Riassunti SERT - parte EMBEDDED

Introduzione ai sistemi embedded - E1

Un sistema embedded è un sistema programmabile specializzato, progettato per svolgere compiti specifici senza la possibilità di essere riprogrammato dall'utente. Questi sistemi interagiscono intensamente con l'ambiente circostante e hanno interfacce utente quasi invisibili. Sono strutture di supporto per il funzionamento di applicazioni con una forte interazione esterna e una ridotta interazione con l'utente. Esempi comuni di sistemi embedded si trovano in contesti come l'automotive, l'avionica e l'automazione industriale.

A differenza dei calcolatori, che sono versatili, riprogrammabili e conformi a standard di riferimento, i sistemi embedded hanno compiti specifici, come l'elaborazione di segnali, il trasferimento e la memorizzazione dei dati. Dal punto di vista commerciale, la creazione di un sistema embedded dipende da fattori come il costo, il tempo necessario per la sua realizzazione, la durata prevista e la quantità di unità vendute.

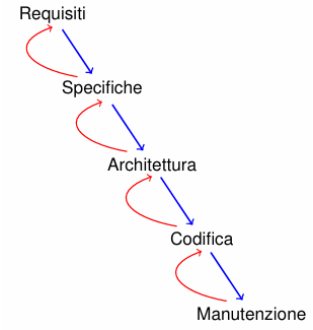
Dal punto di vista hardware, si tende a risparmiare sulle interfacce di comunicazione, incluse quelle utente, limitandosi spesso all'uso di LED, riducendo i consumi energetici e le dimensioni del dispositivo. Dal punto di vista software, le funzioni di calcolo, comunicazione e memorizzazione dipendono dagli obiettivi specifici del sistema. Esistono requisiti di Qualità del Servizio (QoS) e spesso un sistema embedded può anche aggiornarsi autonomamente.

Un parametro fondamentale per i sistemi embedded è la **dependability**, che include aspetti come affidabilità e manutenzione. Questo richiede una valutazione dei possibili guasti, dei tempi di riparazione, della probabilità di funzionamento corretto del sistema, della sicurezza per le persone e della prevenzione di usi illegali.

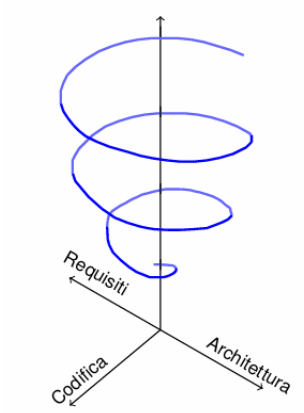
Tecnologie per sistemi embedded - E2

Per realizzare un'applicazione embedded, è necessario identificare sia i **requisiti funzionali**, come i consumi energetici e le capacità di elaborazione, sia i **requisiti non funzionali**, come i costi e il numero di pezzi da produrre. È inoltre fondamentale definire una metodologia di sviluppo basata sugli strumenti e le risorse disponibili. Nel caso dei sistemi embedded, si applica una metodologia ibrida che integra sia la parte hardware che la parte software. Esistono due approcci principali per lo sviluppo dei sistemi embedded:

- **A cascata:** Ogni livello richiede la definizione di quelli precedenti.



- **A spirale**: iterazione su tutte le fasi, non “torno indietro” perché faccio tutto insieme.



Nello sviluppo di un sistema embedded, è cruciale tenere in considerazione sia i requisiti funzionali che non funzionali. Partendo da una specifica RTLevel, si devono definire gli stati e sintetizzare **logiche indipendenti dalla tecnologia** (es: le porte logiche) e **logiche dipendenti dalla tecnologia** (considerazioni sul posizionamento del circuito). Successivamente, si delineano i "percorsi" sul circuito prima della stampa. È fondamentale stabilire il **ruolo del software** (ad esempio, un sensore che invia dati) e **dell'hardware** (che agisce in base ai dati), comprendendo così che questi aspetti sono sia separati che interdipendenti, richiedendo un approccio di co-design.

