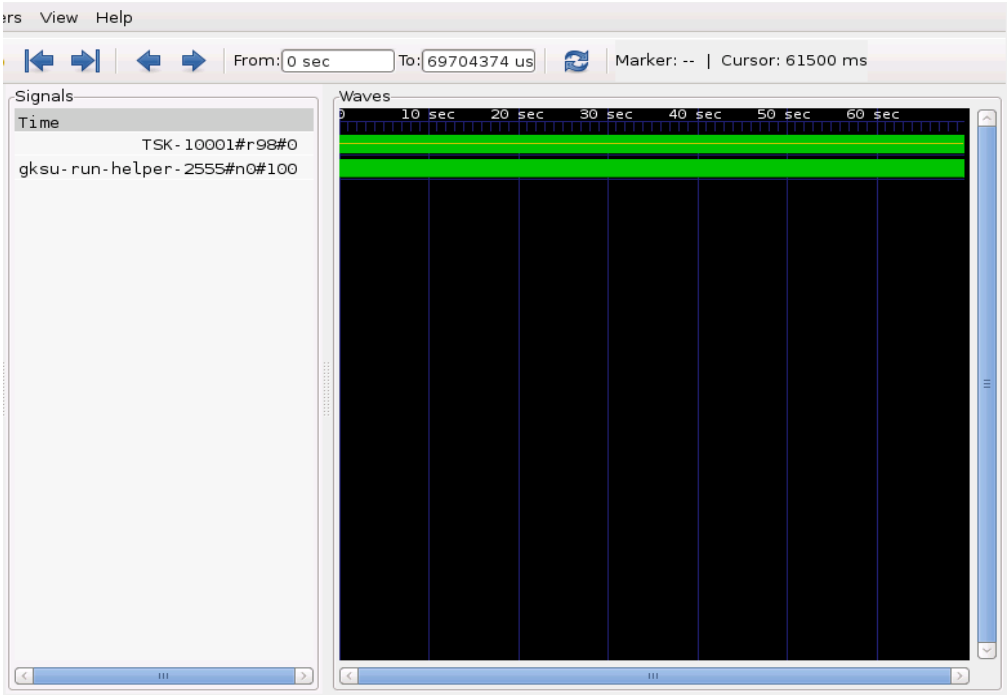

Illustrazione 15: Risultato grafico della prova con un solo processo RTAI (sono tracciati solo task RTAI)

Il tracing dei processi Linux è stato disabilitato per verificare, come prima cosa, il comportamento dei soli task RTAI, come si relazionavano tra loro e, poi, per una esigenza di programmazione, con meno processi si trovano più facilmente gli errori.
Nel file di testo sono evidenti le commutazioni e le sveglie che avvengono tra i task tracciati: `gksu-run-helper` è un processo di tipo Linux, invece `TSK` è il nostro processo RTAI nativo. Nella visione grafica, invece, gli switch sono in corrispondenza di linee verticali sui segnali (task): l'immagine non è ingrandita e le linee sono molto fitte in questo caso.
Lo stesso test andava rifatto, anche, con i task Linux, dopo aver riattivato il loro tracciamento. I risultati sono i seguenti:

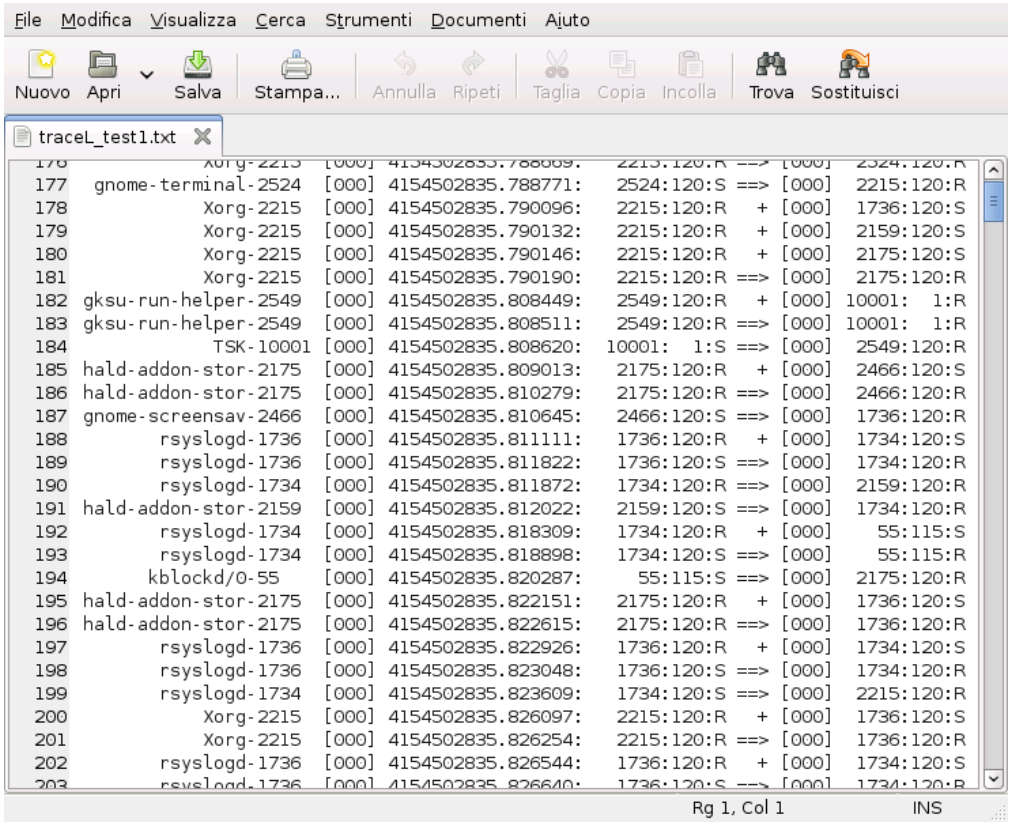

Illustrazione 16: Risultato testuale della prova con un solo processo RTAI (sono tracciati task Linux ed RTAI)

I processi Linux tracciati sono molti, insieme a quelli RTAI: `gksu-run-helper` con pid 2549 e `TSK` con pid 10001. Nell'immagine sotto, le linee verticali rappresentano, anche, qua le commutazioni di contesto che avvengono tra i task.

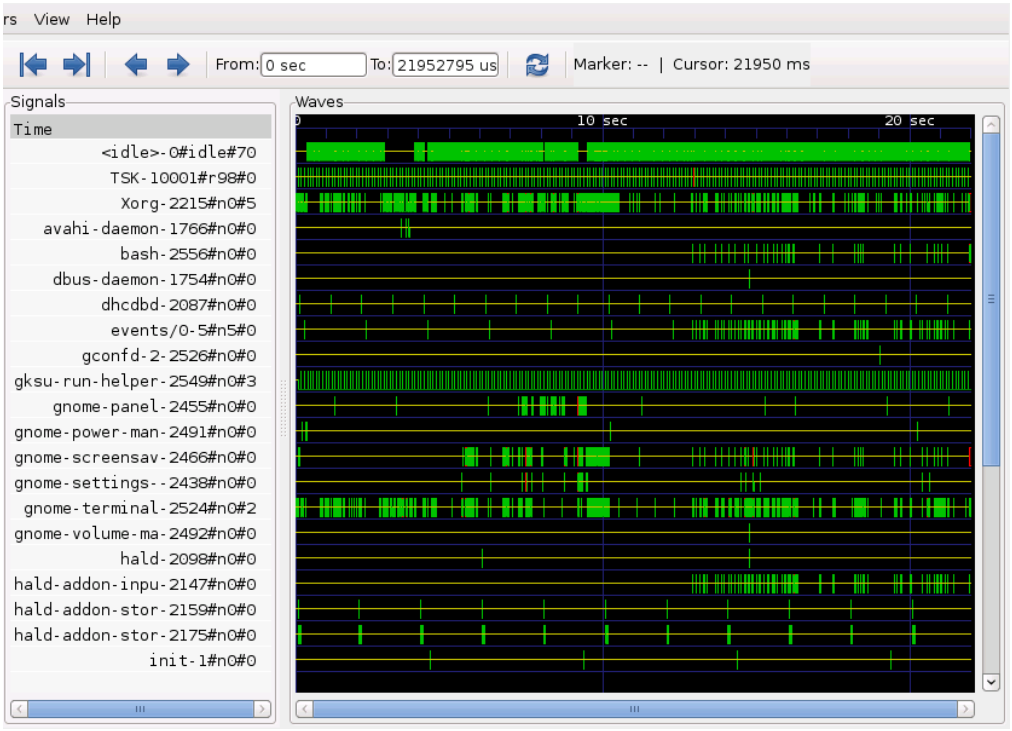

Illustrazione 17: Risultato grafico della prova con un solo processo RTAI (sono tracciati task Linux ed RTAI)

