

Linux e sistemi embedded

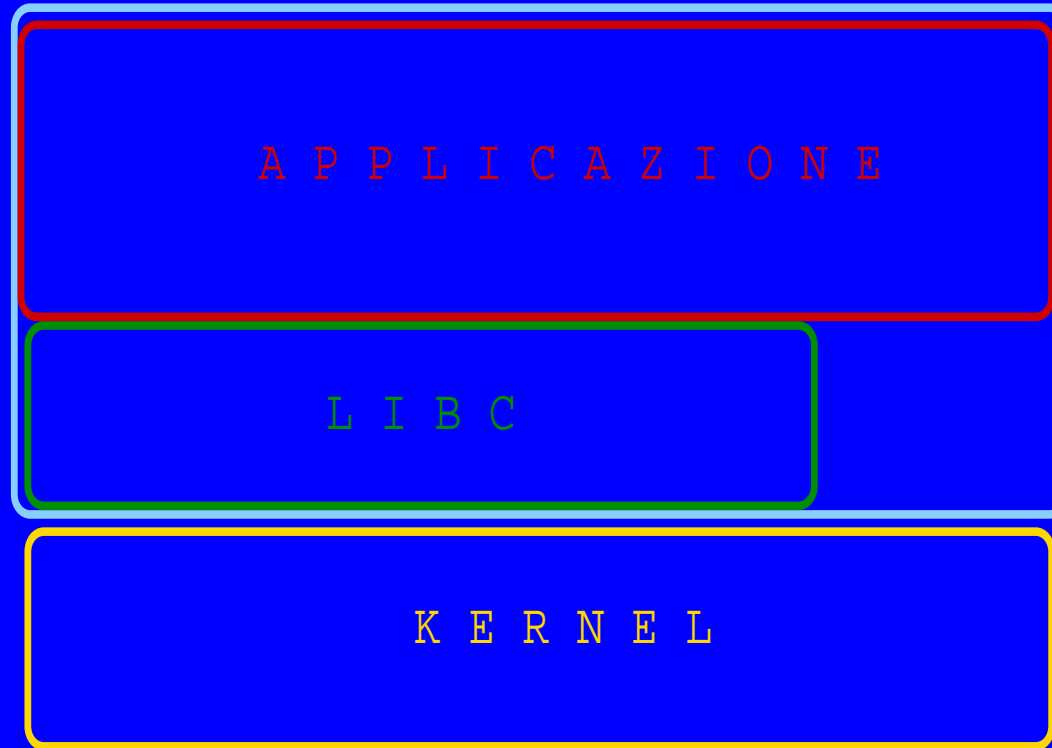
Davide Ciminaghi
ciminaghi@gnuudd.com

Bologna, 16 Novembre 2004

Argomenti

- Premessa: GNU Linux e non solo Linux
- Kernel Linux e sistemi operativi embedded tradizionali
- Il livello applicativo e la modularità
- I problemi della schedulazione real time sotto Linux
- Politiche di schedulazione e gestione delle priorità in Linux
- Kernel non-preemptible e schedulazione
- Linux e real time
- Preemption e low-latency patch
- RTAI: concetti base

GNU Linux e non solo Linux



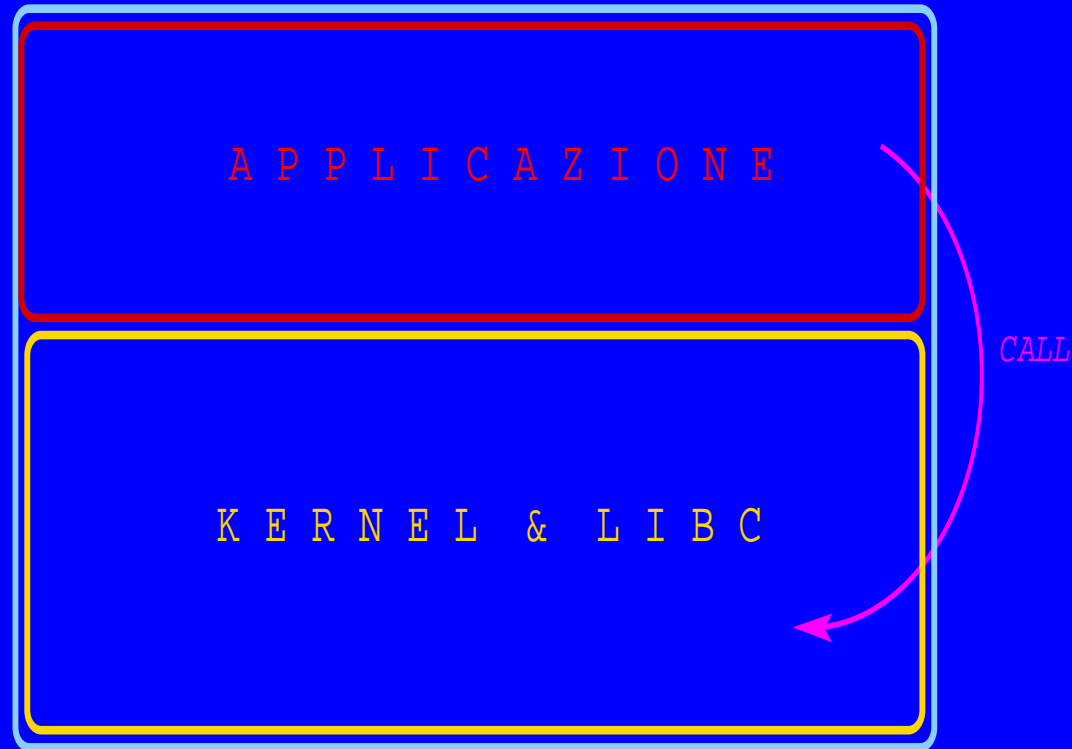
- Linux corrisponde al kernel
- GNU/Linux viene usato per indicare l'intero sistema