Spesso, si lavora su circuiti già pronti, partendo da una piattaforma generica che viene poi utilizzata per progettare qualcosa di specifico. Le tecnologie principali da considerare includono:

1. **ASIC (Application-Specific Integrated Circuit)**:
    - **Caratteristiche**: Un circuito specializzato per una funzione specifica, non riprogrammabile, con costi elevati e prestazioni elevate.

      Si sono susseguite le tecnologie:

        - **Standard Cell**: Circuiti digitali riutilizzabili, raccolti in librerie, con la stessa altezza e ridotta distanza tra le celle.
        - **Gate Array**: Chip parzialmente fabbricato, un'evoluzione degli ASIC.

        - **Attualmente** gli ASIC sono meno utilizzati oggi in favore delle **logiche programmabili** più flessibili e adattabili.

2. **Logiche Programmabili**:
    - **Caratteristiche**: Chip programmabili con risorse logiche (ovvero celle) configurabili e linee di interconnessione utilizzabili.
    - **Tipi di programmazione**:
        - **OTP (One-Time Programmable)**: Configurazione irreversibile.
        - **Configurazione multipla a circuito spento**.
        - **Configurazione multipla a circuito attivo**.
    - Si differenziano in base alla **complessità delle celle**: Le risorse logiche possono essere semplici o complesse.

3. **Microprocessori**:
    - **Caratteristiche**: Molto flessibili e facilmente mantenibili, ma con prestazioni inferiori rispetto ai circuiti "ad hoc".
    - **Tipi**:
        - **Dedicated**: Progettati per compiti specifici.
        - **General Purpose**: Utilizzati per una varietà di applicazioni.
    - **Modalità di acquisto**: Disponibili sia fisicamente che a livello di schemi (design schematics).
    - Rilevanti sono le tipologie **DSP** per le elaborazioni numeriche, basate su $z_{t+1} = z_t + x \cdot y$, e la tipologia **NP** per le reti, oltre ai **microcontrollori**, che si interfacciano con più periferiche.

Sistema Bare Metal - E03a

BeagleBone Black (BBB)

La BeagleBone Black è una scheda singola open-source con un processore ARM che non richiede ventole.

Processo di Avvio

1. All'accensione, viene eseguito un programma nella ROM, che poi carica il contenuto della memoria eMMC o microSD.
2. **uBoot**: Questo bootloader carica il sistema bare-metal dalla eMMC e può ricevere comandi dall'utente o eseguire operazioni automatiche.

Device Tree

- Descrive le periferiche hardware della BBB, permettendo al sistema operativo di interfacciarsi con esse. E' un approccio "statico", non è "on-line".

Architettura e Ambiente di Esecuzione

- **Architettura**: RISC a 32 bit.
- **Ambiente di Esecuzione (SERT)**: Permette di interfacciarsi con l'hardware in tempo reale (predicibile e validabile). Si programma in C o Assembly.

Sviluppo del Software

1. Il codice è scritto su un PC (host) e tradotto in codice macchina per la BBB usando un cross-compiler.
2. Il feedback è fornito tramite LED luminosi sulla scheda.

ARM - E03b

ARM e Smartphone

Gli smartphone utilizzano processori ARM, che hanno avuto una diffusione rapidissima grazie alla licenza ARM che vende la proprietà intellettuale piuttosto che il chip stesso, permettendo una rapida evoluzione.

Famiglie ARM

I processori ARM sono raggruppati in famiglie CORTEX:

- **A (Application)**: Per applicazioni complesse.
- **R (Real-Time)**: Per sistemi real-time.
- **M (Microcontroller)**: Per sistemi embedded.

Caratteristiche Tecniche

- **Architettura**: 32 bit, basata su RISC, supporta sia Little-Endian che Big-Endian.
- **Flag**: Supporta flag per eseguire o meno le istruzioni. Non presenta shift espliciti. Include moltiplicazioni, ma non divisioni.

