Miosix 4 skyward

Federico Terraneo

Sommario

Questo corso tratta tratta tre argomenti

- Multithreading (in modo abbastanza approfondito)
- Scrittura driver di periferche in Miosix (cenni)
- Affidabilità del software (in particolare non-functional requirements e margini)

Gli argomenti sono principalmente trattati tramite esempi di codice che verranno ampiamente commentati, mentre le slide servono solo a complementare gli esempi con le basi teoriche minime.

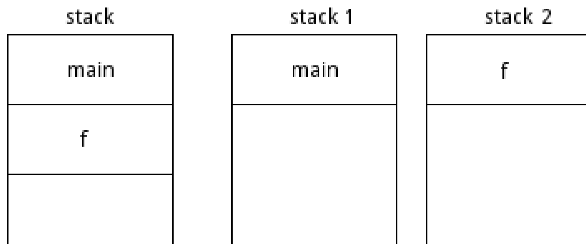
Threading: cos'è un thread

```
void f()
{
    float a=0.0,b=1.0;
    for(int i=0;i<20;i++)
    {
        a=b*(b/20-a);
    }
}

int main()
{
    printf("Before\n");
    f();
    printf("after\n");
}
```

```
void f()
{
    float a=0.0,b=1.0;
    for(int i=0;i<20;i++)
    {
        a=b*(b/20-a);
    }
}

int main()
{
    printf("Before\n");
    pthread_t t;
    pthread_create(&t,NULL,&f,NULL);
    printf("after\n");
}
```



Threading: esempi

0_thread_posix.cpp
1_thread_native.cpp
2_thread_posix_stacksize.cpp
3_thread_posix_join.cpp
4_thread_posix_detached.cpp
5_thread_native_joinable.cpp
6_thread_native_join.cpp
7_thread_posix_param.cpp
8_thread_native_param.cpp
9_thread_param_bug.c

Esecuzioni possibili e non-determinismo

Per massimizzare il parallelismo non si vuole mettere vincoli sull'ordine in cui vengono eseguite le operazioni tra più thread, questo porta al concetto di esecuzioni possibili.
Quando però più thread accedono a delle variabili condivise, è necessario limitare le esecuizioni possibili alle sole che portano a un funzionamento corretto del programma.
Il testing del software in questo caso non è in grado riparare i danni di un cattivo *progetto* del software, dato che alcune esecuzioni possibili accadono molto *raramente*, rendendo il testing poco efficace.

Esecuzioni possibili e non-determinismo

In una esecuzione sequenziale (chiamado le due funzioni una dopo l'altra) è possibile che alla fine sia a che b siano 1? E in caso di thread?

```
void thread1()
{
    if(a==0)
        b=1;
}
```

```
void thread2()
{
    if(b==0)
        a=1;
}
```

Esecuzioni possibili e non-determinismo

```
void thread1()
{
    if(a==0)
        b=1;
}
```

```
void thread2()
{
    if(b==0)
        a=1;
}
```

Esecuzioni possibili e non-determinismo

In una esecuzione sequenziale (chiamado le due funzioni una dopo l'altra) è possibile che alla fine sia i sia diverso da 0? E in caso di thread?

```
int i=0;

void thread1()
{
	i++;
}

void thread2()
{
	i--;
}
```

Esecuzioni possibili e non-determinismo

```
void thread1()
{
    MOVE.L  I,  D0
    ADD.L   #1, D0
```

```
void thread2()
{
```

```
    MOVE.L  I,  D0
    SUB.L   #1, D0
    MOVE.L  D0, I
}
```

```
    MOVE.L  D0, I
}
```

Mutex

Un mutex è un tipo dati astratto con due operazioni: *lock* e *unlock*.
Un mutex può essere libero o occupato.
Un thread che fa *lock* su un mutex libero lo rende occupato, e diventa il propietario del mutex.
Se altri thread (uno o più) tentano una *lock* su un mutex occupato, vengono messi in una lista di attesa, e si *bloccano*.
Appena il thread propietario del mutex fa la *unlock*, viene scelto un thread dalla lista di attesa che diventerà il propietario, e viene sbloccato.
Se il propietario del mutex fa la *unlock* e la lista di attesa è vuota, il mutex torna libero.

Mutex

10_mutex_posix.cpp
11_mutex_native.cpp
12_mutex_bug.c
13_mutex_deadlock.c
14_mutex_deadlock2.c
Use case: shared/events/EventBroker skyward

Condition variable

I Mutex sono utili per serializzare l'accesso a varibili condivise, limitando le esecuzioni possibili alle sole desiderate.

Un thread può essere messo in attesa durante una *lock*, se il mutex è occupato.

In alcuni casi però, si vuole mettere un thread esplicitamente in attesa di un altro thread, perchè ad esempio per continuare ha bisogno di un dato che è calcolato da un altro thread.

Per questo ci sono le condition variable.