GNU Linux e non solo Linux

NOTA IMPORTANTE

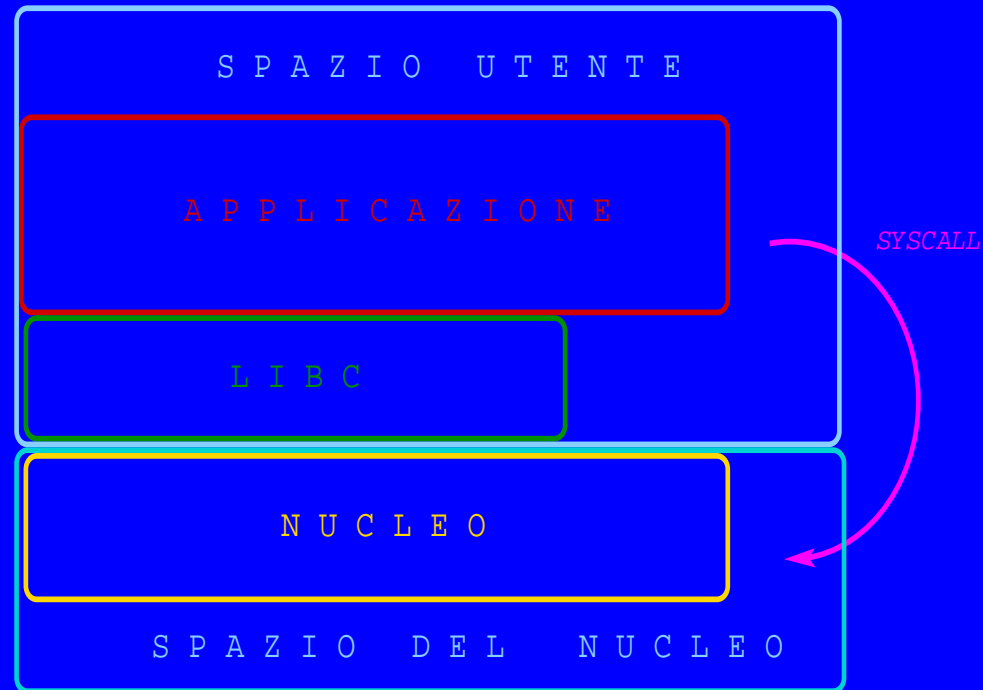
In un sistema embedded servono sia le applicazioni sia i tool di sviluppo, ossia programmi che girano sulla piattaforma host e consentono la cross compilazione ed il debugging del target

Struttura tradizionale



- Le applicazioni sono collegate al nucleo ed alla libreria C.
- Le chiamate al nucleo sono delle call.

Struttura Linux



- Le applicazioni sono collegate con la sola libc.
- Le chiamate al nucleo (kernel) sono "chiamate di sistema"
- Divisione "spazio del nucleo" / "spazio utente".

Linux e sistemi tradizionali

Alcuni punti fondamentali (valgono non solo per Linux, ma per tutti gli UNIX):

- Linux richiede sempre la presenza di un file system per funzionare.
- Sotto Linux l'accesso ai driver di periferica avviene tramite il file system.
- Il nucleo Linux funziona in modo supervisore ed è abilitato a svolgere compiti vietati al codice delle applicazioni.

Linux e sistemi tradizionali (segue)

- Sulle macchine con MMU, ogni processo gira nel proprio spazio di indirizzamento. Il kernel a sua volta risiede in uno spazio di indirizzamento inaccessibile ai processi utente.
- Sulle macchine con MMU, la chiamata `mmap()` consente di mappare in memoria utente parti di file (ma anche le periferiche sono file !)
- Linux crea processi con `fork/exec`.
- Linux dispone di un grande numero di modalità di comunicazione interprocesso, frutto della lunga storia UNIX.

Vantaggi e svantaggi.

- La separazione di compiti tra spazio utente e spazio del kernel nonché la protezione di quest'ultimo e dell'hardware aumenta l'affidabilità del sistema: un malfunzionamento di un processo utente non compromette l'intera macchina.
- La MMU evita la frammentazione della memoria.
- La MMU permette l'implementazione della chiamata `fork()` evitando la copia di tutte le pagine di memoria del processo chiamante (Copy On Write).
- Le prestazioni però diminuiscono (l'esecuzione di una chiamata di sistema è un processo relativamente lungo).

Il livello applicativo

- Filosofia modulare: programmi piccoli che fanno poche cose.
- Redirezione dell'output: uso delle pipe.
- Linguaggi di scripting (bash, tcl, perl, ...)

Conclusione:

- UN' APPLICAZIONE PUO' ESSERE SPEZZATA IN MOLTI MODULI PICCOLI SCRITTI IN UNA VARIETA' DI LINGUAGGI DIFFERENTI.

Un esempio di utilizzo della "filosofia modulare"

Comando per lanciare un programma di test su un target remoto ed elaborare i risultati in locale:

```
#!/bin/sh  
.....
```

```
ssh $target /bin/do_test 2>$data_file \  
1>/dev/null & \  
sleep $time ; \  
ssh $target "kill -9 $(getpid)"
```

```
.....
```


Linguaggi di scripting.

- Esistono applicazioni (o moduli) realizzabili in modo semplice per mezzo dei linguaggi di scripting (es.: elaborazione di regexp).
- L'uso di un linguaggio interpretato può migliorare le prestazioni in termini di velocità ed occupazione di memoria.

Etlinux.

- Etlinux è una distribuzione non commerciale basata su un linguaggio interpretato (ettcl) che è a sua volta un'estensione del classico tcl.
- La particolarità di Etlinux è che molte delle sue applicazioni sono realizzate in tcl anzichè essere programmi compilati.

Etlinux.

Nel caso di etlcl, si ha un risparmio dei tempi di esecuzione quando si esegue un nuovo programma.

Usando un linguaggio compilato, la procedura per eseguire un nuovo programma prevede due passi:

- `fork()` : il processo chiamante produce una copia identica di se stesso.
- `exec()` : la nuova copia elimina tutte le aree di memoria del vecchio programma e carica quelle del nuovo.

Etlinux.