Registri

- **Generali**: r0, r1, ..., r15

r10	r11	r12	r13	r14	r15
size stack	frame pointer	invocazione procedure	stack pointer	link register (indirizzo di ritorno)	program counter (indirizzo della seconda istruzione sotto quella in esecuzione)

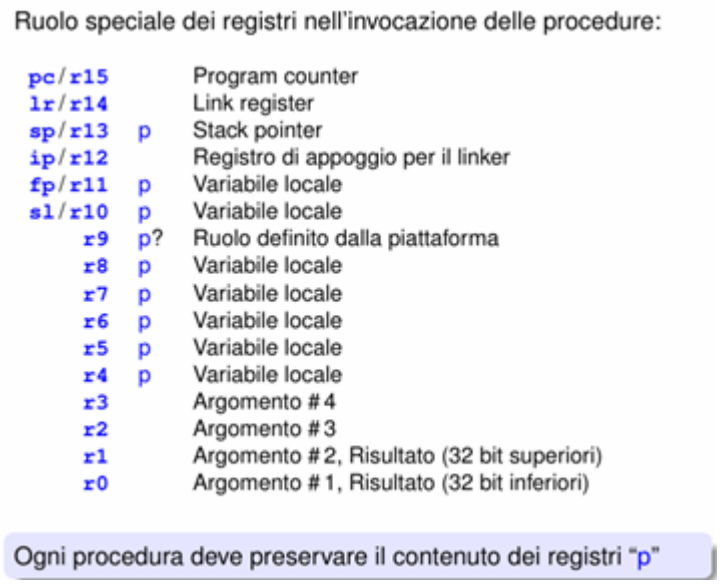
- **Altri registri importanti**:
  - **cpsr** (Current Program Status Register) con bit di condizione (Nullo, Zero, Carry, Overflow) e bit per disabilitare interruzioni.
  - Cinque registri **spsr** che preservano cpsr. Saved Program Status Register

Istruzioni

- **Load/Store**:
  - Caricamento da memoria: `ldr`
  - Memorizzazione: `str`
  - Nota: Indirizzi a 32 bit, devono essere caricati in un registro, non posso inserirli insieme all'istruzione, perchè eccedo i 32 bit.
- **Trasferimento Interi 8-bit**:
  - `ldrb`, `ldrsb`, `strb`
- **Shift**:
  - Destra: `lsr`
  - Sinistra: `lsl`
- **Salto e Procedure**:
  - Salto: `b [dovesalto]`
  - Invocazione procedura: `bl` (devo salvare l'istruzione successiva in `lr/r14`, non nello stack). Non esiste un'istruzione `return`, quindi devo gestire manualmente i salti e i ritorni delle procedure.

ABI - Application Binary Interface

Definisce aspetti dell'ambiente di esecuzione non definiti in hardware, come passaggio dei parametri o formati.



ARM e Gestione delle Procedure

Salvataggio e Ripristino dei Registri

Durante l'invocazione di una procedura:

1. **Salvataggio**: Prima di saltare alla procedura, salviamo il contenuto di alcuni registri sullo stack.
   - `push`: `stmfd sp!,{r4-r8, r10-r11, r14}`
2. **Ripristino**: Al termine della procedura, ripristiniamo i registri dallo stack.
   - `pop`: `ldmfd sp!,{r4-r8, r10-r11, r15}`
3. Se la procedura richiama altre procedure, occorre preservare anche `lr` (r14).

Stati del Processore ARM

Stati principali:

- **User**: Per applicazioni normali.
- **System**: Per processi privilegiati.
- **FIQ**: Interruzioni rapide.
- **IRQ**: Interruzioni normali.

In modalità FIQ, non è necessario salvare i registri r8 - r12.

Eccezioni e Vettori

Esistono sette tipi di eccezioni, ordinate per priorità, tra cui: (su slide fanfa sono complete)

- **RESET**
- **Data Abort**
- **IRQ**: Interruzione normale.
- **FIQ**: Interruzione rapida.