Condition variable

Una condition variable è un tipo dati astratto con tre operazioni: *wait*, *signal* e *broadcast*.
Un thread che fa una *wait* viene messo in una lista di attesa e si blocca.
Un thread che fa una *signal* prende un thread bloccato nella lista di attesa e lo sveglia.
Un thread che fa una *broadcast* sveglia tutti i thread nella lista di attesa.
Una *signal* o *broadcast* quando la lista di attesa è vuota non fa niente.

Condition variable

15_condvar_posix.cpp
16_condvar_native.cpp
17_condvar_bug.c

Thread design patterns

18_producer_consumer.cpp
19_synchronized_data_structures.cpp
20_thread_as_member_fn_bug.cpp
21_thread_as_member_fn.cpp
22_function_bind_basics1.cpp
23_function_bind_basics2.cpp
24_function_bind_basics3.cpp
25_e20.cpp
26_e20_threadpool_problem.cpp
27_e20_threadpool_solution.cpp
Use case: Logger SD writing pipeline skyward and how it solves the problem of 1ms sampling with 1s write latency

Device driver per Miosix

Domanda: come fa il software a interagire con l'hardware?

Device driver per Miosix — Registri di periferica

Il metodo più comune è quello dei registri di periferica. Le periferiche hardware si presentano al software come un set "registri", che non sono altro che locazioni di memoria mappate a specifici indirizzi nello spazio di indirizzamento, e quindi accessibili tramite software.

Caveat:

- I registri di periferica non vanno confusi coi registri della CPU.
- I registri di periferica sono mappati ad indirizzi fisici, non virtuali, quindi nei sistemi operativi con protezione della memoria (Linux, Mac, Windows) sono accessibili solo da dentro il kernel.
  Questo è vero anche in Miosix se si usano i processi.

Device driver per Miosix — Registri di periferica

I registri di periferica sono per certi versi paragonabili a delle variabili allocate in RAM, in quanto

- sono accessibili allo stesso modo (essendo mappati nello stesso spazio di indirizzamento)
- in molti casi sono leggibili e scrivibili dal software (alle volte però capita di avere a che fare con registri read-only).
- hanno una dimensione, solitamente di 8, 16 o 32bit, esattamente come gli unsigned char, unigned short e unsigned int.

Device driver per Miosix — Registri di periferica

Ciononostante, ci sono delle differenze fondamentali tra i registri di periferica e le variabili

- Quello che viene scritto in questi registri causa azioni nel mondo reale (l'accensione di un LED, l'attivazione di un ADC, l'invio di un carattere tramite una porta seriale, etc.)
- Si trovano a specifici indirizzi di memoria. Quando una variabile viene allocata sullo stack o sull'heap, al programmatore non importa se viene allocata all'indirizzo 0xbffffc60 o 0xbffffe12, mentre se il registro di periferica si trova all'indirizzo 0x101e5018 occorre essere sicuri di stare scrivendo esattamente a quell'indirizzo, o non si otterranno i risultati voluti.
- I registri di periferica non sono ad uso esclusivo del programmatore, come le variabili. Sono condivisi tra il software e l'hardware. Per esempio l'hardware puo‘ decidere di flippare bit all'interno dei registri per segnalare eventi specifici, cosa che non succede con le normali variabili.

Device driver per Miosix — Registri di periferica

Come si fa a sapere quali periferiche si hanno a disposizione, quali registri ha una specifica periferica, a che indirizzo sono mappati e come usarli? Per un microcontrollore le periferiche disponibili sono documentate dal produttore in un documento, solitamente chiamato “datasheet” o “programming guide”.

I datasheet sono in genere disponibili sul sito del produttore del microcontrollore. Per esempio il datasheet dell’ATmega328, il microcontrollore usato nell’Arduino, si trova sul sito della Atmel, mentre il datasheet dei microcontrollori stm32 si trova sul sito di ST.

Device driver per Miosix — Registri di periferica

Metodo 1:
Assumiamo che nel micrcontrollore ci sia un registro a 32bit, chiamato IODIR0 all'indirizzo 0xe0028008, e che vogliamo scriverci zero.
Come si fa?

```
void clearReg()
{
    (*((volatile unsigned int *) 0xe0028008)) = 0;
}
```

Ok, un minimo di spiegazione, dato che il codice potrebbe non essere così intuitivo. Prima l'indirizzo viene castato a "volatile unsigned int *", ossia un puntatore a un intero a 32bit. Poi il "*" sulla sinistra dereferenzia il puntatore. Infine, il valore zero viene scritto in quella locazione di memoria. Il "volatile" serve ad evitare ottimizzazioni del compilatore, come il reordering delle istruzioni, che possono creare problemi in quanto il compilatore non è a conoscenza dei side effect "nel mondo reale" causati dall'accesso ai registri di periferica.