Usando ettbl, il primo passo è lo stesso, ma il secondo può essere eliminato grazie all'uso del comando source:

```
#####  
# open a real pipe  
sys_pipe pin pout  
if [sys_fork] {  
    # father  
    close $pin; set w $pout  
    return; # child is reaped later  
}  
# child; don't close sock  
close $pout; sys_dup $pin stdin; close $pin  
sys_dup $sock stdout  
global argv argv0  
set argv0 $exe; set argv ""  
uplevel #0 [list source $exe]  
exit 0  
}
```

Etlinux.

- Il meccanismo di Copy On Write fa in modo che la chiamata `fork()` sia indolore perchè molte delle pagine di memoria del padre potranno essere condivise con il figlio (entrambi sono interpreti etc).
- Con `source` si riesce ad evitare la chiamata `exec()` ed il relativo "spreco" di tempo e risorse.

Nota sulle macchine senza MMU.

- In assenza della MMU il meccanismo di Copy On Write non può essere implementato.
- Per questo motivo la chiamata `fork()` non è implementata, mentre lo è la `vfork()` (che blocca fino a quando il chiamante non muore oppure non esegue un nuovo programma con `exec()`).

Un esempio pratico.

Per l'occasione Etlinux è stato compilato per una piattaforma ARM (at91).

Il file system di root comprende:

- Programmi di base (ettclsh, init,).
- Server http (in ettcl).
- Server ssh.
- Altro (crond, smbd, ...).

Un esempio pratico.

Real time.

Definizioni di termini (tanto per sintonizzarci):

- "Hard real time" : se il software non completa i suoi compiti entro il tempo stabilito dalla specifica, il sistema controllato collassa in modo catastrofico.
- "Soft real time": al software non è richiesto di rispondere in tempi certi pena la morte del sistema sotto controllo. Deve però rispondere in tempo nella maggior parte dei casi. Es.: riproduzione audio o video.

Linux e real time.

Cercheremo di chiarire cosa Linux può e non può fare ed in cosa consistono i meccanismi che consentono le estensioni hard real time del kernel.

In una prima fase parleremo dei kernel fino al 2.4, successivamente vedremo cosa è cambiato nel kernel Linux dal punto di vista delle prestazioni real time.

Infine parleremo dei concetti che stanno alla base di rtai.

Kernel non-preemptible

Nelle slide che seguono vediamo come funziona la schedulazione sotto Linux.

Quello che è importante tenere presente è che (fino al 2.4):

- Linux è un sistema preemptible per quanto riguarda i processi: ad un processo può essere levato il controllo anche se il processo stesso non lo concede esplicitamente.
- Il kernel di Linux non è preemptible: ossia lo scheduler può funzionare solo in certi momenti predefiniti oppure quando il kernel volontariamente ne invoca l'esecuzione.

Politiche di schedulazione.

Per quanto riguarda gli schedulatori UNIX, i processi vengono suddivisi in due grandi categorie logiche:

- Processi "real-time"
- Processi "tradizionali"

Politiche di schedulazione (segue)

Ad ogni processo vengono associate una politica di schedulazione, una priorità statica ed una priorità dinamica.

Le possibili politiche di schedulazione sono tre:

- SCHED_FIFO e SCHED_RR per i processi real time.
- SCHED_OTHER per i processi tradizionali.

Priorità statica.

La priorità statica è rappresentata da un numero intero compreso tra 0 e 99.

Processi con priorità statica più alta sono considerati più prioritari.

Una priorità statica maggiore di zero può essere assegnata solo ai processi real time (ossia ai processi con politica SCHED_FIFO o SCHED_RR)

Politica SCHED_FIFO.

Un processo continua a mantenere il controllo fino a quando non diventa eseguibile un altro processo con priorità statica più elevata.

Una volta che il processo perde il controllo a favore di uno più prioritario, viene inserito in testa alla lista dei processi eseguibili con priorità pari alla sua.

Quando invece un processo FIFO diventa eseguibile, viene inserito in fondo alla lista di processi eseguibili con priorità pari alla sua.

Un unico processo può controllare la macchina per il 100% del tempo.

Time quantum.

Ad ogni processo tradizionale e Round Robin è assegnato un periodo di tempo massimo durante il quale il processo stesso può mantenere il controllo della CPU.

Scaduto tale periodo, lo scheduler sceglie un nuovo processo.

Quando tutti i processi hanno esaurito la propria "fetta" di tempo macchina (in inglese Time Quantum), tutti i contatori vengono reinizializzati.

Politica SCHED_RR.

Ai processi schedulati con politica SCHED_RR viene associato un time quantum.

SCHED_RR funziona come SCHED_FIFO, ma, quando un processo perde il controllo dopo aver esaurito il proprio Time Quantum, viene inserito dallo schedulatore in coda alla lista dei processi eseguibili con la sua stessa priorità.

Se invece un processo SCHED_RR perde il controllo a causa di un altro processo con priorità statica più alta, viene inserito in testa alla lista dei processi eseguibili aventi la sua stessa priorità (deve esaurire la propria fetta di tempo prima di tornare in fondo alla lista).

Priorità dinamica.

Questa priorità si applica ai soli processi convenzionali.

E' costituita essenzialmente da un valore di base sommato al numero di tick mancanti all'esaurimento del time quantum per il processo.

In sostanza si tende a rendere più prioritari i processi che girano per meno tempo.

Livello di nice.

E' l'opposto del valore di base di cui si diceva.

Trucco per ricordarsi cos'è il livello di nice:

- nice = carino => valori di nice più bassi implicano che un processo sia meno "carino" nei confronti degli altri => più prioritario.

I possibili valori di nice vanno da -20 a +19 .

Politica SCHED_OTHER.

Questa politica è quella standard dei sistemi UNIX. Utilizza la priorità dinamica ed il time quantum per decidere quale processo far girare.

La priorità dinamica di un processo viene fatta decrescere nel corso del time quantum associato al processo stesso. In questo modo si favorisce una equa distribuzione del tempo macchina.

Capire meglio.

A questo punto, se avete le idee confuse, vi capisco. Per capire meglio l'interazione tra politica di schedulazione, priorità statica e dinamica e time quantum torna utile guardare il sorgente.