Questa prova è stata realizzata, successivamente, con due processi kernel space. Entrambi venivano tracciati. Il primo task è lo stesso del test precedente, invece, il secondo è simile solo che ha priorità zero e realizza 1000 stampe sul log del kernel e poi finisce di eseguirsi.

```
#define TASK_STACK 2048
#define TASK_PRIO 0
RT_TASK task_desc;

void task_body (long cookie) {
int i=1;
      while (1) {
            printk ("%i\n", i);
            if (i>=1000)
                  break;
            ++i;
            rt_task_wait_period ();
      }
}

static int /*init*/ __init task_init (void) {
      int err;
      printk ("&task_desc: %p\n", &task_desc);
      err=rt_task_init (&task_desc, task_body, 1, TASK_STACK,
TASK_PRIO, 0, 0);
      if (err!=0) {
            printk (KERN_INFO "Error: can't init RTAI task\n");
            return -1;
      }
```

```
        else
                printk (KERN_INFO "RTAI task initialized\n");
        rt_set_oneshot_mode ();
        start_rt_timer (0);
        rt_task_make_periodic_relative_ns (&task_desc, 0,
10000000);
        return 0;
}

void /*cleanup*/ __exit task_exit (void) {
        stop_rt_timer ();
        rt_task_delete (&task_desc);
        printk (KERN_INFO "RTAI task deleted\n");
}

module_init (task_init);
module_exit (task_exit);
```

Lo stack è, sempre, di 2048 byte, la priorità, invece, è zero (questo processo è più prioritario del primo). Stampa il numero delle iterazioni e quando arriva a 1000 termina.
Come nell'esempio precedente, è stato tolto il tracing dei task Linux e verificati gli switch con soli processi RTAI (scheduler `rtai_sched.ko`). Nei risultati si vedono i task RTAI coinvolti nel test, i due `TSK` (pid 10001 e 10002) sono RTAI nativi, l'altro è di tipo Linux:

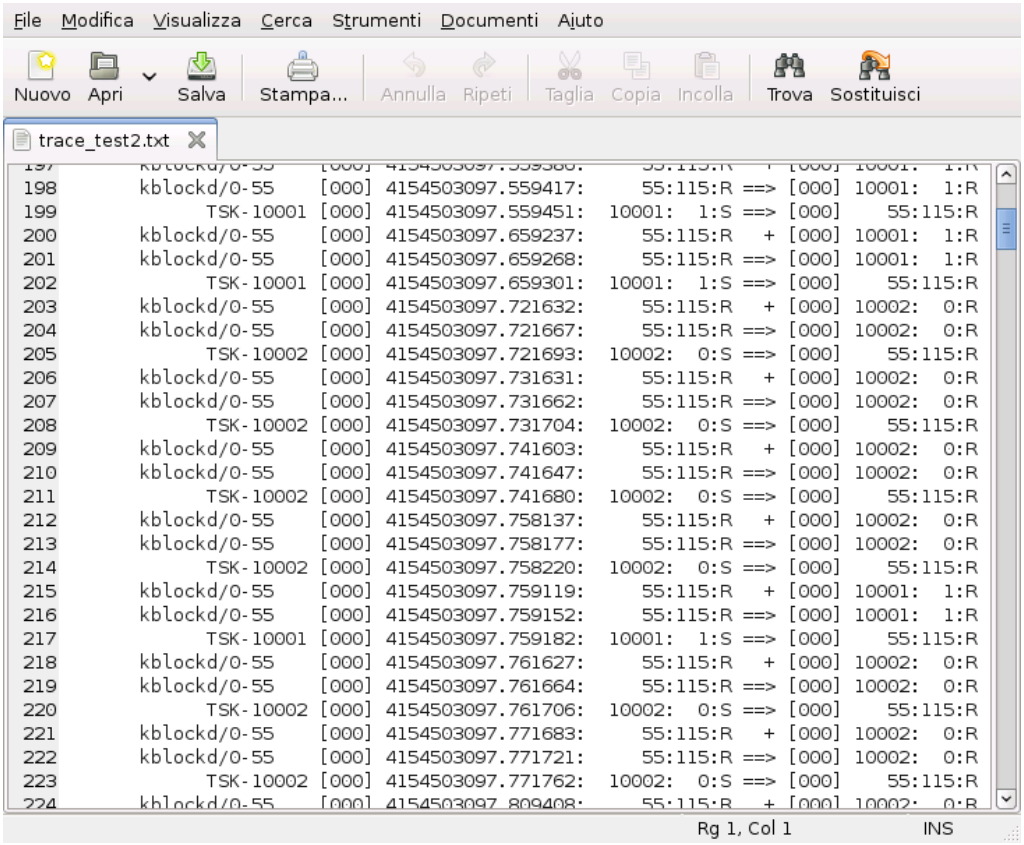

Illustrazione 18: Risultato testuale della prova con due processi RTAI (sono tracciati solo task RTAI)

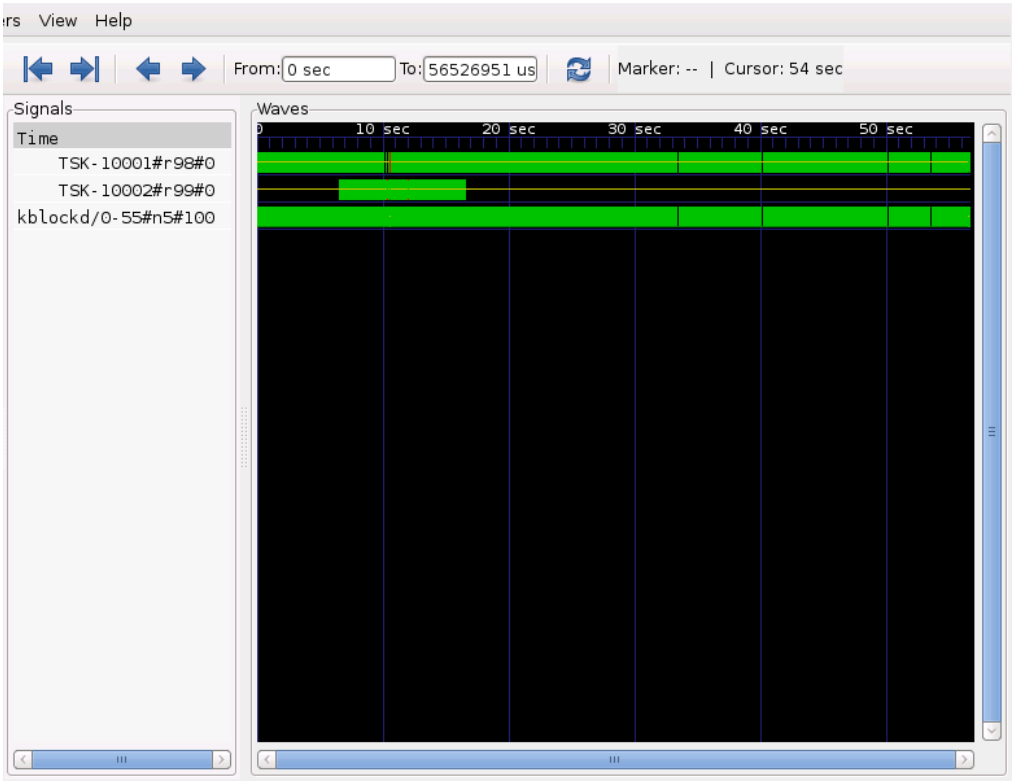

Illustrazione 19: Risultato grafico della prova con due processi RTAI (sono tracciati solo task RTAI)

Nell'immagine successiva sono stati inseriti solo i due processi `TSK` evidenziando, in un frammento temporale, quando sono sospesi (giallo), quando sono ready, ma non hanno la CPU (blocco rosso) e quando sono in esecuzione (verde).

