

GNU/Linux nelle applicazioni embedded

di Davide Ciminaghi

Introduzione

In questo articolo si darà una panoramica generale sulle opportunità, i vantaggi ed i limiti di utilizzo del sistema operativo GNU/Linux per i sistemi embedded. Verranno considerati sia gli aspetti più strettamente tecnici, sia quelli legati genericamente al modello di sviluppo software.

Si noti che parliamo qui di GNU/Linux (e non semplicemente di Linux) per sottolineare il fatto che trattiamo non solo del nucleo del sistema operativo, Linux appunto, ma anche delle applicazioni nonché dei sistemi di sviluppo: editor, sistemi di controllo della configurazione, compilatori, debugger, programmi di test. Buona parte di questi programmi sono il frutto diretto del progetto [GNU](http://www.gnu.org) (<http://www.gnu.org>), mentre la totalità di essi (compreso ovviamente Linux stesso) sono, o meglio possono essere, a seconda delle scelte degli utenti, software libero, distribuito secondo la licenza GNU GPL o altre licenze libere. Alcune riflessioni saranno infatti dedicate ai vantaggi dell'uso di software libero nello sviluppo di applicazioni embedded.

Non ci occuperemo invece specificamente di distribuzioni Linux embedded commerciali, ma molti dei concetti qui esposti si applicano ugualmente anche ad esse.

Kernel Linux e sistemi operativi embedded tradizionali

Il nucleo (kernel) Linux è funzionalmente equivalente ad un kernel Unix, e per questo motivo è molto diverso, nonché molto più complesso, della maggior parte dei tradizionali sistemi operativi per applicazioni specifiche (con esigenze di tempo reale o meno), nati per CPU con limitate risorse di calcolo e sistemi con piccole quantità di memoria (sia RAM sia ROM). Qui di seguito si riportano alcune delle differenze che appaiono tra le più significative, con particolare riferimento agli aspetti che di norma risultano meno familiari per gli utenti che provengano da altre esperienze.

Una prima differenza rilevante riguarda il fatto che il nucleo non è compilato assieme all'applicazione, ma vive in maniera indipendente. Il meccanismo di accesso ai servizi del nucleo è costituito dalle chiamate di sistema, implementate attraverso interruzioni software.

Inoltre il codice del nucleo funziona sempre in modalità "supervisore": ha quindi accesso a tutte le risorse hardware della macchina, gestisce direttamente interruzioni ed eccezioni e si occupa di tutte le funzionalità di basso livello. Al contrario, il codice dell'applicazione viene eseguito in modalità utente, ed è quindi limitato per quanto riguarda l'accesso fisico al sistema, oltre a non poter direttamente gestire interruzioni ed eccezioni.

Dal punto di vista della scalabilità Linux ha lo svantaggio di richiedere sempre un file system per poter funzionare. Si noti tuttavia che ciò non implica necessariamente il fatto che il sistema sia dotato di un dispositivo di memorizzazione di massa. È infatti possibile usare svariati supporti fisici per memorizzare un file system: RAM, ROM, memoria FLASH, il file system di un'altra macchina con cui si è collegati via rete, e così via.

Una caratteristica del kernel Linux è quella di permettere di essere esteso mentre il sistema funziona (mediante il meccanismo dei moduli). Questo consente il supporto di piattaforme differenti dal punto di vista delle periferiche hardware senza dover ricompilare il nucleo.

I driver di periferica fanno di norma parte del codice del kernel e sono accessibili come voci del file system. Uno dei grossi vantaggi dell'uso di software libero nei sistemi embedded è che lo sviluppo dei driver e in generale di porzioni/moduli del kernel è ampiamente facilitato rispetto ai sistemi tradizionali vista la disponibilità totale dei sorgenti del kernel, e quindi di un numero notevole di esempi funzionanti sul campo, nonché di un'ampia documentazione.

Nel caso di utilizzo di macchine con MMU (Memory Management Unit), sempre più frequente in virtù dei costi via via più bassi del silicio, è possibile sfruttare funzioni molto avanzate sia dal punto di vista della protezione della memoria, e quindi dell'affidabilità del sistema, sia da quello delle funzionalità. In particolare la disponibilità di chiamate come `mmap()`, che consente di mappare i contenuti di un file o genericamente di un dispositivo su un intervallo di indirizzi virtuali, rende il kernel Linux particolarmente attraente.