Ogni eccezione ha un vettore contenente la locazione della prima istruzione per gestirla. Le eccezioni hanno priorità per evitare race-condition.

Gestione delle Eccezioni

Quando si verifica un'eccezione:

1. Copia il `pc` (r15) in `lr` (r14).
2. Copia il `cpsr` in `spsr`.
3. Modifica il `cpsr` per cambiare stato e, se necessario, disabilitare FIQ e IRQ.
4. Carica l'indirizzo del vettore dell'eccezione nel `pc` (r15).

Stack Privato

- Ogni modalità (eccetto User e System) ha il proprio stack privato, gestito tramite `r13` (sp).

Istruzioni Speciali

- Le istruzioni che modificano `cpsr` (come `mrs`, `msr`, `cps`) non sono utilizzabili in User mode perché servono per entrare in modalità privilegiata.

Istruzioni Thumb

- Esistono istruzioni Thumb a 16 bit per sistemi con capacità ridotte.

R13 - Sistemi operativi real-time

Sistema Operativo e Interazione con l'Hardware

Il sistema operativo (OS) permette l'interazione con l'hardware, facendo funzionare i dispositivi e controllando l'uso delle risorse. Questo ambiente fornisce astrazione dall'architettura fisica, offrendo interfacce, multitasking e supporto multi-utente.

Differenze tra Sistemi Operativi

- **Classici (General Purpose)**: Progettati per una vasta gamma di applicazioni e utenti.

- **Real-Time (RTOS)**: Progettati per applicazioni che richiedono predicibilità e affidabilità, spesso utilizzate in sistemi embedded che richiedono compattezza, scalabilità e bassi consumi.

Kernel nei Sistemi Operativi Embedded

- **Microkernel**: Utilizzato nei sistemi embedded real-time. Implementa solo i servizi essenziali (comunicazione di base, schedulazione). La maggior parte delle funzionalità è gestita dalle applicazioni, permettendo verifiche accurate grazie alle dimensioni ridotte.

Caratteristiche di un Sistema Operativo Real-Time

1. **Risposte Predicibili**: Gestione delle interruzioni appropriata.
2. **Risposte Efficienti**: Bassi tempi di risposta.
3. **Gestione Affidabile degli Eventi Temporali**: Garantire che gli eventi temporali siano gestiti correttamente.
4. **Schedulazione Predicibile dei Task**: Utilizzo di algoritmi deterministici per la schedulazione.
5. **Gestione della Comunicazione e Sincronizzazione dei Task**: Utilizzo di code e segnali per la comunicazione tra task.
6. **Gestione della Memoria**: Memoria virtuale e protezione dello spazio di indirizzamento, anche se non sempre necessario per tutte le applicazioni embedded.

Non tutte le funzionalità sono necessarie in ogni applicazione embedded real-time. Ad esempio, la protezione dello spazio di indirizzamento potrebbe non essere sempre necessaria.

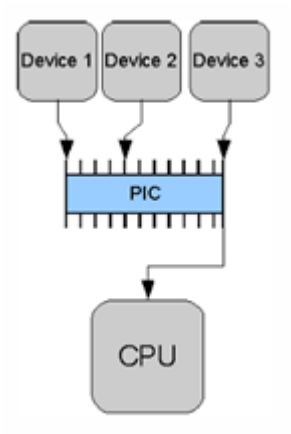
Gestione delle interruzioni

Quando si gestiscono le interruzioni hardware in un sistema operativo, è fondamentale ottimizzare le tempistiche di inizio e fine (latenza), poiché dipendono dall'hardware sottostante.

Il processo di gestione delle interruzioni segue generalmente questi passaggi:

1. **Generazione dell'Interruzione**:
    - Un dispositivo genera un'interruzione hardware e la invia tramite la propria linea fisica al PIC (Programmable Interrupt Controller).
2. **Propagazione alla CPU**:
    - Il PIC propaga l'interruzione alla CPU e aspetta la conferma di ricezione (ack).
3. **Conferma Rapida**:

- La CPU deve confermare velocemente l'interruzione per non bloccare altre interruzioni in arrivo.