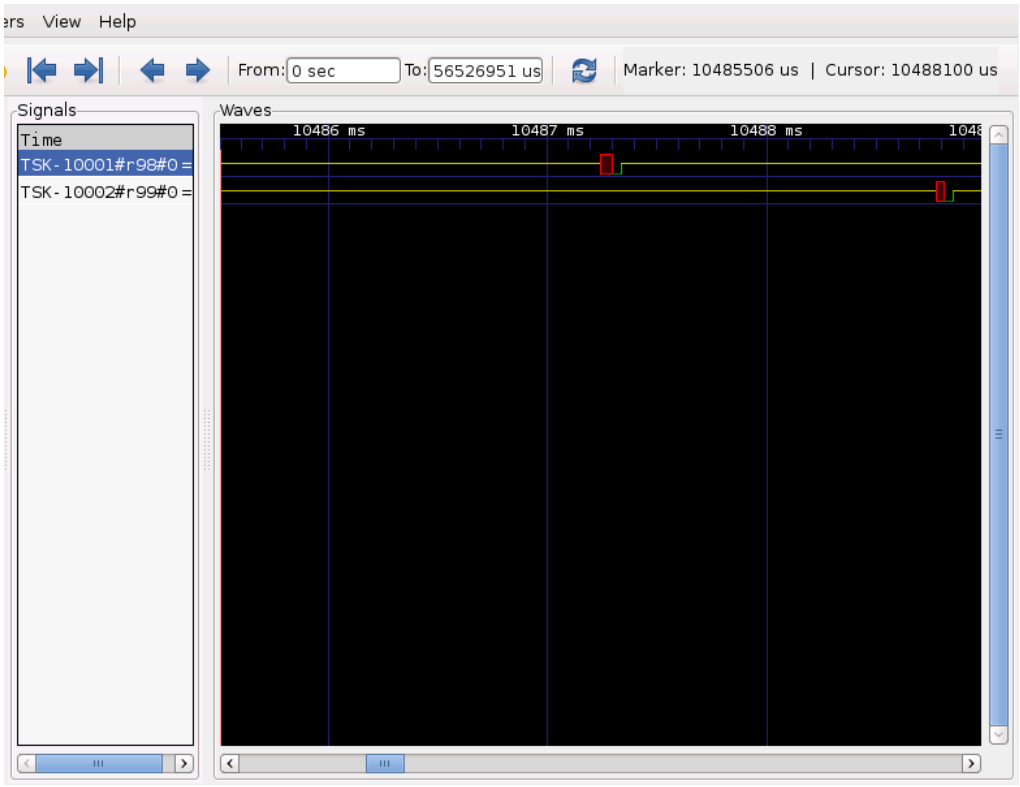

Illustrazione 20: Risultato grafico della prova con due processi RTAI (sono tracciati solo task RTAI): visualizzati solo i task RTAI nativi

Prova ripetuta insieme ai processi Linux. Viene riportata, solo, la visualizzazione grafica:

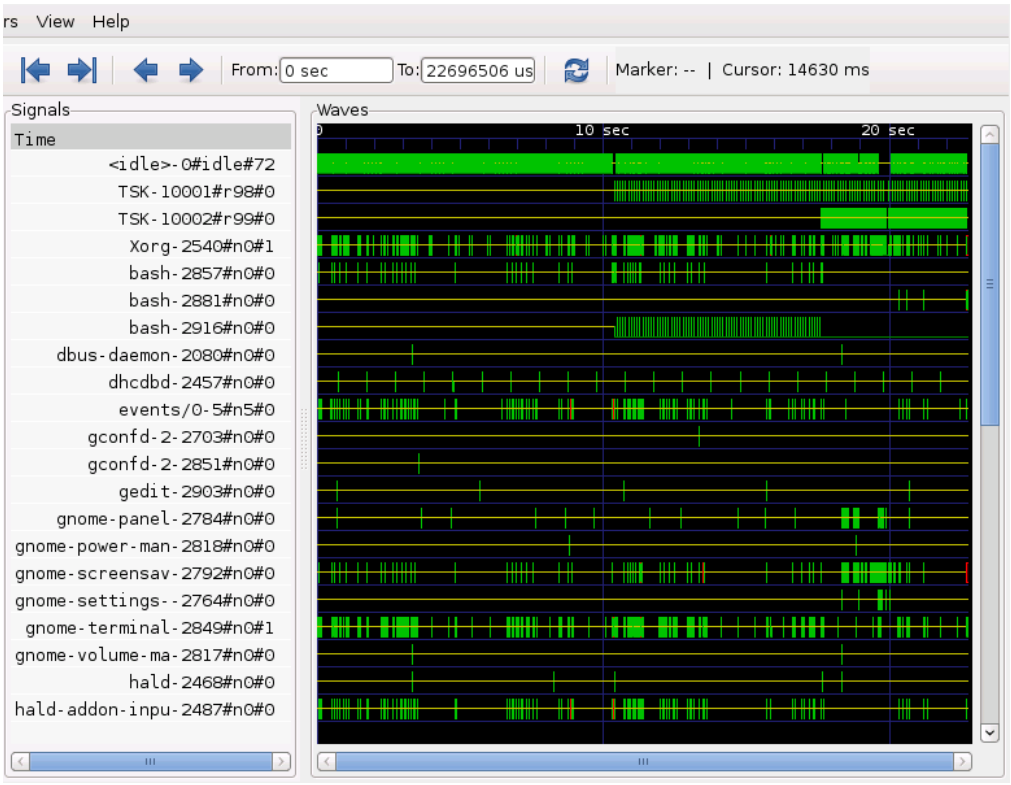

Illustrazione 21: Risultato grafico della prova con due processi RTAI (sono tracciati task Linux ed RTAI)

Appaiono, anche in questo caso, molti task Linux. Si notano, comunque, i soliti TSK (pid 10001 e 10002).

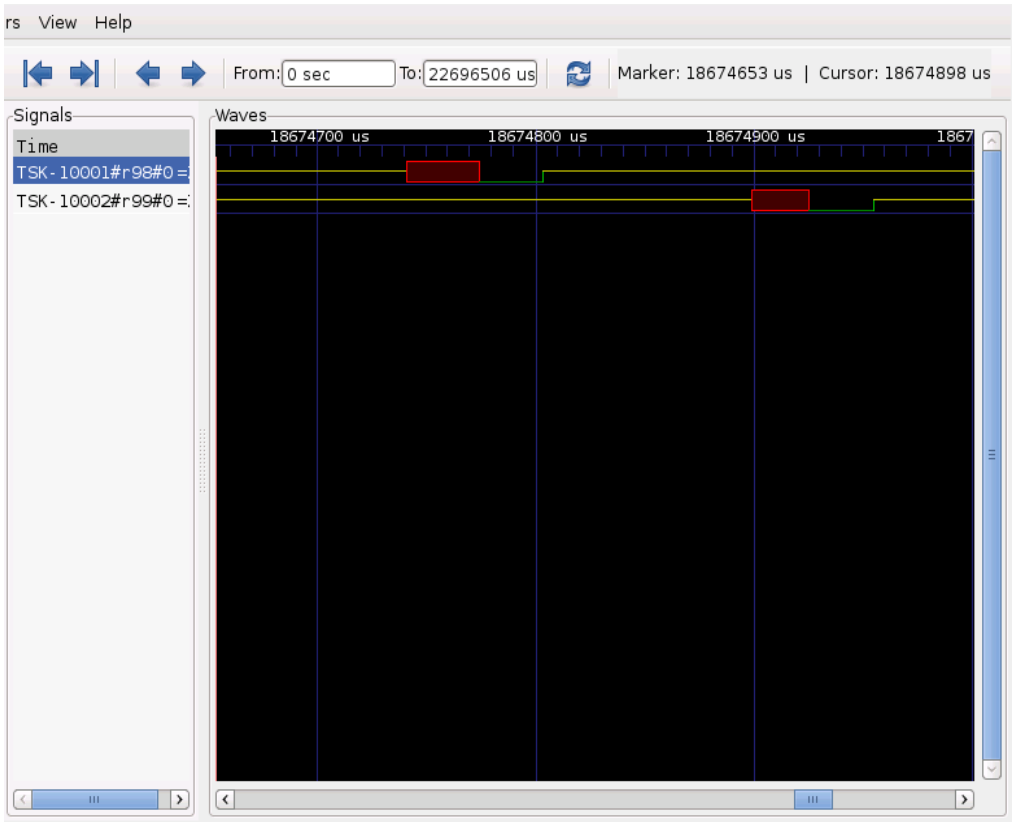

Illustrazione 22: Risultato grafico della prova con due processi RTAI (sono tracciati task Linux ed RTAI): visualizzati solo task RTAI nativi

L'immagine precedente mostra, in un piccolo pezzo temporale, il dettaglio dei due `TSK`.
Con questi test si è verificato il funzionamento di `task_conv` e dell'integrazione complessiva.