In particolare nella decisione sul processo da scegliere è cruciale il ruolo della funzione `goodness()` in `kernel/sched.c`:

Capire meglio (segue)

```
static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{
    int weight = -1;
    if (p->policy == SCHED_OTHER) {
        /*
         * Give the process a first-approximation goodness value
         * according to the number of clock-ticks it has left.
         *
         * Don't do any other calculations if the time slice is
         * over..
         */
        weight = p->counter;
        if (!weight)
            goto out;
        /* .. and a slight advantage to the current MM */
        if (p->mm == this_mm || !p->mm)
            weight += 1;
        weight += 20 - p->nice;
        goto out;
    }
    /*
     * Realtime process, select the first one on the
     * runqueue (taking priorities within processes
     * into account).
     */
    weight = 1000 + p->rt_priority;
out:
    return weight;
}
```

Capire meglio (segue)

Si noti che il time quantum assegnato ad un processo dipende dal livello di nice del processo stesso:

```
/*
 * Scheduling quanta.
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x) ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x) ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x) (x)
#elif HZ < 1600
#define TICK_SCALE(x) ((x) << 1)
#else
#define TICK_SCALE(x) ((x) << 2)
#endif

#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)
```

Nota importante

La funzione `goodness()` viene richiamata tante volte quanti sono i task eseguibili ogni volta che tutti i task hanno esaurito la loro fetta di tempo.

L'algoritmo del calcolo delle priorità richiede quindi un tempo dipendente dal numero di task eseguibili in un certo istante, che non è determinabile a priori.

Kernel non-preemptible e schedulazione.

FINO AL KERNEL 2.4:

Lo schedulatore Linux gira solo nelle seguenti situazioni:

- Al ritorno da interrupt o da eccezione.
- Al ritorno dalle chiamate di sistema.
- In seguito a chiamate interne al kernel.

Al contrario dei processi, il kernel Linux NON è preemptive.

Ritorno da interrupt

Dopo aver servito un'interruzione, il kernel si chiede se sia necessario il rescheduling (il codice della funzione di servizio per l'interruzione potrebbe aver richiesto il risveglio e un processo in attesa).

In caso affermativo chiama lo schedulatore.

Ritorno da interrupt

LA CHIAMATA ALLO SCHEDULATORE VIENE PERO' EVITATA
NEL CASO L'INTERRUZIONE FOSSE AVVENUTA ALL'INTERNO
DELLO SPAZIO DEL KERNEL.

NON E' POSSIBILE LEVARE IL CONTROLLO AD UN PROCESSO
MENTRE STA GIRANDO NELLO SPAZIO DEL KERNEL.

Cosa cambia col kernel 2.6

Due punti fondamentali:

- Scheduler $O(1)$: l'algoritmo che serve ad eleggere il task a cui passare il controllo diventa deterministico.
- Kernel preemptive: la rischedulazione può avvenire anche in occasioni differenti da quelle viste per il kernel 2.4

Non ci occupiamo di macchine SMP (che non sono propriamente "deep-embedded").

Schedulatore O(1)

Esistono (per-cpu) due array ordinati di priorità: l'array associato ai task che non hanno ancora esaurito il loro tempo (t. attivi) e l'array associato ai task che hanno già esaurito il loro tempo (t. non attivi).

I due array vengono visitati tramite puntatori. Quando l'array associato ai task attivi risulta vuoto, i puntatori vengono scambiati e l'array suddetto diventa l'array associato ai task non attivi.

Schedulatore O(1)

Il codice corrispondente (in kernel/sched.c, schedule()):

```
.....  
array = rq->active;  
if (unlikely(!array->nr_active)) {  
    /*  
     * Switch the active and expired arrays.  
     */  
    rq->active = rq->expired;  
    rq->expired = array;  
    array = rq->active;  
    rq->expired_timestamp = 0;  
    rq->best_expired_prio = MAX_PRIO;  
}  
.....
```

Schedulatore O(1)

I task vengono mantenuti già ordinati per priorità. Inoltre esiste una cache "a bitmap" associata agli array, la quale cache riduce il problema di trovare il prossimo task al problema di trovare il primo 1 data una serie di bit.

```
idx = sched_find_first_bit(array->bitmap);  
queue = array->queue + idx;  
next = list_entry(queue->next, task_t, run_list);
```

Schedulatore O(1)

Non appena un task (ri)prende il controllo si ricalcola la sua priorità e lo si reinserisce nella coda associata:

```
if (!rt_task(next) && next->activated > 0) {
    unsigned long long delta = now - next->timestamp;

    if (next->activated == 1)
        delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128;

    array = next->array;
    dequeue_task(next, array);
    recalc_task_prio(next, next->timestamp + delta);
    enqueue_task(next, array);
}
next->activated = 0;
```

Schedulatore O(1)

Non vediamo i dettagli su come si calcolano le priorità, ma in sostanza vengono penalizzati i task che caricano maggiormente la CPU.

Questo corrisponde ad un criterio di equità nella distribuzione del tempo macchina, che non è il criterio che ci serve per il tempo reale.

Tuttavia adottando una politica `SCHED_FIFO` per i processi che vogliamo siano real time e tenendo sotto controllo l'hardware ed il software installato, lo schedulatore O(1) ci può essere di aiuto grazie al suo algoritmo deterministico.

Kernel preemption

Come si è detto, fino al kernel 2.4 il kernel non era preemptive.

Esistevano però due famiglie di patch che avevano come obiettivo quello di abbassare la latenza:

- Low latency patch
- Preemption patch

Le due patch potevano essere applicate contemporaneamente in quanto non erano in conflitto.

Low latency patch

Identificava tutti i cammini "lunghi" all'interno del kernel, li spezzava ed inseriva delle chiamate allo schedulatore.

Preemption patch

Usava gli spinlock come punti di ingresso/uscita per delimitare le zone in cui era vietata la schedulazione. Questo perchè avere una macchina SMP è come avere una macchina UP in cui il kernel sia preemptible => Era possibile usare gli spinlock come protezioni.

