

27/09/2022

INTRODUZIONE AI SISTEMI EMBEDDED

Cos'è un sistema embedded?

- È un sistema programmabile che non è pensato per essere programmato da chiunque: deve poter fare una cosa sola (e BEN precisa).
- È un sistema integrato, che molto spesso non ha un'interfaccia standard (non ha tastiera o mouse).
- È un sistema pervasivo (ad altissima diffusione).

Oltre il 99% dei dispositivi programmabili sono sistemi embedded!

→ Un'infrastruttura costituita da sistemi interconnessi tra loro e a Internet costituisce una INTERNET OF THINGS.

È difficile definire un'architettura di riferimento (standard) per i sistemi embedded; però col tempo ci siamo accorti.

Le 3 capacità di base di un sistema embedded sono: ELABORAZIONE, COMUNICAZIONE, MEMORIZZAZIONE.

Invece, la dependability di un sistema embedded si misura in termini di affidabilità, di manutenibilità, di qualità, di disponibilità, di safety e di sicurezza (security).

→ TEMPO DI RISPOSTA DI UN JOB = istante di completamento - istante di rilascio,
dove l'istante di rilascio è l'istante in cui il job diventa disponibile.

→ SCADENZA ASSOLUTA = ~~ISTANTE DI RILASCI~~ istante di rilascio + scadenza relativa.
È un istante di tempo.

È la ~~data~~ ^{data} max dell'esecuzione di un job
(è un tempo di risposta max).

04/10/2022

Sviluppo dei sistemi embedded:

Per realizzare un sistema embedded è necessario considerare tre aspetti fondamentali: ARCHITETTURA, APPLICAZIONI e METODOLOGIE

Dalla base dello sviluppo c'è sempre la necessità di realizzare una determinata applicazione, con relativi REQUISITI FUNZIONALI e NON FUNZIONALI.

- CAPACITÀ DI ELABORAZIONE
- PRESTAZIONI
- CONSUMO DI ENERGIA
- AFFIDABILITÀ, ROBUSTEZZA
- SAFETY, SECURITY
- COSTI DI PROGETTO (FABBRICAZIONE / MANUTENZIONE)
- SCOPRI DI PROGETTAZIONE
- # PEZZI DA FABBRICARE

→ Definire una buona metodologia di sviluppo è essenziale per avere un ~~lavoro~~ ^{lavoro} ~~presto~~ ^{presto} ripetibile e veloce, e per avere prevedibilità nei costi di progettazione, fabbricazione e manutenzione. Serve anche per far sì che team diversi possano lavorare per lo stesso sistema embedded.

Tuttavia, il processo di sviluppo coinvolge molte fasi, alcune delle quali non dispengono di alcuna strumento di sintesi, quindi è necessario operare diverse analisi e simulazioni.

Inoltre, per lo sviluppo di hw e lo ~~sviluppo~~ sviluppo di sw si utilizzano metodologie differenti.

E per i sistemi embedded come faciamo? Un mix delle due.

→ Conosciamo bene i modelli di sviluppo del sw (a cascata, agile, ...).

→ Per quanto riguarda i modelli di sviluppo dell'hardware, presentano tecniche sofisticate sembra essere a chi sviluppa software.

In particolare, per lo sviluppo dell'hw si utilizza una specifica RTL (REGISTER-TRANSFER). Per ciò c'è un ingegnere che sviluppa tale specifica. Dopo di che è tutto automatico: si crea un modello astratto da cui si effettua una sintesi logica indipendente dalla tecnologia (in cui sono dati dei circuiti elementari, ^{delle} porte logiche, ecc.). Segue poi una sintesi logica dipendente dalla tecnologia.

L'ultima fase, la più complicata, consiste nella realizzazione vera e propria del circuito elettronico; anche qui il lavoro è probabilmente automatizzato.

Vediamo più in dettaglio come deve essere realizzato un sistema embedded.

- 1) Stabilire quali funzionalità realizzare direttamente nell'hardware e quali funzionalità realizzare nel software*. È UN PASSAGGIO CRITICO E FONDAMENTALE (se si sbaglia qui, c'è).
- 2) Definire l'architettura generale del sistema embedded (e.g. ~~come~~ come inserisce il microcontrollore e come fa a interfacciare l'hw col sw).
- 3) Determinare le specifiche dell'hw e del sw.
- 4) Realizzare le componenti hw e sw contemporaneamente e separatamente.

* TIPICAMENTE, INSERIRE UNA FUNZIONALITÀ IN HW AUMENTA IL CONSUMO E INSERIRE UNA FUNZIONALITÀ IN SW DICE SULLE PRESTAZIONI.

- 5) Effettuare l'integrazione e il debugging.

Spesso i sistemi embedded vengono sviluppati definendo^{e poi} utilizzando una PIATTAFORMA di riferimento.

→ PIATTAFORMA = dispositivo/architettura di base che può essere modificata per realizzare nuove funzionalità o modificare quelle esistenti.

→ L'azienda si occupa solo dell'utilizzo della piattaforma scelta: lo sviluppo viene effettuato in precedenza da terze parti.

La base dei sistemi embedded è la TECNOLOGIA MATERIALE (→ utilizzo di circuiti molto piccoli) → GRANZO VANTAGGIO: aumento delle prestazioni.

Ricordiamo che un fotone, in un nanosecondo, percorre 25 cm circa ⇒ per avere una frequenza di 1 GHz, non si possono avere circuiti più grandi dei 25 cm.

Nella pratica, il progettista ha ampia possibilità di scelta fra:

- Componenti cors (Commercial, Off-The-Shelf). → Sono componenti commerciali che si possono trovare già prati.
- Microprocessori. → Sono macchine a stati fissi che eseguono istruzioni leggendo un'apposita area di memoria; sono fissate tra meno parti; comunque sia, è conveniente usare quanto più diverse realizzazioni funzionalità complesse in un sistema embedded.
- Logiche programmabili. → Sono chip che integrano risorse logiche e have di interconnessione completamente fabbricate. Tutto sia per programmazione tali chip, che possono essere caricate (PROGRAMMATE), è RIFIGRABILI e ricopribili.
- ASIC. → È un circuito integrato fatto per compiere una specifica cosa. NB: progettare un ASIC da zero costa anche miliardi di euro. Dov'è la piattaforma di riferimento priva di questa problematica.

del circuito

Tecnologie per lo sviluppo di ASIC: STANDARD CELL - GATE ARRAY.

Tipologie di PLD: PLA - PAL - GAL - CPLD - FPGA.

Tipologie di microprocessori: COTS - IP SOFT-MACRO — IP FIRM-MACRO — IP HARD-MACRO

funzionalità
agiva qui, ciao)

Non tutti i processori sono general purpose, ad esempio, i DSP (Digital Signal Processor) sono ottimizzati per l'elaborazione numerica, alla base di cui c'è l'operazione $Z_{t+1} = Z_t \cdot n + y$.

microcontrol
= Dopo di che ci sono i NP (Network processor), che vengono usati negli apparati di rete e di comunicazione.

I microcontrollori, invece, sono microprocessori che dispongono di molte periferiche e interface in un singolo chip.

11/10/2022

Sistema bare-metal:

Il nostro scopo è definire un sistema SW (che è un ambiente di esecuzione ma non un sistema operativo) real-time "BARE-METAL". Tale ambiente di esecuzione è minimale e auto-contenuto (\rightarrow durante la fase di bootstrap ~~non~~ non possiamo assumere che esiste nulla, né funzioni di libreria, né funzioni di sistema, perché appunto il SO non esiste).
L'ambiente di esecuzione dovrà poi essere volichetto come HARD REAL-TIME.

Più su cui andremo a eseguire il nostro task è l'SBC (Single Board Computer), che si programma in modo simile ai nostri computer (ma ricordiamoci sempre dell'asincronia).

• Jm' alto

- A dx c'

• Jm' bass

Pulsanti:

\rightarrow BOOT

\rightarrow RESET

• (altre,

Jufine,

ca agli

del so). ~~SSS~~ con esame ora

SBC ha:

IL TUTTO IN 40
GRAMMI DI PESO.

→ Una CPU ARM Cortex - A8 di 1 GHz.

→ 512 MB di memoria SDRAM (che non è pochissimo per un sistema embedded).

→ 4 GB di storage on-board flash.

→ Acceleratore grafico 3D, acceleratori di virgola mobile VFP e NEON.

→ 2 PRU (è un core ARM semplificato).

→ 2 porte USB (1 client e 1 host). → CLIENT USB: porta in alto a sinistra; SERVER USB: porta in basso a destra.

→ 1 porta Ethernet, 1 uscita video, 1 porta seriale.

→ Questa schedina ora è utilizzata in molti sistemi embedded; fra l'altro, il progetto della schedina è aperto (chiunque può leggerlo/modificarlo). Però è possibile personalizzare la schedina per adattarla perfettamente al sistema specifico che si vuole sviluppare.

Analizziamo la schedina:

- Al centro c'è il SYSTEM core integrato.

- In alto a dx c'è un circuito integrato.

- A sx c'è l'interfaccia verso la porta Ethernet (ETHERNET PHY).

- In basso a dx c'è un circuito integrato che distribuisce l'alimentazione alle altre componenti della scheda.

- In alto a sx c'è la memoria flash.

- A dx c'è la memoria SDRAM.

- In basso ci sono 6 pin per la connessione con le porte seriali.

Pulsanti:

→ BOOT in alto a dx.

→ RESET e POWER in alto a sx.

Inoltre, in alto a sx ci sono 4 led blu.

Infine, in alto e in basso vi sono due ~~porte~~ di espansione per collegare la rete all'esterno agli altri dispositivi.

Non ci sono ventole: l'ARM non scalda molto, a differenza dell'Intel; funziona con basse energie.

Noi NON rifaremo da zero la procedura di BOOTSTRAP: cerchiamo di capire cos'è.

È l'inizializzazione ed è costituita da 2 fasi:

1) Esecuzione di un programma in una ROM nel chip AM335x +

• caricamento di un programma dalla flash eMMC oppure da una scheda microSD esterna.

Il programma nella flash eMMC è per default uBoot, che ha un codice open-source.
uBoot si dispone a caricare quello che pensa essere il sistema bare-metal.