Per ciò che concerne i processi, il modo Unix/Linux di crearne di nuovi riesce di solito un po' ostico ai programmatori embedded tradizionali. Quello che Linux fa è "clonare" un processo per poi sostituirne l'immagine con quella del programma da eseguire, contrariamente a quanto accade con i sistemi operativi basati su thread, in cui la creazione di un nuovo thread comporta una sola chiamata uno dei cui parametri è già l'indirizzo del punto di ingresso del thread stesso. Si tenga tuttavia presente che anche sotto Linux è possibile la programmazione a thread. Esistono infatti diverse implementazioni dei cosiddetti "thread POSIX", definiti dalla norma POSIX 1003.1c.

Infine Linux dispone di una grande quantità di meccanismi di comunicazione interprocesso, eredità dei suoi predecessori: si va dai segnali, la forma più arcaica, ad altre più sofisticate: pipe, fifo, semafori, code di messaggi, memoria condivisa, socket, eccetera. La varietà di possibili soluzioni impone all'utente una buona conoscenza di tutte le alternative al fine di scegliere la più adatta alle esigenze dell'applicazione.

Il livello applicativo

Soprattutto per ciò che concerne la rete, il sistema GNU/Linux fornisce una grande quantità di applicazioni già scritte e collaudate da tempo, ma l'aspetto più interessante di tutti sta nel cambio radicale di ottica rispetto all'approccio tradizionale.

La "novità" sta soprattutto nel fatto che la filosofia Unix (da cui Linux discende da un punto di vista tecnico) privilegia la modularità. Di conseguenza, ciò che verrebbe visto da un programmatore embedded tradizionale come un'unica applicazione da scrivere tutta in C, può essere spezzato in una serie di programmi tra loro interagenti, alcuni dei quali già esistenti ed altri da scrivere in linguaggi diversi a seconda dell'opportunità tecnica.

È inutile rimarcare tutti i vantaggi dell'approccio modulare. Quello che invece è importante sottolineare è che, con l'approccio presentato, la fase di test di modulo risulta semplificata perché il modulo stesso è già esso stesso un programma eseguibile, rendendo quindi inutile la costosa scrittura di programmi di test ad hoc.

Infine evidenziamo il fatto che l'uso di linguaggi di scripting (ad esempio script della shell, tcl e le sue varianti, perl, eccetera) può risolvere un gran numero di problemi che risultano ostici in C, come ad esempio lavorare con stringhe ed espressioni regolari.

Le applicazioni in tempo reale

Essendo un kernel multitasking e multiutente, Linux, come altri Unix, non è stato scritto pensando al tempo reale "stretto", ossia ad applicazioni in cui il mancato rispetto dei tempi di esecuzione specificati porti ad un malfunzionamento grave.

I problemi della schedulazione real time sotto Linux

Il componente del nucleo del sistema operativo che decide istante per istante quale sia il processo che debba avere il controllo della CPU si chiama *schedulatore*. Ad ogni processo vengono assegnati due attributi fondamentali: una priorità ed una politica di schedulazione. Per motivi di brevità non entreremo nel merito di come questi due attributi vengano gestiti, ma si tenga presente che gli obiettivi dello schedulatore, salvo casi particolari relativi solo a processi appartenenti all'utente di root, sono fondamentalmente quello di un'equa distribuzione di risorse tra i diversi utenti, nonché quello che consiste nell'evitare che un solo processo si impadronisca di tutto il tempo macchina, per evidenti ragioni di sicurezza.

Al lettore interessato e con un sistema Linux a portata di mano consigliamo la lettura della pagina di manuale relativa alla chiamata di sistema `sched_setscheduler(2)`. Per tutti gli altri lettori, diciamo che la conseguenza di quanto si è detto in precedenza è il fatto che lo schedulatore Linux non ha il concetto di "deadline", ossia tempo entro il quale una certa attività andrà svolta, e non è quindi adatto ad applicazioni in tempo reale.

Si potrebbe a questo punto pensare che, sostituendo lo schedulatore tradizionale con uno opportunamente riscritto, sia possibile aggirare il problema, ma questo non è vero a causa di un secondo ordine di motivi. Bisogna infatti chiedersi quando possa avvenire un cambio di contesto, ossia quando effettivamente il controllo della CPU possa passare da un processo ad un altro. La risposta è che i cambi di contesto, sotto Linux, possono avvenire solo al passaggio dallo spazio del kernel allo spazio delle applicazioni, ossia ad esempio al ritorno da una chiamata di sistema oppure al ritorno da un interrupt handler quando l'interruzione avviene mentre la CPU sta eseguendo codice dell'utente. Motivo di ciò è l'esigenza di preservare l'integrità delle strutture dati interne del nucleo, integrità che verrebbe compromessa se il cambio di contesto potesse avvenire in qualsiasi momento.