Preemption nel 2.6

Nel kernel 2.6 al momento è stata integrata la preemption patch (abilitabile via CONFIG_PREEMPT).

In effetti c'è qualche obiezione. Il migliore argomento contro la preemption sembra essere:

- Dal momento che le latenze più lunghe derivano dalle sezioni di codice sotto spin lock, la latenza nel caso peggiore non viene migliorata abilitando la preemption del kernel.

Preemption nel 2.6

Alcune opinioni:

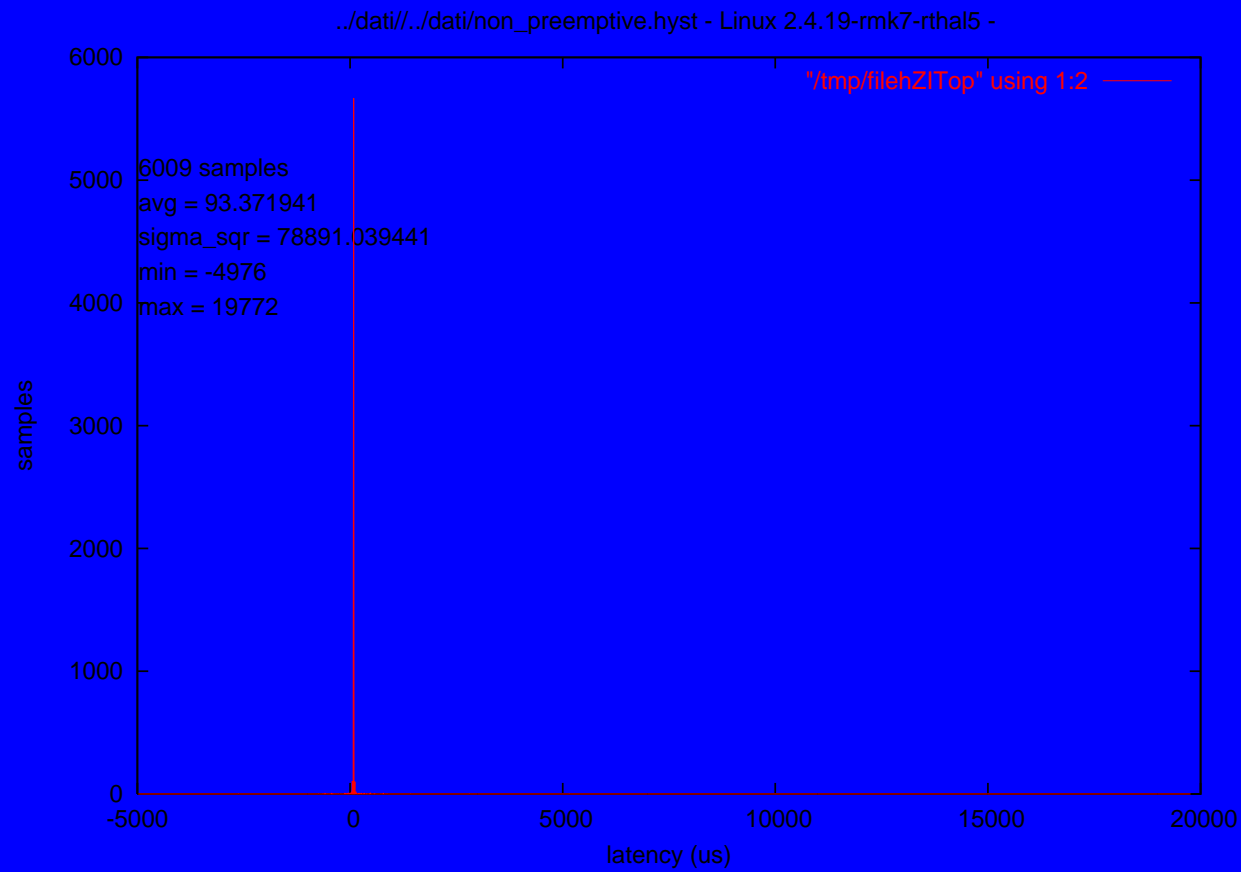
Ingo Molnar è autore di una Voluntary Preemption patch che aggiunge punti di preemption e sposta l'invocazione delle routine di risposta alle interruzioni nel contesto di un kernel thread chiamato irqd.

Andrea Arcangeli sostiene da tempo un metodo speculare rispetto a quello usato dalla preemption patch: lasciare la preemption sempre disabilitata ed abilitarla esplicitamente nelle sezioni di codice "lunghe" e non sotto spin lock (ad es. copie da/per spazio utente/spazio nucleo).