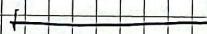
2) La seconda fase del bootstrap dipende da uBoot, che può:

→ ricevere comandi dall'esterno.

→ eseguire una seq. di comandi che carica l'ambiente di esecuzione.

Nel sistema embedded non si ha un BIOS complicato: piuttosto si definisce storicamente un elenco di periferiche presenti nella scheda.

↳ In Linux tale meccanismo è detto DEVICE TREE.



Il processore che andremo a utilizzare è il Beaglebone Black, che è una CPU ARM con:

→ architettura RISC a 32 bit.

→ Cache integrata di livelli 1 e 2.

→ Memory Management Unit (MMU).

→ Translation Lookaside Buffers (TLB).

→ Branch target address cache (→ dove si predice il target delle istruzioni di salto condizionale o indiretto). ↳ VEDI SOA

→ Coprocessori VFP (floating point) e NEON.

Ambiente di esecuzione SERT (System Environment for Real-Time applications):

È un ambiente di esecuzione più funzionale per Technologic Systems, Raspberry PI,

Beaglebone Black: È un sistema che si interfaccia direttamente all'hardware (= BARE-METAL), è compatto ed è real-time (prevedibile, verificabile e validabili). È scritto in C e in Assembler (si interfaccia con l'hw: per forza di cose bisogna usare linguaggi a basso livello).

In realtà noi programmiamo su una macchina general-purpose e non direttamente sull'hw \Rightarrow NECESSITA DI INTRODURRE UN ~~ENTITÀ~~ AMBIENTE DI SVILUPPO IN GRADO DI GENERARE CODICE MACCHINA PER I SISTEMI EMBEDDED (avendo un cross-compiler).

N.B.: La schema embedded non ha uno schermo o una tastiera (non c'è proprio nulla): l'interfaccia di comunicazione più semplice che possiamo usare sono i LED. Dopotutto, si usa anche la PORTA SERIALE.

Dettagli sull'architettura ARM: \rightarrow NON VEDREMO TUTTE QUESTE SLIDE A LEZIONE

\rightarrow Tutti i cellulari (Android, iOS,...) utilizzano l'architettura ARM.

\rightarrow ARM = Acorn Risc Machine (in origine) ~~può essere~~ Advanced Risc Machine.

\rightarrow ARM oggi DOMINA il mercato dei processori (non Intel o altri...) \rightarrow basti pensare che ha superato i 20 mld di chip prodotti.

\rightarrow tra l'altro è in RAPIDISSIMA evoluzione. Esiste una grande famiglia di famiglie di ARM diverse oggi.

\rightarrow FAMIGLIA = organizzazione interna del processore

Ad esempio la FAMIGLIA CORTEX ha 3 profili applicativi:

- 1) Cortex-A (profilo Application). \rightarrow PER SISTEMI DI USO GENERALE
- 2) Cortex-R (profilo Real-time). \rightarrow PER SISTEMI REAL-TIME
- 3) Cortex-M (profilo Microcontroller). \rightarrow PER SISTEMI EMBEDDED

CARATTERISTICHE PRINCIPALI DI ARM:

- Architettura a 32 bit (nelle architetture moderne si ha la convivenza fra instruction set a 32 bit e instruction set a 64 bit, come in Intel).
- Architettura Risc (\equiv lunghezza fissa a 32 bit di tutte le istruzioni macchina) \rightarrow maggiore efficienza nella decodifica.
- Il processore può lavorare sia in LITTLE-ENDIAN che in BIG-ENDIAN. \rightarrow fa lavorare in LITTLE-ENDIAN

- Nel caso di ARM a 32 bit, quasi tutte le istruzioni sono CONDIZIONALI (cioè sono dei flag che determinano se ogni istruzione deve essere eseguita o meno).

- Non ci sono le istruzioni di SHIFT e, talvolta, di DIVISIONE, ma sono supportate ISTRUZIONI DI MOLTIPLICAZIONE.

- Si hanno degli schemi di indirizzamento tipici di un'architettura CISC.

- Ci sono 31 registri a 32 bit: $R_0, R_1, R_2, \dots, R_{30}$, di cui:

- R_{10} : ~~REGISTRO DI DIMENSIONE STACK~~

- R_{11} : FRAME PINTER

- R_{12} : ~~REGISTER PER L'INVOCAZIONE DELLE PROCEDURE~~

- R_{13} : STACK PINTER

- R_{14} : LINK REGISTER

- R_{15} : PROGRAM COUNTER: punta alla seconda istruzione sotto quella corrente (ergo 8 byte dopo all'istruzione corrente).

} SI POSSONO ANCHE USARE

} QUESTI REGISTRI GENERAL PURPOSE

Dunque si hanno i registri di stato, di cui CPSR, che ha bit di condizione e di controllo.

Però abbiamo 5 registri spz.

- Per quanto riguarda l'architettura ARM a 32 bit, è più tradizionale: ad esempio, non ci sono più le istruzioni condizionali (eccetto i classici salti condizionali).

→ diversi modi di esecuzione nel processore ARM:

→ MODO USER

→ MODO SYSTEM

→ MODO FIQ (≡ interrupt veloci) → NON SI POSSONO ABILITARE NEL SOC.

→ MODO IRQ (≡ interrupt normali)

→ MODO SUPERVISOR (≡ Kernel mode)

→ MODO ABORT

→ MODO UNDEFINED (quando si ha un'istruzione non definita che bisogna gestire in qualche modo)

13/10/2022

Algoritmi per la schedulazione real-time:

Sono divisi in 3 categorie:

1) CLOCK-DRIVEN

2) WEIGHTED ROUND-ROBIN

3) PRIORITY DRIVEN

scheltrazione di un job operativo e quindi nel caso in cui non ci sia modo che rispetti le scadenze.

→ Ad esempio possono essere implementati dei test di accettazione.

→ Un modo efficiente per schedellare i job operativi hard real-time è ordinare in base alla loro scadenza. In particolare, si hanno due case:

1) Job rilasciati e non ancora accettati.

2) Job accettati.

L'algoritmo che si basa sulla scadenza + richiesta è detto EDF.

18/10/2022

④ Programmazione bare-metal di un sistema embedded:

Occorre: CAVO USB ; CONVERTITORE SERIALE-USB.

Serve anche un compilatore, in particolare un CROSS-COMPILER → è possibile definirsi una variabile d'ambiente (PREFIX) che punta alla directory dove si trova il nostro cross-compiler.

→ gcc - bilancio - 7.6.1 - 2019 02 è perfetto.

DIRIUSCIRE CHE COMPILA IL CODICE

MA

NUOVA TUTTA ESEGIBILE

NUOVA CHE NON È IN MARE

NUOVA LA CUI CPU NELLA

→ Per compilare: \${PREFIX}gcc -Wall -Wextra -Os -fno-exceptions -mcpu=cortex-a8

SPECIFICA L'ARCHITETTURA CHE ABBIANO

SPECIFICA L'IMPLEMENTAZIONE

INTERFACCE

SPECIFICA I'AFLONTI

POINT UNIT

SPECIFICA IL FPMV3

-march = armv7-a

-mfloat=abi=hard

-mfpu=vfpv3

-march = C name_progr.C

→ Per caricare: \${PREFIX}ld -nostdlib -e reset -O sort.elf name_progr.o

è la funzione che abbiamo scritto

L'ENTRY POINT

SEI ESEGIBILE

FILE ESEGIBILE

LINUX THE VERBIS PROTAG

→ Per estrarre l'esegibile: \${PREFIX}objcopy -S -O binary sort.elf sort.bin

→ Per disassemblare: \${PREFIX}objdump -d sort.elf

⑤ Bisogna dire a Linux che gli stiamo mandando il nostro programma da eseguire.

→ CORTANDO loadaddr (INDRIZZO INIZIALE) → tipicamente 0x2000000

→ Lo facciamo con un particolare protocollo (e.g. KERMIT).

Dopo che bisogna dire a Linux che deve saltare all'indirizzo dell'applicazione.

↳ COMANDO go INSERITO

↳ Ci accorgiamo che il programma sia andato correttamente tramite i LED.

Per evitare tutta questa fatica ogni volta, si ricorre al MAKEFILE.

NB: È necessario anche un file sect.lds per stabilire il come va posizionato correttamente il codice. Esempio:

ENTRY(_reset)

```
mem_start = 0x80000000;  
mem_length=512*1024*1024; // 512 MiB  
mem_end = mem_start + mem_length;
```

MEMORY {

```
ram(rwx) : ORIGIN = mem_start, LENGTH = mem_length
```

}

SECTIONS {

.text : {

```
*(.text) // → si può aggiungere una riga sotto per allineare di 4 byte: . = ALIGN(4);
```

} > ram // tutte le sezioni di testo vanno messe insieme in ram

.data : {

```
*(.data)
```

```
. = ALIGN(4);
```

} > ram

.bss : {

~~```
*(.bss) bss_start = .;
```~~~~```
*(.bss)
```~~

```
*(.COMMON)
```

```
. = ALIGN(4);
```

```
- bss_end = .;
```

} > ram

}

~~INDIRIZZO~~ MEMORY-MAPPED = indirizzo logico che non punta alla RAM, bensì a un registro.

Vediamo di far funzionare i LED:

```
void _reset(void) {
```

to correttamente

```
    int *gpio1 = (int *) 0x4804C000;
```

```
    gpio1[0x194/4] |= (1<<21) | (1<<22) | (1<<23) | (1<<24);
```

```
    for(;;)
```

SETTAGGIO A 1

```
}
```

↑ | LED così NON SI ACCENDONO: bisogna configurarli come output.

Prima di $\text{gpio1}[0x194/4] |= \dots$ bisogna aggiungere la seguente istruzione:

```
    gpio1[0x134/4] &= ~(1<<21) | (1<<22) | (1<<23) | (1<<24);
```

SETTAGGIO A 0

25/10/2022

Tornando al nostro programma scritto la settimana scorsa, è un puro caso che funzioni: per programmi più complessi (composti da più di un'istruzione...) è necessaria della configurazione ulteriore.

→ ACCESSTORE DI UN CIRCUITO = invio di un segnale di clock.