8 Conclusioni

In questa tesi si è affrontato il tracing dei processi, thread (in generale task) gestiti da un sistema operativo. In un sistema multitasking più programmi si contendono l'uso della/e CPU, per questo motivo ogni unità di esecuzione delle diverse applicazioni (thread) si può trovare in un diverso stato: pronto, sospeso, terminato.
Il tracciamento dei processi nei sistemi operativi moderni, la cui complessità è divenuta oggigiorno piuttosto consistente, assume maggior importanza a causa del tempo di esecuzione non più deterministico delle singole istruzioni a livello del processore. Conoscere il tempo di esecuzione di un determinato segmento di codice in applicazioni di tipo real time è di fondamentale importanza per poter determinare se vengono rispettati i requisiti temporali.
Gli strumenti di tracing vengono, quindi, utilizzati non solo per determinare in quale stato un task si trova in un dato istante della sua vita nel sistema operativo, ma anche per determinare quanto tempo richiede la sua esecuzione.
Il mondo del software open source offre svariati strumenti di tracing: solo per Linux abbiamo visto, in uno dei primi capitoli della tesi, che esistono ben tre pacchetti per il tracciamento dei processi. Il tool che abbiamo approfonditamente analizzato ed esteso è ftrace, recentemente integrato nel kernel di Linux (dalla versione 2.6.27).
Il software ftrace è stato analizzato a diversi livelli per capirne dettagliatamente il funzionamento: l'attivazione/disattivazione, cosa può tracciare (può infatti tracciare non solo le commutazioni e le riattivazioni dei task, ma anche interrupt, regioni critiche ecc..), come si interfaccia con l'utente ed infine quale è il suo macro funzionamento (riferendosi, soprattutto, al tracer `sched_switch`).
Tutto questo lavoro ha permesso di tracciare anche i task nativi di RTAI che vengono eseguiti in kernel space e hanno una numerazione di pid completamente diversa da quella dei task di Linux. In un sistema Linux/RTAI si ha a che fare con due kernel che eseguono contemporaneamente sulla stessa CPU, per questo motivo ha senso avere due numerazioni di pid diversi, una che viene gestita da uno scheduler ed una che viene gestita dall'altro.
L'attività di tesi ha reso possibile il tracciamento di tutti i task, real time e non, in un sistema Linux/RTAI, potendo così permettere allo sviluppatore di un sistema in tempo reale di verificare se i task riescono o meno a rispettare le deadline. Nella tesi si è visto quali sono stati i passi per arrivare al tracciamento dei cambi di contesto e della riattivazione dei task (passaggio di questi dallo stato di wait alla coda ready). Il cuore del lavoro di integrazione è una funzione (`task_conv`) che permette di convertire una struttura descrittore di task RTAI in un descrittore di processo Linux.
Questo lavoro di integrazione di ftrace in RTAI è di particolare importanza per la diagnosi e il monitoring di un sistema in tempo reale. Esso ha quindi un valore non limitato al livello accademico; per questo motivo si è iniziato un lavoro di merging con la comunità di RTAI, al fine di includerlo nella distribuzione mainline di RTAI.

Bibliografia

[1] http://lwn.net
[2] http://www.kernel.org
[3] http://www.rtai.org
[4] http://it.wikipedia.org
[5] http://lttng.org
[6] Antonio Barbalace, *Multitasking real-time in Linux2.4.19-RTAI3.1 su architettura Xscale PXA255*, Università di Padova, Facoltà di Ingegneria Informatica, anno accademico 2004/2005
[7] Antonio Barbalace, *Porting e valutazione delle prestazioni di un ambiente per applicazioni di controllo hard Real-Time da VxWorks a Linux-RTAI/Xenomai-RTnet su architettura PowerPc*, Università di Padova, Facoltà di Ingegneria Informatica, anno accademico 2006/2007
[8] Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel Second Edition*, O'Reilly, Dicembre 2002
[9] Robert Love, *Linux Kernel Development Second Edition*, Sams, Gennaio 2005
[10] Steven Rostedt, *Ftrace Tutorial*, slide di presentazione

Ringraziamenti

Questa tesi è stata svolta grazie all'aiuto di diverse persone. Un grazie ai miei genitori che mi hanno consentito di frequentare l'università permettendomi di arrivare fino a questo punto e grazie a mia sorella per i suoi continui stimoli. Grazie al mio relatore Professore Sergio Congiu e al mio correlatore Ingegnere Antonio Barbalace per avermi seguito durante questo lavoro. Ringrazio gli amici universitari che mi hanno aiutato e sostenuto: Samuele, Rudy, Davide, Andrea, Andrea, Nicola, Riccardo. Infine, un grazie va a tutte le altre persone e amici qui non citati. Grazie a tutti.