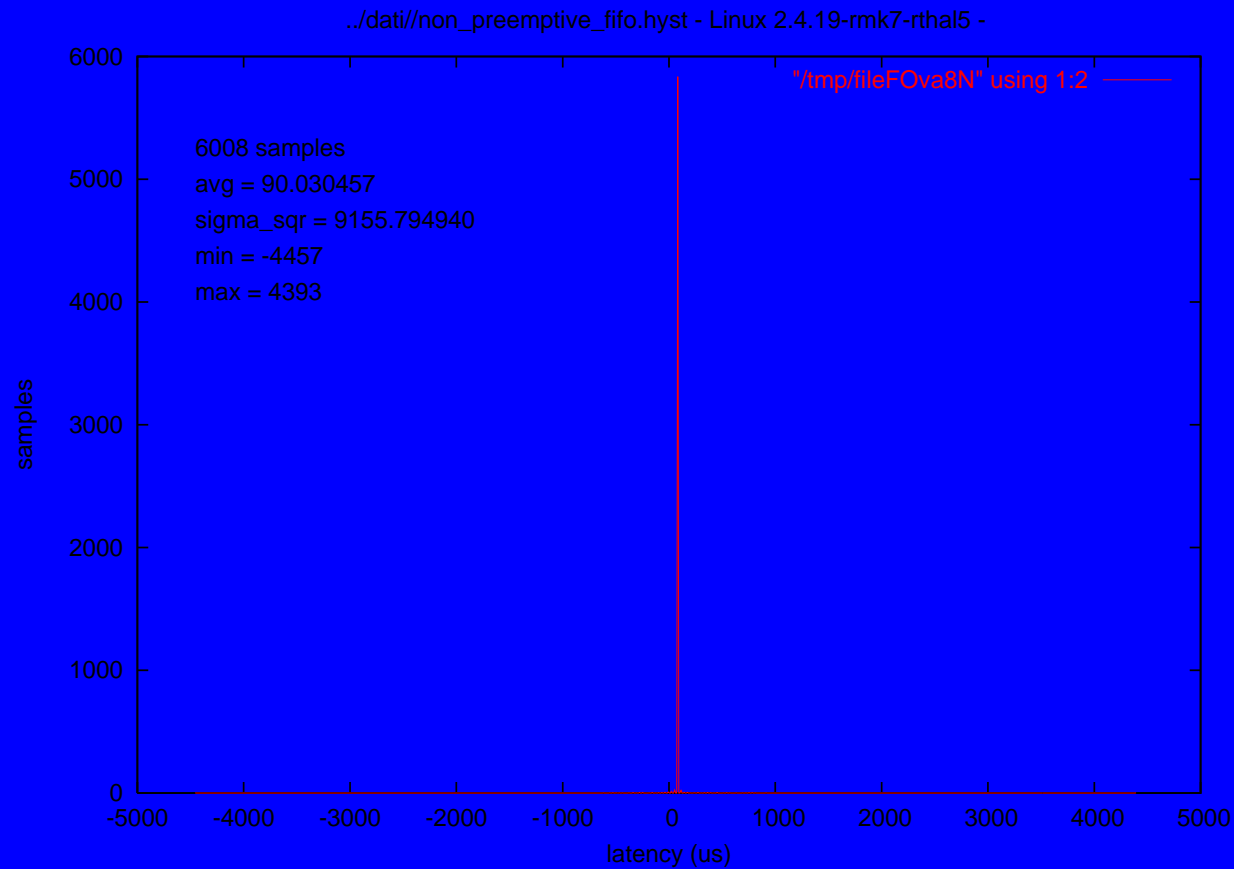
Preemption nel 2.6

Io mi sono limitato a fare qualche misura di latenza confrontando un 2.4 con un 2.6 (e `CONFIG_PREEMPT=y`, nessuna patch applicata) sulla mia scheda ARM.

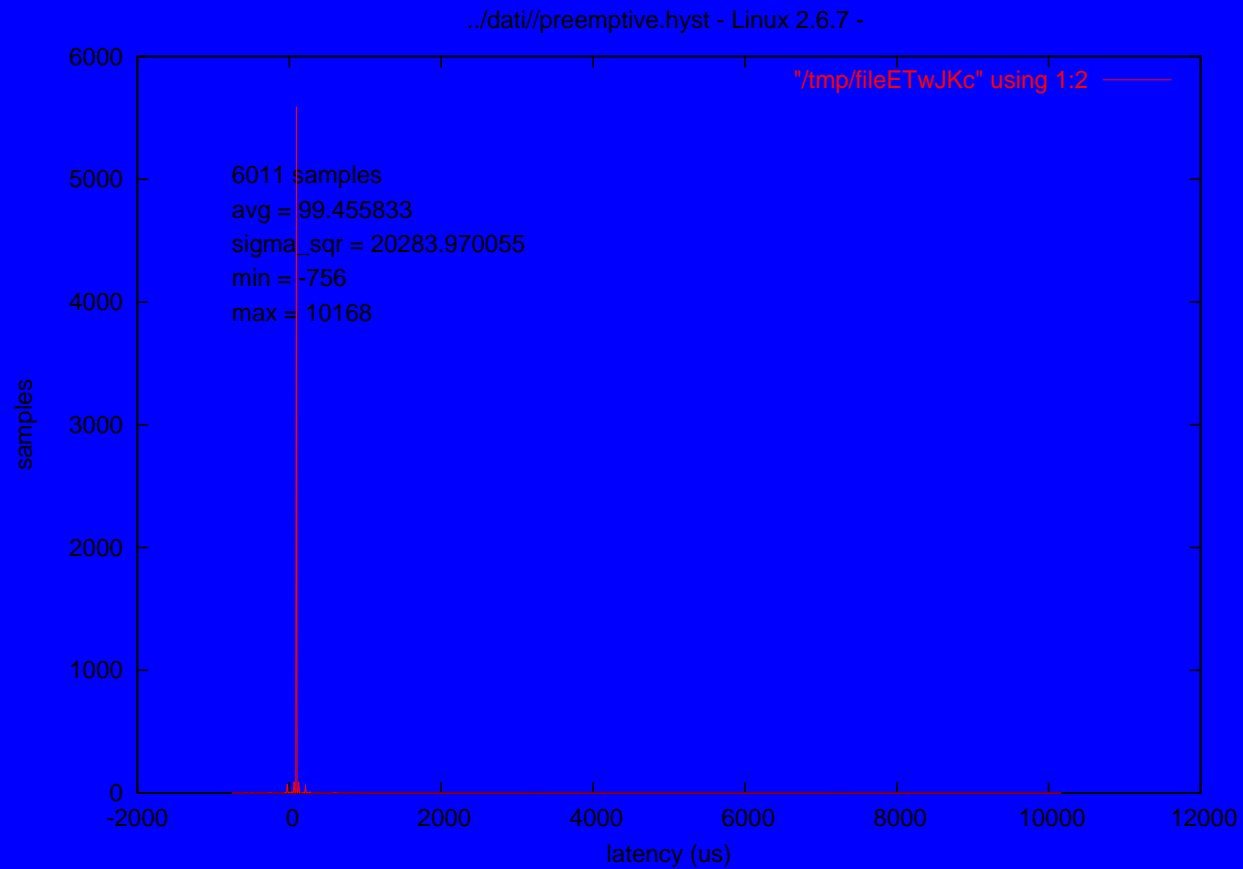
Kernel non preemptibile e politica SCHED_OTHER