↳ Per il GPIO10, il modulemode vale 0 se vogliamo disattivare il circuito, 2 altrimenti.

D'altra parte, il glock vale 0 se vogliamo disattivare il circuito, 1 altrimenti.

↳ Il modulemode si trova nella zona bassa di un apposito registro mentre il glock si trova nella zona medio-alta del registro.

In definitiva, il registro vale 0000 0000 0000 0000 0000 0000 0010.

Nel nostro codice bisognerà aggiungere:

int *cm_per = (int *) 0x48040000;

cm_per[0x4c/4] = 0x00000002;

Ora facciamo lampeggiare i led:

for (i;) {

for (int c=0; c<100000000; c++) {

; }

gpio1[0x130/4] = 0xf << 21;

gpio1[0x134/4] = state << 21;

state = (state + 1) & 0xf;

}

→ Ovviamente non funziona 'na matita.

Il problema qui è che gcc è un compilatore ottimizzante; ma noi non vogliamo uscire per il flag -O dalla compilazione?

Per essere più precisi, gcc ha ritenuto inutile tutto il corpo del ciclo for esterno.

Come facciamo a risolvere il problema?

1) Aggiungere __asm volatile (""); __ell nel ciclo più interno.

2) Dichiare i due vettori cm_per e gpio1 come volatile.

Stiamo usando Blasile per dire al compilatore di non effettuare determinate ottimizzazioni perché le operazioni di interasse sono mappe sul hardware.

Ora il programma funziona, ma risulta, non possiamo buttare tutto dentro a main!

Scriviamo un file header (e.g. beagleboneblack.h):

```
#ifndef BEAGLEBONEBLACK_H  
#define BEAGLEBONEBLACK_H  
#include "bbb_cpu.h"  
#include "comm.h"  
#endif
```

Ora scriviamo in bbb-cpu.h / comm.h :

```
#ifndef BEAGLEBONEBLACK_H  
#error you must not include this directly  
#endif  
def compiler_barrier()  
-asm- volatile ("")
```

Così è possibile importare BEAGLEBONEBLACK_H nel nostro programma e scrivere compiler_barrier() al posto di asm volatile("")

Ora che non funziona più 'na sega un'altra volta!

Così facendo, nell'assembly, <reset> si trova a un indirizzo diverso da 0x8000000, perché il compilatore ha rendarlo tutto come volatile.

Per questo motivo, bisogna dire al compilatore che un determinato file va posta prima degli altri nella compilazione. Scriviamo un file ASSEMBLER startup.o:

```
.text  
.code 32 → codice ARM a 32 bit  
.global reset → guarda, esiste un simbolo globale di nome reset  
reset:  
b main → qui è necessaria rimuovere la nostra funzione nel programma Main
```

E continua a non funzionare... PERCHÉ?!?

Mancava lo stack, che è fondamentale quando si ha più di una funzione.

Torniamo al file `sert.lis` e aggiungiamo:

```
stack : {  
    . = ALIGN(4);  
    stack_end = .;  
    . = . + 4096; → lascia spazio per una pagina  
    stack_stop = .;  
} > ram  
S
```

Andiamo ora a `startup.c` e inizializziamo opportunamente lo stack punto a. Sopra al main introduciamo:

```
ldr sp, =stack_top
```

Inoltre, dopo code 32 aggiungiamo:

```
#define SYS_MODE 0x1f
```

e, subito prima di `ldr sp, =stack_top`:

Stiamo fornendo l'esecuzione all'interno della modalità privilegiata (in bare-metal è banale).

```
cpsid if, #SYS_MODE
```

BARRIERE AL LIVELLO HARDWARE:

Possono essere di sincronizzazione, a livello istruzione o a livello di memoria.

Aggiungiamo in `bbs.cpl.h`:

```
#define __asm data_memory_barrier() \ → così ANCHE PER data_sync  
    __asm volatile __ ("dmb sy" :: "memory")
```

Aggiungiamo in `comm.h`:

```
static inline  
void loop_delay()  
{  
    while (d-- > 0)  
        data_sync_barrier();  
}
```

Al posto del braccio {
 bisogna mettere
 il braccio in startup
 b.init();

INIZIALIZZA
IL GPIO.

Definiamo da un file `int.c` con una funzione che invochi `main()`, e `int gpio()`, e `fill_bss()`.
INIZIALIZZA IL BSS A 0.

Sempre in init.c:

```
void fill_bss(void) {  
    extern U32 __bss_start, __bss_end; // typedef unsigned int U32; in beagleboneblack.h  
    U32 *p;  
    for (p = &__bss_start; p < &__bss_end; ++p)  
        *p = 0UL;  
}
```

```
void init_gpio1(void) {
```

```
}
```

Problema che abbiamo ora: abbiamo bisogno di un modo per definire l'accesso all'hwr.

in beagleboneblack.h, dopo il typedef:

```
volatile U32 *const gpio1 = (U32 *) 0x48040000; // Soluzione che funziona ma non è forte:  
// fa un accesso in RAM e un accesso al  
// registro; vogliamo evitare l'accesso  
// in RAM  
#define GPIO1_DATAOUT (0x13C/4)  
gpio1[GPIO1_DATAOUT] = 0xf;
```

Subito dopo typedef aggiungiamo (al posto dell'elenco definito prima):

```
static volatile U32 *const iomem = (U32 *) 0;  
// define iomemdef(N,V) enum {N=(V)/sizeof(U32)};  
// define iomem(N) iomem[N]
```

```
#define GPIO1_BASE 0x48040000
```

```
iomemdef(GPIO1_OE, GPIO1_BASE + 0x134); // Si possono definire in modo analogo le macro  
iomem(GPIO1_OE); relative a vari registri
```

Ora possiamo anche definire delle funzioni di servizio:

```
static inline void iomem_high (unsigned int reg, U32 mask) {
```

```
    iomem(reg) |= mask; // Analogamente iomem_low:  
    iomem(reg) &= ~mask;
```

```
}
```

Teorema:

Un sistema T di task periodici, interrumpibili e indipendenti le cui scadenze relative \geq i risp. periodi ha una schedulazione RM fattibile su un solo processore $\Leftrightarrow U_p \leq 1$.

Teorema:

Se per un sistema di task periodici, indipendenti e interrumpibili che sono in fase e hanno scadenze relative \leq i risp. periodi \exists algoritmo a priorità fissa che produce una schedulazione fattibile \Rightarrow anche DM produrrà una schedulazione fattibile.

03/11/2022

\rightarrow I codici numerici all'interno del codice delle funzionalità della nostra Beaglebone è buona norma scrivere all'interno delle macro; stessa cosa per particolari blocchi di codice ricorrenti. Altrimenti il codice diventa illeggibile!

ESEMPIO: `#define gpio1_mask(V) do {
 iomenm(GPIO1_DATAOUT) = (V); } while(0)`

ALTRÒ ESEMPIO: `#define gpio1_on(V) gpio1_out(1<<(V))`

PER ACCENDERE IL LED N°1: `#define _led_on(V) gpio1_on(21+(V))`

PER SPENGERLO: `#define _led_off(V) gpio1_out(21+(V))`

\hookrightarrow Le macro possono includersi in cui file header, il quale viene specificato con un #include all'interno del file di singolare sorgente (la define va dopo rispetto a quella di gptk perché è la quest'ultima libreria che dipende da macro).

Usiamo ora i led come segnale di panico...

`#include "beagleboneblack.h"`

`static inline void panic(int l) {`

`leds_mask(l, 1&0xf);`

`for(;;)`

`loop_delay(3000000);`

`leds_toggle_mask(0xf);` \rightarrow è una delle macro di cui sopra

```
void panic1 (void) {  
    panic(s);  
}
```

```
void panic2 (void) {  
    panic(s);  
}
```

→ Bisogna anche definire i prototipi delle funzioni; lo facciamo in comm.h.

→ Bisogna anche inizializzare la tabella delle eccezioni del processore. Esistono 7 tipi di eccezione:

→ RESET

→ UNDEFINED INSTRUCTION

Ad esempio lo si può fare così: asm volatile ("swi r");

→ SOFTWARE INTERRUPT (definito dal programma utente)

→ PREFETCH ABORT (seg fault per accesso ^{non valido} in memoria per un'istruzione)

→ DATA ABORT (seg fault per accesso non relativo in memoria per recuperare un dato)

→ TRQ (interrupt)

→ FIQ (fast interrupt)

Tocca capire dove sta questa tabella delle eccezioni (o vettore delle eccezioni). Per inizializzarla in realtà, dobbiamo ricorrere a dei registri speciali.

Definiamo la seguente funzione all'interno di un file header:

```
static inline u32 *get_vectors_address (void) {
```

u32 v;

```
    __asm__ __volatile__ ("mrc p15,0,%0,c1,c0,0\n":  
        "=r"(v)::);
```

} Questo non è altro che
 un ASM inline.

return &v;

```
if (v & (1<<13))
```

return (u32 *) 0xffff0000;

```
    __asm__ __volatile__ ("mrc p15,0,%0,c12,c0,0\n":  
        "=r"(v)::);
```

return (u32 *) v;

Nella pagina seguente scrivremo l'inizializzazione effettiva del vettore delle eccezioni.

```
static void init_vectors(void) {
```

```
    volatile U32 *vectors = get_vectors_address();  
    #define LDR_PC 0x00000000  
    vectors[0] = LDR_PC; // istruzione di memoria inizializzata dal Program Counter  
    vectors[1] = LDR_PC; // la fa a mettere nel Program Counter.
```

```
    vectors[2] = LDR_PC;
```

```
    vectors[3] = LDR_PC;
```

```
    vectors[4] = LDR_PC;
```

```
    vectors[5] = LDR_PC;
```

```
    vectors[6] = LDR_PC;
```

```
    vectors[7] = LDR_PC;
```

```
    vectors[8] = (U32)_reset;
```

```
    vectors[9] = panic_1;
```

```
    vectors[10] = panic_2;
```

```
    vectors[11] = {
```

```
    vectors[12] = {
```

```
    vectors[13] = { E così via
```

```
    vectors[14] = {
```

```
    vectors[15] = }
```

Se la posiamo in vectors[0] saltiamo
in vectors[8] (ricordiamo che per il profilo
→ facciamo 2 istruzioni da qui puntate da lì).