Per ottimizzare questo processo e garantire una gestione rapida delle interruzioni, il sistema operativo implementa due fasi distinte:

1. **Interrupt Handler**:
    - Ha priorità elevata e si occupa di confermare la ricezione dell'interruzione al PIC, salvare e ripristinare i contesti di esecuzione del processore.
2. **Interrupt Service Routine (ISR)**:
    - Ha priorità inferiore rispetto all'Interrupt Handler.
    - Esegue le operazioni specifiche per l'interruzione generata, come il processamento dei dati ricevuti dal dispositivo che ha generato l'interruzione.
    - Viene eseguita dopo che l'Interrupt Handler ha completato le operazioni di gestione iniziale dell'interruzione.

Schedulazione

La fase di schedulazione è cruciale poiché determina il prossimo task da eseguire nel modo più efficiente possibile. La semplicità è un obiettivo importante, specialmente considerando che i test di accettazione online per nuovi task non sono supportati.

Quando si cerca di mantenere la semplicità, è comune utilizzare algoritmi a priorità fissa e scheduler clock-driven con diversi livelli di priorità. Tuttavia, un numero eccessivo di livelli di priorità può portare a problemi di schedulabilità. Per risolvere questo problema, è utile mappare le schedulabilità dei task, assegnate dall'algoritmo di schedulazione, alle priorità di sistema, che sono meno numerose.

Questo approccio può causare un problema non menzionato nella teoria: **task con identica priorità**. La soluzione è considerare i task con priorità uguale quando si calcolano i tempi di risposta. Questo perché non è possibile prevedere quale task verrà eseguito successivamente quando le priorità sono identiche.

In foto: T_E task di pari priorità, T_H task di priorità superiore.

$$w_i(t) = e_i + b_i + \sum_{T_k \in T_E(i)} e_k + \sum_{T_k \in T_H(i)} \left\lceil \frac{t}{p_k} \right\rceil e_k$$

eseguono al più una volta sola, perchè hanno stessa priorità

$$w_{i,j}(t) = j \cdot e_i + b_i + \sum_{T_k \in T_E(i)} \left(\left\lceil \frac{(j-1)p_i}{p_k} \right\rceil + 1 \right) e_k + \sum_{T_k \in T_H(i)} \left\lceil \frac{t}{p_k} \right\rceil e_k$$

stiamo contando i "j-1" periodi precedenti dei task Tk, +1 perchè consideriamo l'esecuzione nel periodo attuale.

Come realizzo questa associazione? Linearmente? No, vorrei essere "più preciso" quando la priorità è alta, e quando la priorità è più bassa essere meno preciso.

In pratica si può cercare di mantenere approssimativamente costanti i rapporti

$$g_k = \frac{\pi_{k-1} + 1}{\pi_k} \quad (1 < k \leq \Omega_s)$$

Nell'esempio precedente considerando $\pi_1 = 1$, $\pi_2 = 4$, $\pi_3 = 10$ si ottengono rapporti uguali a 1/2

Quindi 1 è mappato su π_1; 2, 3 e 4 su π_2; da 5 a 10 su π_3

Ciò impatta anche su alcuni teoremi:

Teorema (Lechoczky & Sha, 1986) implicite

Per l'algoritmo di scheduling RM, con scadenze relative pari al periodo e numero n di task elevato, usando l'associazione a rapporto costante con $g = \min_{1<k\leq\Omega_s} g_k$ allora:

$$U_{RM}(g) = \begin{cases} \ln(2g) + 1 - g & \text{se } g > 1/2 \\ g & \text{se } g \leq 1/2 \end{cases}$$

La *schedulabilità relativa* indica la perdita di schedulabilità dovuta ad un numero limitato di livelli di priorità nel sistema:

$$U_{RM}(g)/\ln 2$$ In 2 deriva da Liu-Layland, utilizzazione max.