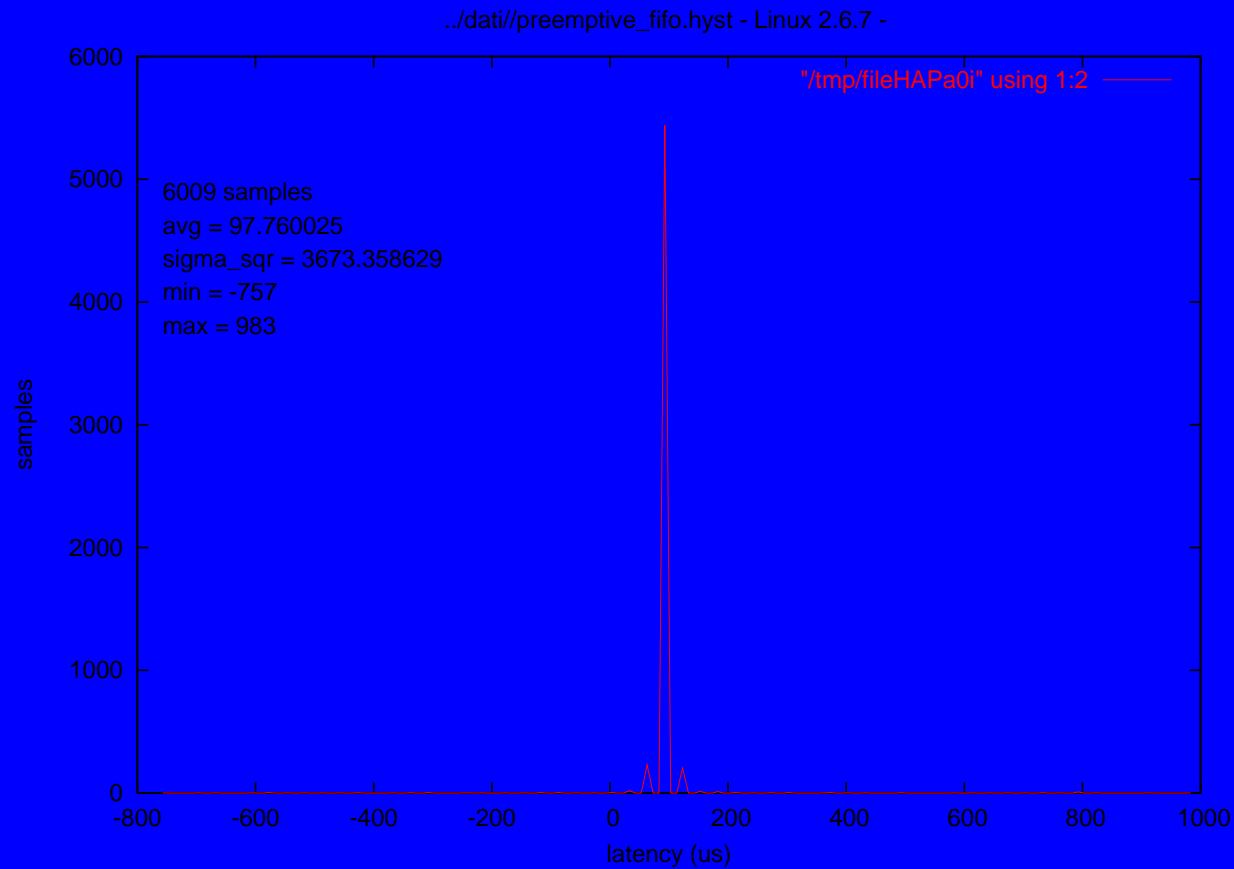
Kernel non preemptible e politica SCHED_FIFO



Kernel preemptibile e politica SCHED_OTHER



Kernel preemptibile e politica SCHED_FIFO



Timer POSIX

Implementazione norma POSIX 1003.1b-1993 Sezione 14.

E' possibile:

- Creare e gestire timer associandoli a degli allarmi
- Leggere ed impostare uno degli orologi.

Timer POSIX

Orologi previsti:

- `CLOCK_10MS` : per timer da 10ms
- `CLOCK_HIGH_RES` : per timer con la massima risoluzione supportata dal sistema
- `CLOCK_1US` : per timer con risoluzione 1us (se supportata dal sistema)
- `CLOCK_UPTIME`: riporta l'uptime
- `CLOCK_REALTIME`: per timer con risoluzione 1/HZ.

RTAI - concetti base

Finora si è visto come si stia tentando di migliorare i tempi di risposta del kernel Linux.

Tuttavia si parla sempre di applicazioni SOFT real time.

Nessuno dei metodi visti può portare al soddisfacimento di requisiti HARD real time.

RTAI - concetti base

RTAI affronta il problema del tempo reale "stretto" sostanzialmente introducendo un secondo kernel che si occupa degli eventi real time e lascia il controllo a Linux solo quando c'è il tempo.

RTAI - rthal

E' il primo "motore" su cui RTAI è stato basato.
Il cuore di rthal è la struttura struct rt_hal:

```
/*
 * RTAI
 * Definition of rthal.
 * Do not change this structure unless you know what you are doing!
 * Filled with values in arch/arm/kernel/irq.c
 */
struct rt_hal {
    void (*do_IRQ)(int, struct pt_regs*);
    long long (*do_SRQ)(int, unsigned long);
    int (*do_TRAP)(int, struct pt_regs*);
    void (*disint)(void);
    void (*enint)(void);
    unsigned int (*getflags)(void);
    void (*setflags)(unsigned int);
    unsigned int (*getflags_and_cli)(void);
    void (*fdisint)(void);
    void (*fenint)(void);
    volatile u32 timer_match; /* Soft next Linux timer-interrupt */
    void (*copy_back)(unsigned long, int);
    void (*c_do_IRQ)(int, struct pt_regs*);
} __attribute__((__aligned__(32)));
```

RTAI e le interruzioni

I campi `enint` e `disint` di `rthal` vengono riempiti con dei puntatori a funzione.

Normalmente `enint` e `disint` puntano a delle funzioni che abilitano e disabilitano le interruzioni della cpu.