A causa di questo fatto, non è possibile calcolare un limite superiore per il tempo che intercorre tra un cambio di contesto ed il successivo. Prendiamo per esempio il caso in cui avvenga un grande numero di interruzioni annidate mentre la CPU si trova già all'interno del codice del kernel. Se si verifica una condizione di questo genere, non si può prevedere quando si ritornerà al codice utente e quindi procedere ad un eventuale cambio di contesto.

Ad esempio se il controller IDE continua ad interrompere senza che sia possibile uscire dallo spazio del kernel, il processo utente di elaborazione (anche se a priorità più alta possibile) non può riprendere il controllo perché lo scheduler non può girare. Per maggiori informazioni si può consultare l'articolo "*Linux e real time (prima parte)*" su <http://www.freego.it/articles.php>.

Le soluzioni

Nel corso degli anni si sono andate affermando due diverse famiglie di tecniche:

- 1) quelle che tendono a modificare il nucleo in modo da aumentare le possibilità per i cambi di contesto,
- 2) quelle che prevedono l'introduzione di un secondo componente (un secondo kernel per esempio) adatto al tempo reale che veda Linux come la propria attività meno prioritaria.

La prima famiglia di tecniche ha dato origine a due insiemi di modifiche (patch) del kernel chiamate "*preemption patch*" e "*low-latency patch*". La prima patch identifica le zone del kernel all'interno delle quali i cambi di contesto sono effettivamente impossibili, ed abilita la schedulazione al di fuori di tali zone. La seconda patch, invece, spezza i cammini troppo lunghi del codice del kernel, introducendo delle chiamate esplicite allo schedulatore. A partire dalla versione 2.5.4-pre6 del kernel, le due patch sono state riunite in una sola e fanno parte integrante dei sorgenti del kernel stesso. Si noti tuttavia che tutte queste tecniche sono solo migliorative, e quindi non adatte ad applicazioni con esigenze strette di tempo reale.

La seconda famiglia di tecniche è invece rappresentata, nell'ambito del software libero, dal progetto [RTAI](http://www.rtai.org) (<http://www.rtai.org>). Al momento RTAI si basa su un componente software chiamato [ADEOS](http://home.gna.org/adeos) (<http://home.gna.org/adeos>), che si occupa in sostanza di intercettare ed elaborare tutti gli eventi significativi (interruzioni ed eccezioni, fondamentalmente),

e poi passarli in modo selettivo al kernel RTAI ed al kernel Linux. Questo consente, separando opportunamente le parti dell'applicazione veramente in tempo reale da quelle non real-time, di ottenere tempi di risposta deterministici. Per ulteriori informazioni sulle strade possibili per rendere Linux real time e un esempio di utilizzo di RTAI si può consultare l'articolo "*Linux e real time (seconda parte)*" su <http://www.freego.it/articles.php>.

Portabilità

Sebbene in origine sia nato come un sistema operativo esclusivamente per piattaforma x86 (e in particolare PC), il kernel Linux è stato portato su un grande numero di architetture differenti, ed anche su macchine non dotate di MMU.

Anche a livello di applicazioni, sono disponibili distribuzioni Linux libere che coprono diverse architetture. Ad esempio la distribuzione Debian, pur non specifica per l'embedded, dispone di pacchetti per svariate piattaforme comunemente impiegate su macchine ad impiego specifico (x86, ARM, PowerPC, SH ed altre).

Anche a livello di librerie C, esistono progetti liberi nati per dare origine a librerie particolarmente poco esigenti dal punto di vista delle risorse richieste.

Limiti di applicabilità

Dal punto di vista tecnico bisogna tenere presente che Linux non è un sistema operativo che possa essere utilizzato per sistemi con risorse limitatissime.

L'aspetto più delicato è costituito dalla quantità di memoria a disposizione: di norma si prende come limite inferiore quello dei 2MB di memoria RAM + 2MB di memoria ROM (o FLASH), anche se il limite effettivo varia a seconda della piattaforma.

La potenza di calcolo richiesta alla CPU dal kernel in se stessi non rappresenta invece un fattore particolarmente critico.

Sistemi di sviluppo

Nonostante venga qualche volta considerato "scomodo" e/o "poco usabile", Linux rappresenta una piattaforma ideale anche per ciò che concerne la macchina host, ossia quella sulla quale si sviluppa il codice che girerà sul campo. Questo è vero anche per quello che riguarda i compilatori, i sistemi di controllo della configurazione, il test ed il debugging.