Più "g" è grande, meglio mappiamo le priorità del sistema sulle priorità dei task.

Casi limite:

$g = 1$: $U_{RM}(g)/\ln 2 = 1$ $\Rightarrow$ nessuna perdita

$g = 1/2$: $U_{RM}(g)/\ln 2 = 1/(2\ln 2)$ $\Rightarrow$ perdita del 28%

I sistemi operativi Real-Time Operating Systems (RTOS) offrono l'impostazione della deadline relativa, EDF è poco supportato, in quanto l'ordinamento basato su deadline assoluta è poco efficiente (si preferisce usare strutture semplici). Ovviamente, si cerca di **standardizzare** i RTOS, anche per avere portabilità e interoperabilità. Vediamone alcuni standard:

1. **POSIX**:
    - Fornisce un'ampia gamma di funzionalità, inclusa la misurazione dei tempi, la gestione della memoria condivisa, la sincronizzazione e la mutua esclusione tramite priority inheritance.
    - Offre quattro profili per adattarsi a diversi tipi di sistemi: piccoli, robotici, avionici e real-time/general purpose.
2. **OSEK**:
    - Si concentra sulla scalabilità, ma non offre protezione della memoria.
    - Non fornisce interfacce per i sistemi di I/O e utilizza strutture allocate staticamente.
3. **ARINC 653**:
    - Utilizzato nell'ambito avionico, divide la memoria in partizioni in cui operano sottosistemi diversi.
    - Le partizioni non interferiscono tra loro ma possono scambiarsi messaggi tramite code o sovrascrivere dati.
4. **TRON**:
    - Caratterizzato da una standardizzazione meno rigida, fornisce API a livello di sorgente senza requisiti di compatibilità.
    - Non offre protezione della memoria e ha una standardizzazione meno rigida rispetto ad altri sistemi.

Nonostante le differenze nei dettagli, i vari RTOS condividono microkernel ridotti con basso overhead e utilizzano lo stesso sistema di gestione delle interruzioni. Tuttavia, nessuno di questi sistemi implementa la paginazione.

Task di priorità uguale schedulabili con FIFO o Round Robin, priorità cambiabili a runtime, EDF o server a conservazione poco diffusi, spesso includono priority-inheritance (disabilitabili). I task possono comunicare e sincronizzarsi tramite code di messaggi, pipe, FIFO, segnali…

Gestione della memoria

In ambito real-time si cerca di non penalizzare i tempi di esecuzione, per cui c'è poco spazio per memoria virtuale (I dati hanno size fissa ridotta, come l'overhead, i blocchi di memoria allocati sono contigui) o protezione dello spazio (un unico spazio comporta più semplicità e quindi predicibilità).

1. **VxWorks**:
    - Utilizza l'ambiente Eclipse e fornisce API proprietarie.
2. **LynxOS**:
    - Compatibile con l'Application Binary Interface di Linux, utilizza un kernel monolitico.
3. **Neutrino**:
    - Basato su microkernel, supporta interruzioni annidate ed è conforme a POSIX.
4. **eCos**:
    - È modulare e permette miglioramenti grazie alla sua licenza. È scritto in C++ e offre supporto per interruzioni, schedulazione e thread.
5. **FreeRTOS**:
    - È diffuso, compatto e semplice, scritto in C. Appare come una libreria.
6. **Windows Embedded**:
    - Esiste una versione Compact, anche se non è così compatta come ci si potrebbe aspettare. Utilizza un kernel monolitico, è progettato per sistemi monoprocessore e non supporta POSIX.
7. **Zephyr**:
    - Combina kernel, driver e librerie. Utilizza uno spazio di indirizzamento singolo e offre protezione della memoria. Utilizza uno scheduler a priorità fissa.