Quando `rtai` inizia a funzionare (viene "montato") `enint` e `disint` vengono fatti puntare a delle funzioni che "fanno finta" di abilitare e disabilitare le interruzioni. In realtà le interruzioni della cpu rimangono abilitate anche quando Linux pensa di averle disabilitate.

Task real time

I task real time sono separati rispetto ai normali task Linux.

Nell'RTAI "antico" (che è però attuale per architetture non-x86) i task real-time girano esclusivamente nello spazio del kernel e comunicano con lo spazio utente (ad es. via FIFO o memoria condivisa).

RTAI e le interruzioni

Un'interruzione può anche essere condivisa tra RTAI e Linux.

In tal caso viene prima lanciato l'handler di RTAI e poi quello (o quelli) Linux.

RTAI ed i timer

Il componente base di RTAI è il timer, ossia ciò che produce la base dei tempi che fa girare il sistema.

Il timer RTAI può essere impostato per funzionare in due modi fondamentali:

- PERIODICO
- ONE-SHOT

RTAI ed i timer

MODO PERIODICO: il timer viene programmato una volta sola => poco flessibile nel caso ci siano diversi task con periodi non multipli tra loro, ma molto efficiente.

MODO ONE-SHOT: molto flessibile (il tempo della prossima interruzione viene ricalcolato di volta in volta), ma meno efficiente perchè si perde del tempo per la riprogrammazione del timer.

RTAI e gli scheduler

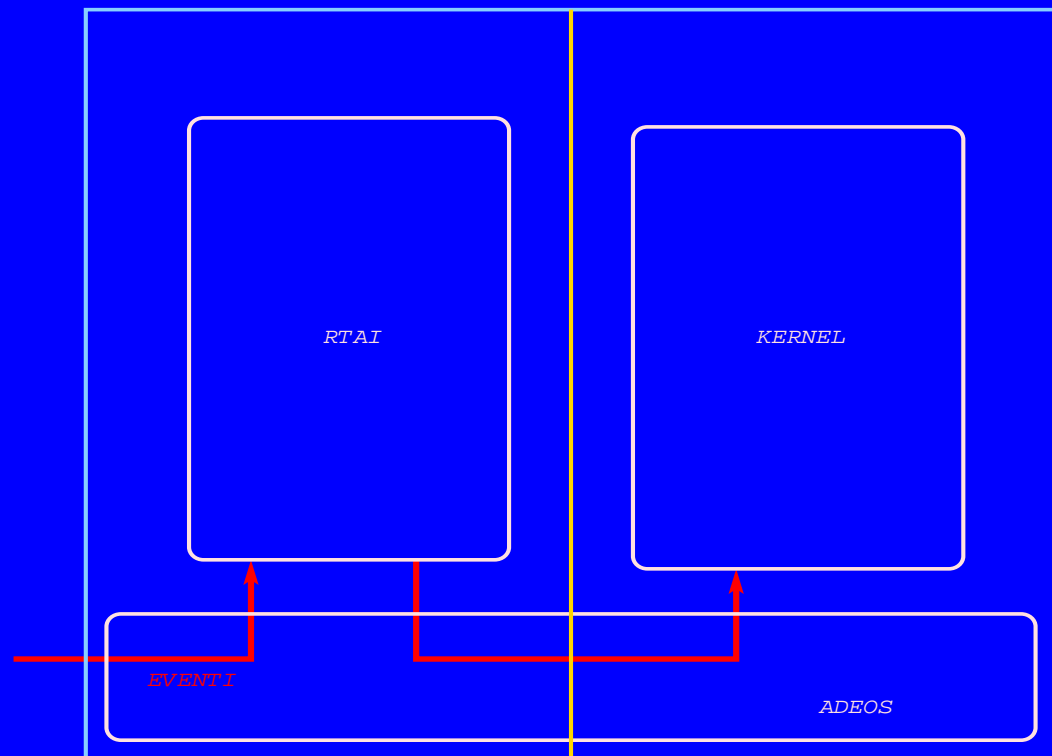
Per x86 sono disponibili diversi tipi di scheduler (UP, MUP, SMP, ...)

Per altre architetture solo UP.

ADEOS

E' il nuovo motore su cui è basato RTAI.

E' un "virtualizzatore" di interrupt ed eventi in genere.



ADEOS

Si noti che ADEOS può essere usato con RTAI ma anche da solo.

E' possibile intercettare interruzioni o eccezioni anche a scopo di profiling o tracciamento o debugging.

Per il momento RTAI+ADEOS è disponibile solo su x86.

LXRT

E' una tecnologia che consente di programmare in tempo reale nello spazio utente.

I task LXRT possono comunicare tra loro e con i task in kernel space tramite un'unica API.

I task hard real-time nello spazio utente non possono effettuare chiamate al kernel Linux, pena il declassamento a soft real-time.

Nuovi sviluppi.

- XENOMAI: un unico nucleo e più "pelli" => API diverse realizzabili a partire dagli stessi servizi di base (emulazione di altri RTOS).
- FUSION: l'integrazione delle diverse tecnologie (LXRT, XENOMAI), ed altri passi avanti (i servizi Linux saranno invocabili da contesti real-time senza declassare in modo permanente il chiamante a soft real time).

Conclusione.

Che fine fanno i tempi di latenza sulla nostra piattaforma ARM ?

Facendo girare il test di latenza RTAI si ottiene (valori in ns):

```
## RTAI latency calibration tool ##  
# period = 1000000 (ns)  
# avrgtime = 1 (s)  
# check overall worst case  
# do not use the FPU  
# start the timer  
# timer_mode is periodic
```

```
1969/12/31 01:16:53 min: 7080 max: 7081 average: 7080  
1969/12/31 01:16:54 min: -23438 max: 37598 average: 7080  
1969/12/31 01:16:55 min: -23438 max: 37598 average: 7080  
1969/12/31 01:16:56 min: -23438 max: 37598 average: 7080  
1969/12/31 01:16:57 min: -23438 max: 37598 average: 7080
```

.....
Sempre tra -23 e +38 microsecondi.