Compilazione

L'insieme di binutils (assembler, linker ed altri programmi di utilità) e di gcc (GNU compiler collection, il compilatore GNU) costituisce la toolchain standard sotto Linux. Si tratta di strumenti di sviluppo di livello professionale, in grado di produrre codice binario per un numero vastissimo di CPU e sistemi operativi.

Debugging e test

Il debugger standard sotto Linux fa parte del progetto GNU e si chiama gdb. La versione "base" di gdb prevede l'uso da linea di comando, ma esistono anche versioni grafiche (come ad esempio Insight e ddd).

Inoltre sotto Linux il debugging delle applicazioni può essere effettuato sia mediante il debugger vero e proprio sia attraverso strumenti di tracing come ad esempio strace(1), che evidenzia la sequenza delle chiamate di sistema effettuate dall'applicazione sotto esame.

Per ciò che consente il testing, soprattutto se si parla di applicazioni di rete, la disponibilità di linguaggi di scripting e di programmi come netcat(1) o tcpdump(1) rende possibile la scrittura rapida di programmi di test anche relativamente complessi. Il debugging del kernel e dei suoi moduli è invece più problematico. Per scelta progettuale, infatti, il kernel non dispone in modo nativo di un modulo per il debugging. Si ricorre quindi di norma ad altri metodi, tutti sostanzialmente riconducibili ad attività di tracciamento. Esistono però anche patch del kernel che consentono il debugging vero e proprio.

Strumenti di supporto

Per chiudere, Linux fornisce strumenti validi anche per ciò che concerne l'editing (ad esempio emacs), il controllo della configurazione dei moduli (cvs), la stesura della documentazione (texinfo, sgmttools), ed in generale tutto ciò che concerne in generale lo sviluppo di software.

L'approccio allo sviluppo

Il modello di sviluppo da adottare quando si utilizza software libero deve essere diverso rispetto a quello che si impiega tradizionalmente.

Salvo infatti che non ci si affidi ad una distribuzione commerciale o si stipuli un contratto di consulenza, il sistema GNU/Linux è distribuito senza garanzia di funzionamento. Questo viene spesso considerato da molti utenti, o meglio dai loro manager, come un difetto. In realtà non si riesce ad afferrare che il software libero fornisce anche dei diritti e di conseguenza delle opportunità in più rispetto al software proprietario: la disponibilità del codice sorgente per la lettura, lo studio e le modifiche consente di correggere malfunzionamenti, effettuare personalizzazioni, nonché imparare dal codice e dagli errori altrui e in qualche caso contribuire attivamente allo sviluppo.

Infine è bene tener presente che sia il kernel sia gli altri pacchetti liberi di un sistema GNU/Linux sono opera di comunità di sviluppatori, con i quali si può stabilire un contatto per ricevere aiuto, scambiare esperienze e consigli. Essere utenti di software libero senza mettersi in un'ottica di collaborazione con altri utenti e sviluppatori può essere fonte di insuccessi e frustrazione.

L'autore:

Davide Ciminaghi lavora come libero professionista su progetti in campo embedded utilizzando GNU/Linux e più in generale software libero.

E' raggiungibile agli indirizzi e-mail ciminaghi@prosa.it oppure ciminaghi@gnudd.com

APPROFONDIMENTI:

Il real-time

Cosa significa che un pezzo di software deve essere "real-time"? Significa in parole povere che il software deve essere in grado di garantire che una certa cosa verrà fatta entro un certo tempo. Quanto tempo effettivamente non importa. Real-time non è sinonimo di veloce, bensì di deterministico, ossia prevedibile e garantibile a priori.

E quando si parla di "Hard real-time"? In sostanza significa che, se il software non completa i suoi compiti entro il tempo stabilito dalla specifica, il sistema controllato collassa in modo catastrofico.

Per finire, manca la definizione di "Soft real-time": in contrasto con "Hard real-time", si vuole indicare che al software non è richiesto di rispondere in tempi certi pena la morte del sistema sotto controllo. Semplicemente si vuole che il software risponda in tempo nella maggior parte dei casi. Un esempio tipico è costituito dalla riproduzione audio o video: se un brano musicale si interrompe per un attimo la conseguenza è solo un po' di fastidio per l'utente, nulla di irreparabile.