R14 - Linux in ambito real-time

Linux non è real time, ma ha alcune caratteristiche interessanti (gestione del tempo, gestione separata delle interruzioni, schedulazione con priorità, interrompibilità anche in kernel mode, gestione risorse condivise, configurazione a grana fine, facilità nel creare task kernel mode (per avere unico spazio di indirizzamento), supporto alla memoria virtuale in user mode, kernel monolitico ma ridotto). **Ciò che manca è la predicibilità**, in quanto il kernel non è

completamente interrompibile, le ISR possono avere durata non predicibile, la latenza nella schedulazione è elevata. Soluzione? **Modificare il kernel Linux.** Ma ha senso essendo monolitico? Si, in quanto molto configurabile. Esaminiamo i vari punti da correggere:

Gestione interruzioni

Le interruzioni sono un aspetto cruciale dei sistemi real-time e la loro gestione avviene a tre livelli:

- **Top Half**
    1. L'Interrupt Handler si occupa di salvare e ripristinare il contesto, confermare la ricezione dell'interruzione e gestire le interazioni iniziali con l'hardware.
    2. La parte immediata dell'ISR interagisce con la periferica H/W ed eventualmente "prenota" la successiva esecuzione del bottom half.
- **Bottom Half**
    3. Svolge operazioni lunghe generalmente interrompibili. È costituita da una procedura da eseguire quando nessun altro top half è in esecuzione, quindi:
        - immediatamente prima di tornare ad eseguire un processo.
        - oppure per mezzo di un kernel thread.

Le interruzioni, a loro volta, possono essere interrotte e non hanno una gerarchia di priorità tra di loro. Questo significa che non è una buona pratica disabilitare l'interrompibilità delle interruzioni, poiché potrebbe causare blocchi nel sistema.

Ad esempio, potrebbe verificarsi la situazione in cui un'ISR associata a un task di priorità media sta eseguendo, ma nel frattempo arriva un'interruzione associata a un task di priorità superiore. Anche se il task di priorità superiore dovrebbe teoricamente avere la precedenza, poiché le interruzioni non hanno una gerarchia di priorità, il task di priorità superiore potrebbe essere costretto ad attendere finché l'ISR di priorità media non completa la sua esecuzione.

Scheduler Linux

Esso è basato su **classi di schedulazione** (diverse politiche per la gestione dei processi), ovvero lo scheduler chiede classe per classe se c'è qualche processo da eseguire. Ciò andrebbe bene in ambito real-time se tali classi avessero priorità massima.

Nel caso base, prendiamo processi interrompibili a priorità fissa:

- La schedulazione real time è molto differente rispetto a non real time.

- A parità di priorità, si usa FIFO o Round Robin.
- Lo scheduler ha singola coda di processi con 100 livelli di priorità per ogni processore, consentendo quindi di implementare RM o DM.

Esiste meccanismo che evita ad un processo RM di non rilasciare mai la CPU, bloccando il sistema.

Esiste la classe "**Deadline**" utile per usare EDF. Presenta i parametri *tempo esecuzione*, *periodo*, *deadline relativa*, e le *scadenze assolute* si calcolano con **CBS**. Ogni task della classe SCHED_DEADLINE è analogo a un server CBS!

Non era meglio EDF puro? Il fatto è che i task possono avere comportamenti scorretti; quindi, ciò che si fa con tale implementazione è fare in modo che il task "prenoti" una porzione di esecuzione dello scheduler, il quale, quando chiamato, aggiorna scadenze ed aggiorna in base a EDF. Quando un task termina il budget, viene sospeso fino alla deadline usata. Quindi è un misto tra EDF e CBS, non è "puramente" nessuno dei due.

Esistono però alcuni problemi:

- Un processo real time di priorità alta può comunque essere interrotto e rallentato.
- Esistono meccanismi di **load balancing** sui sistemi multiprocessore, ovvero realizza la migrazione dei processi senza considerare la priorità, aumentando la latenza e diminuendo la predicibilità. Si può risolvere? Si, perché tale problema si ha in sistemi multiprocessore e con schedulazione globale dei task (in quanto c'è migrazione), allora basta definire l'affinità tramite bitmap, forzando un sistema statico.