Plottare la porta seriale (UART - Universal Asynchronous Receiver/Transmitter):

Nell'ultimo semestre abbiamo visto che il processore è veloce e la porta seriale è lenta. Lo si fa col polling. In particolare, si usa il THR Register e il registro LSR (ISR_UART).

→ Il 5° bit di LSR vale 0 se non posso scrivere sul registro. Vale 1 se posso scriverci.

In un altro file header andiamo a definire:

```
#define UART0_BASE 0x44e0000
```

```
iomemdef(UART0_LSR, UART0_BASE+0x14)
```

```
iomemdef(UART0_THR, UART0_BASE+0x00)
```

```
#define LSR_TXIFOE UART0_LSR<<5
```

Il polling lo si fa
su questo

Scriviamo una funzione che scrive un (solo) carattere.

```
int putc(int ch) {
    while((iorem(UART0_LSR) & LSR_TXFIFO)) ; /* do nothing */
    iorem(UART0_THR) = ch;
    return 1;
}
```

Stampiamo anche una stringa:

```
int puts(const char *st) {
    int n=0;
    while (*st) {
        n += putc(*st);
        if (*st++ == '\n') n += putc('\r'); // Serve per trovare a colonna o equivalente che si trova a capo
                                                // Serve insomma a evitare di rinnovarsi.
                                                // SERT
    }
    return n;
}
```

Stampiamo il valore esadecimale di un ~~numero~~ numero:

```
int pdth(unsigned long n) {
    int i, d, w=0;
    u32 mask;
    mask = 0xffffffff;
    for (i=0; mask!=0; i+=4, mask>>=4) {
        d = (n&mask) >> (28-i);
        w += putc(d+((d>9)?'a'-10:'0')));
    }
    return w;
}
```

Allora il sistema è Schedulabile con RM se: Missing

10/11/2022

Scriviamo una funzione che prende un numero reale numerico e lo stampa sotto forma di stringa:

int putv (unsigned long r) {

char buf[11]; int i, w=0;

if ($r < -10^{10}$) {

w+=putc ('0' + r);

return w;

}

i = 10;

buf[i] = '\0';

while ($r \neq 0$) {

unsigned long t = r/10;

r = r - t * 10

)

Numeri in virgola mobile:

int putf (double r, int prec) {

if ($r < 0.0$) {

w+=putc ('-');

r = -r;

}

w+=putv(r); // qui il

w+=putc ('.');

for (i=0; i<prec; ++i) {

r = r - (long) r;

r = r * 10;

w+=putc ('0' + (long)r

)

return w;

```
 $n = t$ 
buf[-i] = (char)(t + '0');
```

```
}
```

```
w += puts(buf + i);
```

```
return w;
```

```
}
```

Consideriamo ora tutti gli interi (anche negativi):

```
int putd (long v) {
```

```
    int w=0
```

```
    if (v<0) {
```

```
        w += putc('-');
```

```
        v = -v;
```

```
}
```

```
w += putv(v);
```

```
return w;
```

```
}
```

Numeri in virgola mobile:

```
int putf (double v, int prec) { // prec è PRECISIONE
```

```
    if (v<0.0) {
```

```
        w += putc('-');
```

```
        v = -v;
```

```
}
```

```
w += putv(v); // qui il C fa una conversione automatica da double a unsigned long
```

```
w += putc('.');
```

```
for (i=0; i<prec; ++i) {
```

```
v = v - (long)v;
```

```
v = v * 10;
```

```
w += putc('0' + (long)v);
```

```
}
```

```
return w;
```

La printf() scritta così va in PANIC a causa di un data abort (Segfault). Questo perché i numeri in singola precisione vengono utilizzati dal co-processore matematico che di default non è attivo. Bisogna dunque scrivere una funzione che si occupi di attivare il co-processore matematico.

GESTIONE DELLE INTERRUZIONI

Una interruzione hardware è un modo che hanno i dispositivi hardware per segnalare all'CPU la necessità di cambiare flusso di esecuzione, che deve essere diretto verso il gestore delle interruzioni.

Esistono 256 linee di interruzione all'interno del SOC; ciascuna di queste è relativa a un dispositivo hardware che può lanciare un interrupt.

Per gestire la faccenda degli interrupt, si ricorre a dei particolari registri (all'interno dell'Interrupt Controller). Tra questi registri troviamo:

- ~~I~~ Register MIR (→ contiene la maschera per la linea).
- ~~I~~ Register IRR (→ indicano il tipo e la priorità degli interrupt; sono 128: uno per ogni linea di interruzione).
- ~~I~~ Register ISR (→ indicano lo stato della linea prima del mascheramento della linea stessa).
- Pending IRQ (→ indica lo stato della linea dopo il mascheramento).
- Threshold (→ contiene il valore di priorità al di sotto della quale l'Interrupt controller non setta l'interruzione del flusso di esecuzione corrente).
- ISR-IRQ (→ indica l'interrupt tra quelli attualmente pendenti, con priorità maggiore).
- New IRQ flag (→ se vale 1, blocca l'IRQ dal dispositivo e abilita la generazione dell'IRQ successivo).
Interrupt request

La gestione delle interruzioni si basa su 3 livelli:

- Basso (da scrivere in assembly): si preoccupa di salvare lo stato dell'esecuzione.
- Medio (da scrivere in C): si occupa di comunicare la causa e il dispositivo che ha scatenato l'interrupt (dopo che invierà il gestore dell'interrupt).
- Alto (da scrivere in C): non è altro che il gestore dell'interrupt per quel dispositivo.

- Bisogna fare un po' di definizioni per indicare il posizionamento dei registri elencati sopra.
- Altre macro da definire sono riportate di seguito.

```

per #define irq_enable() \
    __asm__ volatile ("cpsie i" ::: "memory")
esso #define irq_disable() \
    __asm__ volatile ("cpsid i" ::: "memory")

```

Scriviamo nel file int una funzione che inizializza l'interrupt controller:

```

static void init_intc(void) {
    iomem(INTC_MIR_SET_BASE + 0) = 0xffffffffUL;
    " " + 8 " "
    " " + 16 " "
    " " + 24 " "
    iomem(INTC_THRESHOLD) = 0xff; // la priorità più bassa di tutte
    irq_enable;
}

```

Scriviamo ora il gestore di livello medio.

```

e) #define NULL ((void *) 0)
typedef void (*isr_t)(void); // funzione che riceve void* e restituisce void
// tipo funzione

static isr_t ISR[NUM_IRQ_LINES];
unsigned long irqcount [NUM_IRQ_LINES] = {0,}; // conta le linee occupate

void bsp_irq(void) {
    iomem(INTC_SR_CLEAR_BASE + 0) = 0xffffffffUL; // zero la sua interrupt bit
    " " + 8 " "
    " " + 16 " "
    " " + 24 " "
    for (i;) { // finché ci sono interruzioni da gestire
        if (iomem(INTC_PENDING_IRQ_BASE + 0) == 0 && // archio alle interruzioni sparse, che sono "phantom";
            " " + 8 == 0 && // se scriviamo la condiz. d'uscita così, lo stiamo
            " " + 16 == 0 && // prendendo in considerazione
            " " + 24 == 0) ) // è praticamente una condiz. d'uscita
            return; // è praticamente una condiz. d'uscita
        irqno = iomem(INTC_SR_IRQ); // se scriviamo la condiz. d'uscita così, lo stiamo
        if (irqno < NUM_IRQ_LINES) {
    }
}

```

```

    ISR = ISR[irqno];
    if (!ISR)
        panic();
    ISR();
    irq_disable();
    ++irqcount[irqno];
}

```

```

jolmenu(NTC_CONTROL) = NEWIRQAGR;           // sempre nel ciclo for
data_sync_barrier();                         // da qui in poi non c'è più
}                                              // alcuna interruzione
}

```

Le interrupt service routine vanno registrate:

```

int register_isr(int n, ISR_t func) {
    if (n >= NUM IRQ + LINES) {           // funz. che abbiamo aver definito noi
        printf("ERROR...\n");
        return 1;
    }
    ISR[n] = func;
    return 0;
}

```

15/11/2022

Schedulazione di job bloccanti e aperiodici:

Il rallentam. / ritardo nell'esecuzione di job può essere di due tipi:

1) TEMPO DI BLOCCO = tempo in cui il job, pur essendo stato rilasciato, non può essere eseguito per qualche motivo esterno.

↳ Il TEMPO MASSIMO DI BLOCCO si è la lunghezza massima dell'intervalllo in cui un job di Ti può essere bloccato.

17/11/2022

Timer:

È un circuito della nostra Beaglebone Black che serve per l'implementazione del tick periferico.

→ AUTO-RELOAD MODE è modalità di caricamento automatico di un registro con un valore prefissato.

→ Per la risoluzione del tempo, se la regola maggiore (è più elevata), abbiamo bisogno di più CPU (per cui avremo meno CPU per i task normali), e viceversa. → TMR0S+0F

Scriviamo in un file header a questo:

```
#define Timer0_Freq 32768 /* Hz */  
#define HZ 1000 /* Tick frequency */  
#define TICK_TLDR (0xoooooooooooo + 1) - (Timer0_Freq / HZ) + 1  
/* VALORE DI OVERFLOW */  
#define Timer0_IRQ 65 /* Linea di interruzione */  
#define Timer0_IRQ_Bank (Timer0_IRQ / 32)  
#define Timer0_IRQ_Mask (Timer0_IRQ % 32)  
#define DMTIMER0_BASE 0x44e05000
```

Tutto quanto è ~~0~~ nella
seconda parola da interruttore
a una altra.

irq_enable();

panic(); //Se entriamo qui, lo cos-

?

iomem(DMTIMER0_TLDR) = TICK_TLDR;

iomem(DMTIMER0_IRQENABLE_CLR) = TCAR -

iomem(DMTIMER0_IRQENABLE_SET) = OVF_IT -

iomem(NTC_1LR_BASE + Timer0 IRQ) = 0xC

iomem(NTC_NIR_CLEAR_BASE + 8 * Timer0 IRQ)

iomem(DMTIMER0_TCLR) = 0x3; // c

loop_delay(10000); // farci passare un
alla volta frequen-

Tutto quanto è ~~0~~ nella
da cui ripartire in caso di
overflow.

iomem(DMTIMER0_ITGR) = 1;

irq_enable();

?