Il "real-time" e lo scheduling sotto Linux

Lo scheduling riguarda le modalità secondo cui il sistema decide di assegnare la CPU ai processi pronti per l'esecuzione. Ad ogni processo corrispondono due parametri fondamentali: la politica di schedulazione e la priorità. Il kernel Linux nella sua versione "standard" permette tre politiche di schedulazione, divise in due categorie: politiche a priorità dinamica (OTHER) e politiche a priorità statica (FIFO e RR).

Più in dettaglio le politiche di schedulazione sono le seguenti:

- **OTHER:** A dispetto del nome, è la politica standard usata per tutti i processi. Il tempo viene visto dallo scheduler come suddiviso in "epoche" all'interno delle quali ad ogni processo viene assegnato un tempo massimo di funzionamento. La priorità di un processo è la somma di un termine fisso e di un termine che diminuisce man mano che il processo consuma il proprio tempo di CPU.
- **FIFO:** È la politica più "ingorda". Un processo con politica FIFO a priorità più alta degli altri può monopolizzare completamente la CPU.
- **RR (Round-Robin):** È simile alla FIFO, ma in questo caso il processo vincente può funzionare per un periodo massimo di tempo, scaduto il quale il controllo gli viene levato.

I processi a priorità statica sono visti come processi "real-time" (nell'accezione di SOFT real-time). Quello che è importante tenere presente è che:

- Linux è un sistema preemptible per quanto riguarda i processi: questo significa che 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. Questo significa che anche un processo "real-time" deve attendere che lo scheduler gli assegni la CPU e questo può avvenire con tempi massimi (*deadline*) non predicibili ("soft" real-time).

In particolare lo scheduler Linux può girare solo nelle seguenti situazioni:

- 1) al ritorno da interrupt o da eccezione (se questa avviene durante l'esecuzione di codice utente),
- 2) al ritorno dalle chiamate di sistema,
- 3) in seguito a chiamate interne al kernel.

Ne segue che, ad esempio, se una periferica continua a lanciare *interrupts* senza che sia possibile uscire dallo spazio del kernel (Linux ammette che le interruzioni possano essere tra loro annidate), lo scheduler non può girare e quindi il processo di elaborazione "real-time" (anche se alla priorità più alta possibile) non può riprendere il controllo della CPU.

RTAI: concetti base

Si tratta di una delle soluzioni attualmente più efficaci e diffuse per rendere Linux un sistema veramente real-time. Essa prevede che un secondo kernel, di tipo real time, prenda il controllo effettivo della macchina e faccia girare Linux come la propria applicazione a priorità più bassa.

Inoltre il kernel Linux perde il potere di disabilitare le interruzioni, che vengono controllate dal kernel real time.

La struttura più importante in RTAI è la struct `rt_hal`, che "descrive" l'accesso all'hardware. Questa struttura contiene una serie di puntatori a funzioni e ad altre strutture dati (ad esempio la Interrupt Descriptor Table nell'architettura x86).

Durante il normale funzionamento della macchina, i campi di struct `rt_hal` puntano alle implementazioni Linux dei metodi. Non appena RTAI comincia a funzionare, i puntatori vengono "dirottati" sulle funzioni RTAI. Una conseguenza importante di questo sta nel fatto che Linux non ha più il controllo diretto sull'abilitazione e la disabilitazione delle interruzioni. Inoltre le interruzioni stesse vengono intercettate da RTAI che sostituisce gli handler Linux con dei dispatcher.

A questo punto:

- 1) se l'interruzione appartiene a Linux, viene messa in coda per essere gestita non appena possibile.
- 2) se l'interruzione appartiene a RTAI, viene chiamato immediatamente lo handler opportuno.
- 3) se l'interruzione appartiene sia a RTAI sia a Linux vengono chiamati i rispettivi handler. Si noti che un'interruzione Linux può essere condivisa da più handler.

Le interruzioni per RTAI possono essere di tre tipi:

- IPI (InterProcessor Interrupt), di cui non ci occupiamo.
- Chiamate di sistema Linux: vengono trattate come interruzioni "solo Linux" e servono ad interruzioni abilitate (quando è il momento giusto, ovviamente), per emulare la trap di Linux.
- Interrupt delle periferiche: possono essere solo Linux, solo RTAI o entrambe le cose.

Esistono anche delle chiamate di sistema RTAI (ad es. usate per la memoria condivisa) che utilizzano delle interruzioni software separate per cui è possibile (lavorando direttamente sulla IDT del processore) rispondere alle interruzioni senza passare per alcun dispatcher.

Per maggiori informazioni consultare il sito www.rtai.org o l'articolo "*Linux e real time (seconda parte)*" su <http://www.freego.it/articles.php>.