Modifica del Kernel

Abbiamo detto che è necessario attuare delle modifiche, esistono due strade:

- **Approccio mono-kernel**: modifica radicale, tolgo tutto ciò che causa impredicibilità. Il kernel diventa quasi interrompibile, applico priority-inheritance. I suoi vantaggi risiedono nel fatto che le applicazioni real time sono normali applicazioni, l'applicazione real-time può usare stack e comunicazioni implementate da Linux, l'ambiente di sviluppo è flessibile, portando il kernel standard a continui miglioramenti.
- **Approccio dual-kernel**: poche modifiche, c'è strato software intermedio tra kernel e dispositivi hardware, per impedirne l'accesso diretto. Non punto a migliorare la predicibilità di Linux, bensì miglioro la predicibilità di un RTOS in esecuzione insieme a Linux, mediante lo strato intermedio appena citato. Nella pratica, ho due sistemi operativi con priorità diverse su stesso hardware, RTOS ha priorità maggiore ed esegue processi RT, Linux è attivo

solo quando non ci sono. Lo strato intermedio intercetta e consegna le interruzioni a Linux quando è in esecuzione; quindi, tale strato sostituisce sempre la gestione iniziale delle interruzioni di Linux. RTOS è attivabile dopo aver inizializzato Linux. I vantaggi risiedono nell'indipendenza dal kernel Linux, nella possibilità di usare un RTOS specifico, devo certificare solo lo strato intermedio, e non necessito di modificare radicalmente Linux. Un fattore da non sottovalutare nelle certificazioni è **il partizionamento spaziale di un sistema**, che richiede una precisa condivisione di risorse tra SO o applicazioni evitando interferenze di funzionamento tra di loro (Ad esempio partiziono la RAM in due zone diverse per due programmi diversi). È realizzato tramite il programma **separation kernel**.

Anche il **partizionamento temporale** è richiesto nelle certificazioni, partizionando il tempo, in modo da non avere interferenze sui tempi di esecuzione. Realizzato tramite schedulazione ciclica delle partizioni.

Esempi dual kernel

RTAI è un sistema operativo real-time che combina un separation kernel con un modulo Linux che funge da kernel RTOS. Offre schedulazione sia clock-driven sia event-driven, attivata da interruzioni di timer, uscita da ISR o sospensione spontanea del task. È in grado di schedulare anche processi Linux, ma senza utilizzare le syscall di Linux. Supporta anche "task nativi", schedulati solo da RTAI, con un context switch veloce e senza protezione sulla memoria, ma poco integrabili con Linux. Su sistemi multiprocessore, è possibile forzare i task su una CPU specifica. RTAI supporta politiche di scheduling come FIFO, Round Robin, RM, EDF e priority inheritance parzialmente.

ADEOS è un nano-kernel con un approccio di virtualizzazione che permette a diversi sistemi operativi di avere il proprio dominio. ADEOS intercetta le interruzioni e le smista ai vari sistemi operativi. Ogni dominio ha la propria catena di consegna delle interruzioni e può decidere se accettare o meno un'interruzione, incluso disabilitarla.

Esempi mono kernel

La patch **PREEMPT_RT** riduce le sezioni non interrompibili del kernel e permette l'esecuzione dei gestori delle interruzioni in un contesto schedulabile. Questo è ottenuto introducendo gli *rt-mutex*, in grado di sospendere l'esecuzione del processo corrente (senza attendere cambi di stato come con le primitive del kernel classiche).

Interruzioni brevi sono inalterate, interruzioni meno brevi come *spinlock* diventano interrompibili. Si cerca anche di non annullare il bilanciamento del carico, realizzandolo prima o dopo l'esecuzione (non durante), distribuendo i processi a priorità elevata.