Scriviamo ora la interrupt service routine:

```
static void ist_tick(void) {
```

iomem(DMTIMER0_IRQSTATUS) = OVF_IT_FLAG;

++ticks;

```

iomemdef(DMTIMER0_IRQ_STATUS, DMTIMER0_BASE + 0x78);
    " IRQ_ENABLE_SET      " 0x2C
    " IRQ_ENABLE_CLR      " 0x30
    " TCLR                " 0x38
    " TLDR                " 0x40
    " TTGR                " 0x44
    TCP = Time Critical Register

#define TCAR_IT_FLAG (1U<<2)
#define OVFLITFLAG (1U<<1) // OVFL = overFlow
#define MAT_IT_FLAG (1U<<0)

All'interno di un altro file (Tick) andiamo a scrivere una funzione di inizializzazione.

void init_ticks(void) {
    irq_disable();
    if(register_isr(Timer0_IRQ, isr_tick) != 0) {
        irq_enable();
        panic();
    }
}

iomem(DMTIMER0_TLDR) = TICK_TLDR;
iomem(DMTIMER0_IRQENABLE_CLR) = TCAR_IT_FLAG | MAT_IT_FLAG;
iomem(DMTIMER0_IRQENABLE_SET) = OVFLITFLAG;

if(iomem(NTC_LIR_BASE + Timer0 IRQ) = 0x0); // priorita massima possibile
iomem(NTC_MUR_CLEAR_BASE + 8 * Timer0 IRQ_Bank) = Timer0 IRQ_Mask;
iomem(DMTIMER0_TCLR) = 0x3; // autoReload - start
loop_delay(10000); // faccio passare un po' di cicli di CPU per far abituare la CPU
                    // alla nuova frequenza.

iomem(DMTIMER0_TTGR) = 1;
irq_enable();

Sorviamo ora la interrupt service routine: → con una variabile globale
static void isr_tick(void) {
    iomem(DMTIMER0_IRQ_STATUS) = OVFLITFLAG;
    ticks++;
}

Così non sto ho dichiarato l'uso di ticks
una variabile: viene sparsa incrementata anche se non c'è nient'altro che vi accede

```

Ora un'altra funzione, che serve a dare una cadenza precisa ai tick:

inline void msDelay (unsigned long msec) {

posta a un orologio elementi
per esercizio.

 unsigned int ticks = (msec * Hz + 999) / 1000; → $\frac{x+999}{1000}$: funzione SLVING

 unsigned long expire = ticks + ticks; → Così abbiamo un tick ogni 1000 msec.

 while (ticks < expire)

 cpu_wait_for_interrupt();

}

↳ PROBLEMA: qua il tick $\overset{\text{ticks}}{\rightarrow}$ va in overflow $\overset{\text{ticks}}{\rightarrow}$ a un certo punto: qui abbiamo un certo numero di volte consecutive $\overset{\text{ticks}}{\rightarrow}$ in cui non si $\overset{\text{ticks}}{\rightarrow}$ invoca cpu_wait_for_interrupt().

Soluzione:

~~while (before (ticks, expire))~~ → anche while (ticks < expire),
fare before() è da definire.

↳ Questo è esattamente ciò che è stato fatto nel Kernel Linux.

Sistema bare-metal real-time:

#define MAX_NUM_TASKS = 32

typedef void (*job_t) (void *);

struct task {

 int valid;

 job_t job; ← funz. da eseguire per questo task

 void *arg; ← argomento della funzione

 unsigned long period;

 unsigned long priority; ← ASSUNZIONE: è uno scheduler a priorità fisica (del task)

 unsigned long release_time; ← indica l'istante dell'ultimo rilascio (o, equivalentemente, del rinvio)

 unsigned long released; ← # job rilasciati ma non eseguiti

 const char *name; ← nome del task

};

Ora in un file apposta scriviamo:

struct task taskset [MAX_NUM_TASKS];

int num_tasks;

```

void init_taskset (void) {
    int i;
    num_tasks = 0;
    for (i=0; i<MAX_NUM_TASKS; ++i)
        taskset[i].valid = 0;
}

```

Ricordo di attivare; possiamo pensarlo come
una lista.

```

int create_task (job_t job, void *arg, int period, int delay, int priority, const char *name)
{
    int i; struct task *t;
    for (i=0; i<MAX_NUM_TASKS; ++i) /* cerchiamo una entry all'interno dell'array di
                                         task non valida (in modo tale che vi possiamo
                                         inserire un task per rendere la sua entry valida)*/
        if (!taskset[i].valid)
            break;
    if (i == MAX_NUM_TASKS) // caso in cui abbiamo già il num. max di task
        return -1;
    t = taskset + i;
    t->job = job;
    t->arg = arg;
    t->name = name;
    t->period = period;
    t->priority = priority;
    t->release_time = ticks + delay;
    t->released = 0;
    dq->disable();
    t->valid = 1;
    ++num_tasks;
    dq->enable();
    // restituisco l'indice assegnato al task creato
    return i;
}

```

Ora dobbiamo scrivere lo scheduler. Creeremo ~~scriviamo~~ il file sched.c.
dove verificiamo che i job siano tutti intercompatibili.

```

void check_periodic_tasks(void) {
    unsigned long now = ticks;
    struct task *f;
    int i;
    for (i=0; f=taskset; i<num_tasks; ++f) {
        if (!f->valid)
            continue;
        if (f->release_time <= now && eq(now, f->release_time)) {
            ++f->released;
            f->release_time += f->period;
            ++global_releases;
        }
        ++i;
    }
}

```

Il riferibile globale che conta il num. totale di rilasci;
Da prossima volta, capiremo a che serve.

Se ci pensiamo, siamo implementando uno scheduler basato sul clock (\rightarrow usiamo i tick!).

- REGOLA DI CONSUMO: il budget è decrementato per i task eseguiti, non si accresce come a livello avvio.
- REGOLA DI RIFORNIMENTO: il budget è imposto a $K \cdot p_s$, $K=1,2,\dots$

Significa che il budget torna al valore iniziale in esecuzione task aperiodici per almeno un periodo.

È possibile applicare i test di schedulabilità
il server precastabile non è proprio lo stesso.

Lemma: \rightarrow Se non è possibile il caso peggiore.

In un sistema di task periodici, invece, interviene

SERVER PRECASTINERILE (p_s, e_s) con priorità
si verifica all'istante t_0 se:

- \rightarrow A t_0 è rilasciato un job di tutti i task.
- \rightarrow A t_0 il budget del server è e_s .
- \rightarrow A t_0 è rilasciato almeno un job aperi-
- \rightarrow L'inizio del success. periodo del ser-

messun

24/11/2022

Scriviamo una funzione che effettua la scelta del migliore task da eseguire:

```
static inline struct task* select_best_task(void) {  
    unsigned long maxprio;    struct task *best, f;    int i;  
    maxprio = MAXUINT;        // priorità peggiore possibile come valore iniziale di maxprio  
    best = NULL;              // miglior task finora incontrato  
  
    taskset for (i=0, f=taskset; i<num_tasks; ++f) {  
        if (f->taskset >= MAX_NUM_TASKS) {  
            panic();  
        }  
        // NB: f->taskset è la sommazione tra due puntatori; è un'istruzione "pulita"  
        // perché i due puntatori fanno riferimento allo stesso vettore.  
        // f è il task  
        // che stiamo controllando  
        // correntemente  
    }  
}
```

```
if (!f->valid)
```

```
    continue;
```

```
    ++i;
```

```
    if (f->released == 0)
```

Claramente stiamo cercando task validi e già rilasciati.

```
    continue;
```

```
    if (f->priority < max_prio) {
```

```
        best = f
```

Il migliore task da eseguire è chiaramente quello con priorità migliore.

```
        max_prio = f->priority;
```

```
}
```

```
?
```

```
return best;
```

```
}
```

④

ORA

C'è

Scriviamo una funzione che esegue i task periodici (i.e. sceglie il migliore e lo esegue; sceglie il migliore e lo esegue; e così via):

```
void run_periodic_tasks (void) {
```

```
    struct task *best;
```

```
    for (;;) { // ciclo infinito; il sistema real-time continua ad eseguire  
              // finché non lo spegno fisicamente.
```

```
        best = select_best_task();
```

```
        if (best != NULL) { // ricordati che NON dobbiamo definirlo noi a mano.
```

```
            best->job(best->arg); // esecuzione del job con parametro pari a best->arg.
```

```
            best->released--; // ora c'è un job pendente in meno.
```

```
}
```

```
}
```

Di

più

se

un

task

pro

Ci

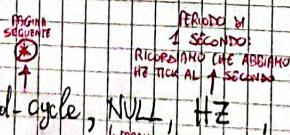
→

Definiamo un task che fa lampeggiare i led:

```
void main(void) {
```

```
    banner();
```

```
    if (create_task (led_cycle, NULL, Hz, 5, Hz, "led-cycle") == -1)
```



MIGRATI PER AL PERIODO \Rightarrow
STANNO LAMPEGGIANDO RATE HUNDREDTH

```

④ static void led_cycle(void *arg) {
    static int state = 1;
    leds_off.mask |= 0xf;
    leds_on_mask(state);
    state = (state + 1) & 0xf;
}

```

→ SE SI ENTRA DENTRO L'IF (\equiv se non si riesce a invocare con succ
esso create_task()) \Rightarrow SI VA IN panic().

→ ALTRIMENTI SI INVOCÀ run_periodic_tasks().

ORA SIAMO PASSATI SULLE SLIDE...

C'è un problema in quello che abbiamo fatto: nell'esecuzione dei task, è possibile che vengano lasciati nuovi job e MENTRE stiamo scandendo la lista dei task di conseguenza, è possibile NON considerare un job appena arrivato ma con priorità migliore. Per mitigare il problema, si sfrutta la variabile globale global_releases che abbiamo definito l'altra volta.

In particolare, in run_periodic_tasks() ...

```

state = global_releases;
best = select_best_task();
if (best != NULL && state == global_releases) {
    ...
    // come prima.
}

```

PROSSIMO OBIETTIVO: costruire uno scheduler con job intercompatibili.

Gi facciamo attenzione dal meccanismo delle INTERRUZIONI.

→ Qui segue un altro problema: ciascun task deve disporre di un proprio stack personale, così che un job che viene interrotto da un altro job non rischia di perdere le informazioni relative alla propria esecuzione.

\Rightarrow Ogni volta che facciamo il cambio di contesto (\equiv cambio di job in esecuzione),

bisogna aggiornare il registro SP

→ Nel nostro caso, però, NON cambiamo stack quando passiamo da modalità "task" (\equiv SYSTEM) a modalità "interruzione" (\equiv IRQ). \Rightarrow quando entro in modalità interruzione, non posso sovrascrivere lo stack (o comunque dovo lasciarlo come l'altro trovato all'inizio).

↳ Quello che facciamo noi è salvare il contesto di esecuzione appena entriamo in modalità interruzione
= STATO DEI REGISTRI
($r_{13}, \dots, r_{15}, cpsr, spsr$)

N.B.: Esistono registri "duplicati", che hanno valori diversi per le due modalità IRQ e SYSTEM.

Quando avviene un'interruzione, i valori di tutti i registri vengono preservati, tranne:

- Il 2° valore (\equiv VALORE IRQ) del link register (r_{14}), che assume il vecchio valore del program counter.
- Il program counter (r_{15}) assume il valore dell'indirizzo della 1° istruzione del gestore dell'interruzione (#).
- CPSR, che salva i flag relativi alla modalità SYSTEM (Q).

L'handler è fatto così:

* - IRQ- handler:

mov r13, r0 // ora 2° valore di r13 = A (= vecchio valore di r0)

sub r0, lr, #4 // ora ~~r0~~ r0 = P+4

mov lr, r1 // ora 2° valore di r14 = B

mtfsr r0, spsr // ora r1 = Q

msr cpsr_c, #(SYS_Mode | NO_IRQ) // nel bit del registro di stato relativi alla modalità di esecuz. e all'attivaz. della interruzione, fanno i valori SYS_Mode e NO_IRQ

push {r0-r2, r12, lr} // push di r0, r1, r2, r12, lr sullo stack.

push {r2-r3, r12, lr} // push di r2, r3, r12, lr sullo stack.)

Questi sono i registri che NON vengono preservati ^{dal compilatore} a seguito di chiamata a funz.

mov r0, sp // ora ~~r0~~ $r_0 = sp \rightarrow r_0 = N-24$.
 sub sp, sp, # (2*4) // ora ho 2 locazioni libere in cima allo stack.
 msr cpsr_c, #(IRQ_MODE | NO_IRQ) // torno in modalità IRQ.
 stmfd r0!, {r13, r14} // push di r13, r14 sullo stack (infatti $r_0 = sp$).
 msr cpsr_c, #(SYS_MODE | NO_IRQ)
 ldr r12, = bsp-irq // ~~indirizzo~~ del gestore di medio livello (scritto in C) caricato in r12 ($\rightarrow r_{12} = \text{rq}$).
 mov lr, pc // $lr = \text{indirizzo di ritorno da -bsp-irq}$.
 bx r12 // invocazione del gestore di medio livello dell'interrupt.
 msr cpsr_c, #(SYS_MODE | NO_INT) // qui disabilito sia IRQ sia FIQ.
 mov r0, sp // ora $r_0 = sp \rightarrow r_0 = N-32 \equiv \text{cima dello stack}$.
 add sp, sp, # (8*4) // ora ~~r0 + 8~~ $\rightarrow sp$ = valore iniziale per modal. SYSTEM (tanto ho sp che punta in cima allo stack).
 msr cpsr_c, #(IRQ_MODE, NO_INT)
 mov sp, r0 // ora solo per la modalità IRQ, $sp = N-32 (= r_0)$.
 ldr r0, [sp, #(7+4)] // ora $r_0 = Q \rightarrow \text{valore di cpsr}.$
 msr cpsr_cxsf, r0 // ora cpsr = valore che dovranno scrivere più nel cpsr.
 ldmdf sp, {r0-r3, r12, lr} // pop di r0, r1, r2, r3, r12, lr dallo stack.
 L'APICELO INDICA CHE IL VALORE DI LR VIENE AGGIORNATO PER LA MODALITÀ SYSTEM ANCHE SE STAVO IN MODALITÀ IRQ.
 mop // no operation.
 ldr lr, [sp, #(6+4)] // recupero del valore del Link register lr (dove siamo andando a mettere ~~l'~~ l'indirizzo di ritorno p).
 mors pc, lr // ora $pc = lr$ ma anche $cpsr = cpsr$.
 // Ora la gestione dell'interrupt è effettivamente conclusa.
 → Se ci facciamo caso, le ultime istruzioni servono a risistemare i registri per riportarli allo stato iniziale.
 → Ora, sopra a questo meccanismo, dobbiamo implementare il nostro meccanismo di gestione delle ~~interruzioni~~ interruzioni.
 → Anti, la prossima volta...

→ SE PROVO QUESTO CHE Succede, Priority Ceiling Enviroment

01/12/2022

È giunta l'ora di implementare lo scheduler con job interdipendibili.

→ Iniziamo ad definire uno stack per ogni task. Al cambio di contesto, sappiamo che in Scheduler.c, se → $T_0, T_1, T_2, T_3, T_n, T_s, M_s, CPSI$ vengono salvati sullo stack di per sé; tutti implementati proprio → gli altri registri vanno memorizzati esplicitamente nel DESCRITTORE DEL TASK.) #define Switch

→ Inoltre, il cambio di stack implica il cambio dello stack pointer Sp. nel struct
che definisce il
task -- asin --

→ Noi cambieremo il contesto alla terminazione della funzione bsp_irq() interna alla gestione dell'interrupt.

→ ATTENZIONE: il cambio di contesto si può fare solo se c'è torniamo nel l'interruzione + esterna, nel caso in cui si apre un annidam. di interruzione

Gesati si era scordato =
(ci va) = perché è un

entro ↗ Tra l'altro, dobbiamo cambiare la definizione dello stack: invece di allocarlo nel linker, lo allociamo in una particolare variabile $\#C$ (un vettore):

```
#define STACK_SIZE 4096  
char stacks[MAX_NUM_TASKS * TASK_SIZE]  
    __attribute__((aligned(STACK_SIZE), section(".stack"))); // Il vettore deve finire  
                                                               // nella sezione stack dell'ad.  
                                                               // diress space.  
const char *stack0_top = stacks + MAX_NUM_TASKS * STACK_SIZE; // È "top" perché gli stack  
                                                               // crescono per inizio da destra.
```

↳ Dopo di che inizializziamo il registro sp alla cima dello stack del task 0.

↳ In tal modo, quando compiliamo, ci vuole molto + tempo: per la sezione "stack", si ha un'inizializzazione esplicita a 0 da parte del compilatore; si può evitare questo facendo in modo che gli stack finiscano (multi-linee) in sect. .bss.

Ormai ci serve un puntatore alla struttura relativa al task che sta eseguendo correntemente. Inizialmente, vale &taskset[0] (\rightarrow si definisce current = &taskset[0] nella funzione init_taskset()).

↳ Aggiungiamo due campi alla struttura task:

→ U32 sp
→ U32 rags[8]

↳ In scheduler.c, scriviamo una macro task (nuova sezione di funzione) che implementa proprio il cambio di contesto.

```
#define Switch_Task (from,to)  
asm volatile ("sh sp,%0\n\t" \ → Spazio dello stack pointed in me=/  
             "ldr sp,%1\n\t" \ → memoria, più esattamente in from->sp/  
             : "=m" (from->sp) : "m" ((to)->sp) \ → dato from e pointed alla strutt del task  
             : "sp", "memory") \ → da che deve riconoscere il processore.
```

(Qui si era scritto "m" →
(ci va "=" perché è un registro di input)

Scriviamo ora la seguente funzione:

void _switch_to (struct task *to) { // il "from" è current.

 save_regs (current->regs); // salvataggio dei registri nel campo regs del current
 load_REGS (to->regs); // carica i registri con valori memorizzati nel campo
 switch_tasks (current, to); // regs del task "to".
 current = to;

→ NB: la funzione è definita con naked_attributes ((naked)) - switch_to(..).
In tal modo, non si ha né prologo, né epilogo (→ il compilatore non aggiunge istruzioni che toccano lo stack e non aggiunge l'istruzione di return).

MA → la routine ci serve; allora dividiamo la funzione con ~~switch~~
l'invocazione alla seguente procedura (MACRO):
#define naked_return() __asm volatile ("bx lr").

→ Dobbiamo definire pure save_REGS() e load_REGS().

#define save_REGS (regs) \
 __asm __volatile ("stmia %0, {r4-r11}" \
 : "r" (regs) : "memory")

#define load_REGS (regs) \
 __asm __volatile ("ldmia %0, {r4-r11}" \
 : "r" (regs) : "r4", "r5", "r6", "r7", "r8", "r9", \
 : "r10", "r11" : "memory")

Al cambio di esecuzione, dobbiamo allora lo stack del job che entra in esecuzione già inizializzato ⇒ l'inizializzazione deve avvenire nella creazione del task.

↳ Sullo stack vengono inseriti i valori dei registri: r0, r1, ..., da recuperare in fase di cambio di contesto ⇒ per far funzionare tutto, prima della

1° esecuzione del nuovo task dobbiamo avere un "contesto fisso". OK, inizializzando:

Guardo:

```
void init_task_context (struct task *t, int mtask) {  
    unsigned long *sp; int i;  
    sp = (unsigned long *) (slack_top - mtask * STACK_SIZE);  
    /* (sp--) = 0x1fUL; */  
    /* (sp--) = (unsigned long) task_entry_point; */  
    /* (sp--) = 0UL; */  
    /* (sp--) = (unsigned long) t; */  
    /* puntatore alla struttura task corrispondente */  
    t->sp = sp;  
    for (i=0; i<8; i++)  
        t->tags[i] = 0UL;  
}
```

→ La funzione task_entry_point() che sta sullo slide è il punto di ingresso di ogni task.

→ Se il task è non valido o non rilasciato ⇒ c'è qualcosa che non va.

→ Nel momento in cui un job termina, viene invocato sys_schedula() per schedularne un altro job.

→ Mentre però il job è in esecuzione, le interruzioni sono abilitate (ed è questo il motivo di irq_enable()).

Perchiamo ora di adattare lo scheduler vero e proprio.

→ In create_task() deve esserci una differenza rispetto a prima: il task zero (lo zero-esimo) è l'IDLE TASK, ovvero è un task fittizio che si ha quando non c'è alcun altro task in esecuzione; tale task, quindi, NON

è periodico, per cui non viene preso in considerazione dallo scheduler quando si ha un cambio di contesto e così via (\rightarrow new relative cycle for di ricezione di un task si parte dall'indice $i=1$ anziché da $i=0$).

\rightarrow Definiamo la seguente variabile globale:

Volatile unsigned long trigger_schedule = 0; \rightarrow indice ~~de~~ lo scheduler va, invocato il prima possibile \rightarrow (0 = false, 1 = true).)

VIENE POSTA A 1 ALLA FINE DELL'ESECUZIONE DI UN TASK.

VIENE POSTA A 0 ALLA FINE DELL'ESECUZIONE DELLA FUNZIONE schedule().

\rightarrow La funzione schedule() non deve tenere f conto esplicitamente dell'esecuzione del task corrente; da qual è il task corrente si occupa il context switch.

\rightarrow Ma perché in task_entry-point() abbiamo chiamato sys_schedule() e non a_schedule()?

\hookrightarrow schedule() viene invocata da sys_schedule() dopo che sys_schedule() si è occupata esplicitamente di mantenere le informazioni per il cambio di contesto.

ACCORTEZZE DI BASSO LIVELLO:

\rightarrow Non posso invocare schedule() se trigger_schedule == 0 e & anche se ci trovo immo in un'interruzione annidata.

\downarrow Dobbiamo tenere traccia della variabile reg_level, che ci dice in quale livello di annidamento dello interrupt ci troviamo (all'inizio chiaramente vale 0).

\rightarrow La non-invocazione di schedule consiste nell'istruzione ldmfd sp,[r0,-12,r1,r2]

\rightarrow Altrimenti, PRIMA di invocare schedule(), bisogna preservare tutti i valori sullo stack; per far ciò, bisogna salvare 32 dello stack pointer.

\rightarrow L'invocazione a schedule() avviene tramite l'istruzione bx r12 del codice Assembly sulle slide.

\rightarrow Se il valore di ritorno di schedule è NULL (~zero), allora non bisogna effettuare lo switch context, e viceversa.

Al termine se puntava sulla cima dello stack del task che deve essere eseguito.

→ Il caso NO SWIICH (.Lnoswitch) deve fare un po' di giochi di prestigio:
carica Q nel registro R₀; poi carica V nel registro R₀ → R₃, R₁₂, bz. ~~bz~~
~~bz~~ Dopo che incrementa il valore di SP in modo da puntare alla cima dello
stack. Infine, carica in PC il valore di SP+8 e SP torna a puntare al valo-
re X. → VEDI STA COSA CON LE SLIDE SE NO MUORI.

06/12/2022

Sezione critica dei job operativi.

13/12/2022

La funzione _sys_schedule():

È invocata in modalità SYSTEM, e non da parte di un handler che si occupi di salvare i registri: dobbiamo salvare noi i registri sullo stack.

Analizziamo tale funzione:

IMPLEMENT

Dobbiamo

→ unsig
chiam

↑

→ basta

- sys_schedule:

Aggiornamento di esp al nuovo valore (è nuova cima dello stack).

str lr, [sp, #-4*2] \Rightarrow QUI SOTTRAGGO 8 A SP (=NUOVO 2 POSIZIONI SULLO STACK)
E SALVO LR SULLA CIMA DELLO STACK.

mrs lr,cpsr \Rightarrow Qui leggo il registro speciale cpsr

str lr, [sp, #4]

2. ldr lr, [sp]

stmfld sp!, {r0-r3,r12-lr} \Rightarrow AGGIORNO SP E MI RIFRENDO I REGISTRI.

b .Lnosub2 \Rightarrow SALTO VERSO IL PEZZO DI CODICE CHE INVOCIA schedule().

↳ Praticamente con questo codice ci stiamo re-investiti nella posizione di codice che chiama schedule() avendo tutti i registri salvati correttamente.

Dopo tutto 'sto casino, il codice non è ancora totalmente corretto: c'è un problema nella funzione schedule(). Di fatto, qui c'è l'invocazione di select-best-task() con le interruzioni ABILITATE (\rightarrow è una funzione protetta perché, per cui non vogliamo eseguirla con gli interrupt disabilitati).

↳ Questo provoca un problema: un interrupt può sopravvenire durante l'esecuzione di select-best-task() e può provocare una nuova invocazione a schedule() rispetto all'invocazione o, schedule() originale.

↳ schedule() è detta funzione RINTRANTE.

Le funzioni rientranti non sono un problema da per sé, ma se giocano con delle variabili GLOBALI, possono causare delle inconsistenze nelle variabili stesse.

Quello che facciamo noi è aggiungere della logica che impedisca le invocazioni a schedule() innestate.

IMPLEMENTAZIONE DELLO SCHEDULER EDF

Dobbiamo aggiungere alla struct task:

→ unsigned long deadline (\rightarrow scadenza relativa; la scadenza assoluta lo chiamiamo al campo priority GIÀ ESISTENTE).

↑ Quando non usiamo EDF, possiamo porre deadline = 0 (che è un valore insensato per noi) → basta :)

Alla funzione create_task() dobbiamo aggiungere un parametro:

→ int type (vale 0 per fixed priority, 1 per EDF).

Nell'implementazione di create_task() dobbiamo cambiare la definizione di t->priority:

```
if (type == EDF) {  
    if (prior_dead) == 0  
        return -1;  
    t->priority = prior_dead + t->releasetime;  
    t->deadline = prior_dead;  
} else /* FPR */  
    t->priority = prior_dead;  
    t->deadline = 0;  
}
```

All'

Ora bisogna prendere una decisione:

• 0 i task FPR hanno priorità migliore dei task EDF.

• 0 i task FPR hanno priorità peggiore dei task EDF.

Nei adotteremo la prima soluzione (→ così, quando implementeremo i task aperiodici, non avremo problemi di sincronizzazione / di rispetto delle scadenze dal punto di vista dei task aperiodici).

↳ In base a questo, modifichiamo select_best_task() aggiungendo queste controlli

```
if (fpz) { // ho già incontrato un task fixed priority  
    if (f->deadline != 0) // if (task di tipo EDF)  
        continue;  
    if (f->priority < max_prio){  
        max_prio = f->priority;  
        best = f;  
    }  
    continue;  
}
```

P2:

gl

che

sta

?

```

if (f->deadline == 0) {           // if (task di tipo FPR)
    for = 1;
    max prio = f->priority;
    best = f;
    continue;
}
if (!edf && || time (f->priority, max prio)) {
    edf = 1;           // edf = ho incontrato un task edf che sto considerando di schedulare
    maxprio = f->priority;
    best = f;
}

```

All'interno della funzione task_entry_point() aggiungiamo:

```

if (t->deadline != 0) {           // if (task di tipo EDF)
    if (time_after (ticks, t->priority)) // if (scadenza viene mancata)
        printf ("EDF task '%s': deadline miss!\n", t->name);
    t->priority += t->deadline period; // ricordiamoci che si tratta di task periodici.
}

```

Watchdog:

È un timer che quando scade, provoca il reset del processore.

↳ È giunto lo scopo di disattivarlo quando non serve, di grazia.

Prendiamo il registro WDT_WTGR: se ci scrivo sopra, il timer del watchdog si resetta.

Il watchdog è di default impostato a un minuto: ci basterà definire un nuovo task che, periodicamente (e.g. ogni 30 secondi), vada a sovrascrivere WDT_WTGR.

```

static void rearm_watchdog (void *arg) {
    arg = arg;           // magari di dire al compilatore che non stiamo usando l'argomento
                         // della funzione.
    iomem(WDT1_WTGR)++; // 
}

```

```

void init_watchdog(void) {
    if (create_task(&watchdog_task, NULL, WDT_Ticks, 1, WDT_Ticks, FPR,
        "watchdog_rearm") == -1)
        puts("ERROR: cannot create task \"watchdog\"\n");
    panic();
}

```

→ Scopo dei LED:

- 1) Comunicare che il sistema funziona correttamente.
- 2) Segnalare quando il processore è attivo e quando no.
- 3) Scopi applicativi.

→ È conveniente utilizzare un solo led per ciascuno scopo (così evitiamo uno spreco di LED).

→ Definiamo un idle_task che va in esecuzione quando non ci sono altri job.

→ Definiamo una funzione heartbeat, che accende il led se è spento e spegne il led → se è acceso. → Questo led se lampeggia, indica che il sistema funziona correttamente.

→ Dobbiamo fare in modo che il led della CPU si spegna ogni volta che stiamo eseguendo un nuovo ciclo dell'idle_task, e si accenda ogni volta che best!=current alla fine della funzione principale di sched.

i.e. quando sto per selezionare un job diverso da quelli che ho appena eseguito.

15/12/2022

SISTEMI REAL-TIME MULTIPROCESSORE

Sono sistemi con + processori, che possono essere dello stesso tipo (→ eseguono job dello stesso tipo) oppure di tipo diverso (→ eseguono job di tipo differente).

Non esamineremo il caso in cui abbiamo n processori dello stesso tipo.

→ che comunque è una complicazione pesante rispetto al caso del singolo processore.

• Fattore di approssimazione = 1,7 •

→ Guardando Wf-FF, notiamo che, per $\beta \rightarrow \infty$ (caso in cui abbiamo solo task picco, massimi), si può raggiungere un'utilizzazione del 100%.

10/

20/12/2022

Constant Bandwidth Server (CBS):

num È un server di task aperiodici corallentato da un periodo Δ , un budget massimo, un budget corrente e una scadenza assoluta.

→ Noi vogliamo implementarlo all'interno del nostro sistema SERT assieme ai task periodici EDF che abbiamo già.

• Definiamo una struttura dati per descrivere il CBS:

struct obs_queue {

 job_t workers[MAX_NUM_WORKERS]; // vettore di job aperiodici da eseguire; in alternativa, avremmo potuto definire molteplici CBS

fat struct task *task; // puntatore alla struttura del task EDF che descriverà ANCHE i job aperiodici gestiti dal CBS.

 void *args[MAX_NUM_WORKERS]; // argomenti da passare ai job dei task aperiodici (una entry per ogni task aperiodico).

 int num_workers; // numero di task aperiodici correntemente attivi (esistenti).

 unsigned int pending[MAX_NUM_WORKERS]; // numero di job per ogni TASK APERIODICO che sono stati rilasciati ma sono ancora pendenti.

All'interno della struttura task (già esistente), dobbiamo aggiungere:

→ unsigned long budget; // se budget == 0, stiamo sicuramente parlando di un task FPR o EDF, perché è un valore che non ha senso per un server CBS (che, quando raggiunge budget 0, viene subito rifiutato).

→ Union {

 unsigned long deadline; // la union è come una struct, ma permette di utilizzare 1 solo campo (o deadline o budget_max, non entrambi), così che risparmiamo memoria. Di fatto, deadline non ci serve per il CBS.

 unsigned long budget_max;

}

Scriviamo ora la routine del CBS che implementa il ciclo di esecuzione del Wf-FF (dei task).

```

static void cbs_server(void *arg) {
    struct cbs_queue *q = (struct cbs_queue *) arg; // vogliamo che l'argomento sia
                                                    // un puntatore alla struttura cbs_q
    int i;
    for (i=0; i < num_workers; ++i)
        if (q->pending[i] > 0)
            break;
    if (i == num_workers) { // in tal caso non c'è alcun job aperiodico pendente
        puts("WARNING: Useless activation of CBS\n");
        return;
    }
    q->workers[i] (q->args[i]); // stiamo invocando la funzione che esegue il job
                                  // dell'i-esimo task
    irq_disable(); // se invece di --q->pending[i] è un'operazione critica
                  // anche dal punto di vista della consistenza: disabilitiamo numerose
                  // reattivamente le interruzioni (così risolviamo la race condition).
    irq_enable();
}
}

NB: per i task aperiodici, il campo released compie significato: stavolta conta il numero
di job dei task aperiodici che sono stati rilasciati.

NB2: ci stiamo discostando dalla teoria su una cosa: se ci facciamo caso, non
stiamo schedulando i job aperiodici secondo una disciplina FIFO, perché abbiamo
i task ordinati secondo una priorità: più hanno una massima spartizione alto nel
l'array, meno alta priorità hanno.

Scriviamo una funzione che rilascia un Worker al servizio CBS:
void bqsafe_activate_cbs_worker(struct cbs_queue *q, int wid) {
    if (wid >= q->num_workers)
        panic("C");
    struct task *t = q->task;
    q->pending[wid]++; // incremento il numero di job pendenti
}

```

$t \rightarrow released++;$ // incremento il num di job rilasciati.

if ($t \rightarrow released == 1$) {

 m32 pd = $t \rightarrow priority * t \rightarrow budget_max;$ // $pd = ds \cdot es$

 m32 tdbp = ticks * $t \rightarrow budget_max + t \rightarrow budget * t \rightarrow period;$ // $tdbp = t \cdot e + cs \cdot p$

 if ($time_after \geq tdbp, pd$) { // EQUIVALENTE A $cs \geq (ds - t) \cdot us$

$t \rightarrow priority = ticks + t \rightarrow period;$

Poli teoria

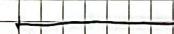
 trigger_schedule = 1;

 ++global_releases;

}

}

}



Ma dove mettiamo la logica per decrementare il budget mano a mano che il job aperiodico continua la sua esecuzione? \rightarrow Nel tick del sistema.

\rightarrow Andiamo in static void iso_tick (void) del file tick.c e aggiungiamo:

if ($current \rightarrow budget > 0$)

 decrease_cbs_budget (current);

Definiamo questa funzione:

void decrease_cbs_budget (struct task *t) {

 t->budget --;

 if ($t \rightarrow budget > 0$);

 return;

 t->budget = $t \rightarrow budget_max;$ // riporto il servizio appena il budget va a zero.

 t->priority += $t \rightarrow period;$

 trigger_schedule = 1; // è cambiata una priorità: per evitare un'eventuale inversione di priorità, bisogna ricarica lo scheduler appena possibile.

Necessitiamo anche di una funzione che crea un nuovo servizio CBS.

```

int init_cbs (unsigned long max_cap, unsigned long period, struct cbs_queue *q,
              cbs_q, const char *name) {
    Budget HASIKA
    cbs_q->num_workers = 0;
    int tid;
    data_sync_barrier();
    tid = create_task (cbs_server, cbs_q, period, 10, max_cap, name, name);
    if (tid == -1) // caso in cui la creazione del task sia fallita.
        return -1;
    cbs_q->task = taskset + tid;
    data_sync_barrier();
    return 0;
}

```

Definiamo una funzione che registra i nostri worker:

```

int add_cbs_worker (struct cbs_queue *q, job_t worker_fn, void *worker_arg) {
    int i = q->num_workers;
    if (i >= MAX_NUM_WORKERS) // caso in cui abbiamo già raggiunto il max
        return -1; // numero di worker possibili.
    irq_disable(); // in realtà sarebbe meglio disabilitare le interruzioni sin da sub
    q->workers[i] = worker_fn; // meglio fare tutto atomicamente
    q->args[i] = worker_arg;
    q->pending[i] = 0;
    q->num_workers++;
    irq_enable();
    return i;
}

```

Nella funzione task_entry_point() dei task ci sono delle cose da fare solo se il task è periodico (i.e. if $t \rightarrow \text{budget} == 0$) :

→ Aggiornamento della priorità : $t \rightarrow \text{priority} += t \rightarrow \text{period};$

Nella funzione in cui creiamo i task, dobbiamo inserire un pezzo di codice ad hoc per il caso CBS:

```
else if (type == CBS) {
    t->priority = ticksing_0; // altrimenti il CBS non funziona (o potrebbe non funzionare) al primo rilascio (vedi teoria).
    t->budget_max = prio_dead;
    t->budget = prio_dead;
}
```

In check_periodic_tasks(), invece, abbiamo tutta una logica che NON deve essere applicata per i task CBS:

```
if (f->budget == 0) {
    ++f->released;
    trigger_schedule = 1;
    ++global_releases;
}
}
```

Definiamo un nostro server CBS e vediamo se funziona:

```
struct cbs_queue cbs0;
void test_cbs_job(Void *arg) {
    struct cbs_queue *q = (struct cbs_queue *) arg;
    struct task *t = q->task;
    static unsigned int count = 0;
    printf ("Tutte le cose che ritieniamo utili riportate sulla porta seriale\n");
}
```

Nella funzione init() (che inizializza tutto) facciamo in modo che venga inizializzato il nostro server CBS:

aggiunge a sua
scrittura: metti un
parametro di parametri
per il processo

```
if (int_cbs (30, 250, &cbs0, "cbs0") == -1)
```

```
    panic();
```

```
if (add_cbs_worker (&cbs0, test_cbs_job, &cbs0) == -1)
    panic();
```

Tocca ancora attivare il nostro job aperiodico: lo facciamo in show-ticks! È possibile
(così non abbiamo nulla di effettivamente aperiodico ma vabbè, è solo una prova).
Si, si
job:

↳ Cioè, stiamo schedulando il job aperiodico periodicamente :)

→ FUNZIONA, YAY!

0011010000

Algjoritmi

→ G

→ U

12/01/2023

Oscilloscopio:

È uno strumento per misurare l'andamento di segnali elettrici nel tempo (e in particolare l'andamento delle tensioni).

Ci sono due ingressi (relativi a due ~~due~~ segnali) che vengono campionati; i valori campionati vengono mostrati su schermo.

ultimo di
ci avvicina
accanata
tack; se

TRIGGER =
(è un punzecchio)

Noi vogliamo far uscire il segnale elettrico dalla nostra bob per farlo acquisire all'osciloscopio. Ad tale scopo possiamo utilizzare le linee GPIO (a cui i led sono collegati). In particolare, esistono due connettori per tirare fuori segnali.

- x. Nonostante abbiamo due soli ingressi, utilizziamo quattro linee: delle linee sono sempre basse per dare un riferimento del valore di "terra", mentre sono le altre due linee alte a costituire il segnale.

→ SEGNALE DI BASE: due onde contrapposte (quando una è bassa, l'altra è alta e viceversa).

to → Se mi vogliano misurare l'heartbeat (il nostro heartbeat ~~non~~ misura 1 secondo, ricordi?), dobbiamo costruire una forma d'onda lunga 1000 tick (\equiv 1000 ms).

→ È possibile misurare la lunghezza d'onda posizionando i due cursori temporali sui due fronti d'onda (l'inizio e la fine dell'onda).

→ Nel nostro caso, la lunghezza d'onda è troppo bassa (sai 770 ms): questo è un problema dell'hardware, perché significa che le interruzioni avvengono troppo spesso. In particolare, abbiamo implementato il meccanismo di misurazione del tempo sul timer 0, che è troppo sensibile alle variazioni di temperatura ed è quindi troppo impreciso per i nostri scopi.

Dunque, dobbiamo usare TIMER-1 ms, un timer basato non più su un circuito RC, bensì su un circuito a quarzo, un materiale ~~non~~ sensibile alle variazioni di temperatura.

↳ Qui possiamo sfruttare il meccanismo dell'ADJUSTMENT, in cui alcuni tick durano un filino di + di 1 ms e altri tick durano un filino di meno di 1 ms ma, mediamente, ci avviciniamo molto al millisecondo. Tale meccanismo va bene se vogliamo misurare il + accuratamente possibile il tempo reale ma non ci interessa molto la durata dei singoli tick; se questo non va bene, disattiviamo l'adjustment.

TRIGGER = fisso la forma d'onda sullo schermo. Dove inizia a distinguere ^{dell'oscillatore, nel punto} l'onda stessa (è un punto di ancoraggio).

Fissato questo, è possibile misurare la variabilità massima del tick quando ADJUSTMENT = 1.