Device driver per Miosix — Registri di periferica

Per aumentare la leggibilità del codice, si può usare una macro come:

```
#define IODIR0 (*((volatile unsigned short *) 0xe0028008))
```

Così facendo, il codice può essere riscritto in questo modo

```
void clearReg()
{
    IODIR0 = 0;
}
```

Questo codice rende meglio l'idea di quello che sta succedendo, ossia l' assegnamento del valore zero al registro IODIR0. Inoltre, l'uso del nome simbolico del registro (IODIR0) invece che dell'indirizzo di memoria (0xE0028008) rende il codice self documenting. E' pratica comune raggruppare tutte queste macro in un header, e usare una #include "file.h" in tutti i sorgenti che devono accedere alle periferiche.
Ancor meglio, alcuni produttori di microcontrollori rendono disponibili degli header già pronti, da scaricare e includere nei propri progetti.

Device driver per Miosix — Registri di periferica

Metodo 2:
Raramente una periferica, come un ADC o una seriale, ha un solo registro. Il caso tipico è quello in cui ogni periferica viene controllata da un insieme di registri, che spesso sono mappati ad indirizzi di memoria consecutivi. Questo rende possibile raggrupparli in una struct, come questa

```
struct GpioPeripheral
{
    volatile unsigned int CRL;
    volatile unsigned int CRH;
    volatile unsigned int BSRR;
    volatile unsigned int BRR;
};
#define GPIO ((struct GpioPeripheral *)0xfeeeaba0)
```

La struct definisce in un sol colpo tutti i registri della periferica, mentre la macro da un nome "self documenting" ad un puntatore alla struct, mappato all'indirizzo corretto.

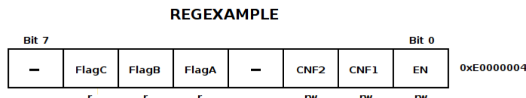
Device driver per Miosix — Registri di periferica

Usando questo metodo il codice per azzerare il registro CRL della periferica GPIO è questo

```
void clearReg()
{
    GPIO->CRL = 0;
}
```

Non ci sono differenze, neanche di performance tra i due metodi, entrambi portano allo stesso risultato.
Alcuni produttori forniscono header con questa rappresentazione delle periferiche, altri con solo le macro, quindi è necessario conoscere entrambi i metodi.

Device driver per Miosix — Registri di periferica



La figura mostra un tipico registro di periferica così come viene documentato nei datasheet. Tra le informazioni disponibili ci sono il nome del registro (REGEXAMPLE) e l'indirizzo del registro (0xE0000004).
Come si può vedere ci sono tre bit leggibili e scrivibili (indicati come rw), e tre bit solo leggibili (r). Ci sono infine due bit inutilizzati.
Una possibile rappresentazione in codice di questo registro può essere

```
#define REGEXAMPLE (*((volatile unsigned char *) 0xe0000004))

#define REGEXAMPLE_EN (0x01)
#define REGEXAMPLE_CNF1 (0x02)
#define REGEXAMPLE_CNF2 (0x04)
#define REGEXAMPLE_FLAGA (0x10)
#define REGEXAMPLE_FLAGB (0x20)
#define REGEXAMPLE_FLAGC (0x40)
```

La prima macro definisce il registro, le altre sono opzionali e servono per dare un nome anche ai bit all'interno del registro.

Device driver per Miosix — Registri di periferica

Il fatto che all'interno di un registro ci siano più bit con funzioni indipendenti fa sorgere la necessità di alterare un singolo bit all'interno di un registro. Il codice per fare ciò non è difficile, ma può essere non poi così immediato la prima volta che lo si vede.

Questo è un esempio di codice che setta a 1 il bit EN lasciando inalterati gli altri

```
REGEXAMPLE |= REGEXAMPLE_EN;
```

Mentre questo è il codice per settare a 0 lo stesso bit

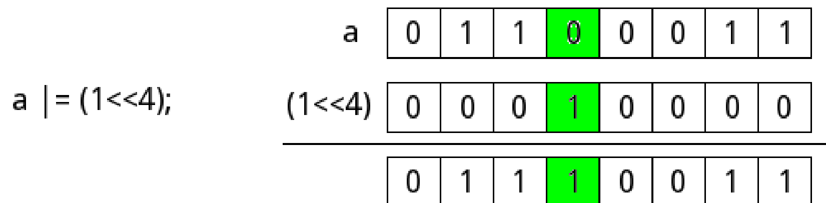
```
REGEXAMPLE &= ~REGEXAMPLE_EN;
```

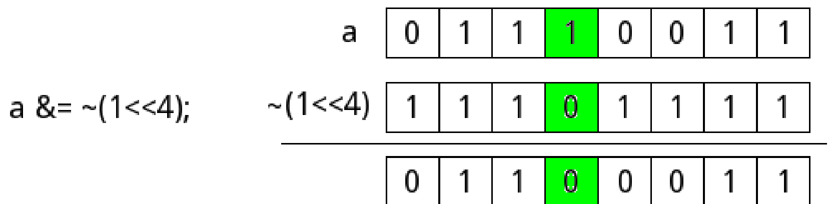
Infine, il codice per testare se il bit FLAGA è a 1 è il seguente

```
if(REGEXAMPLE & REGEXAMPLE_FLAGA)
{
    [...]
}
```

Il trucco sta nell'usare le operazioni logiche di or ed and con una bitmask.

Device driver per Miosix — Registri di periferica





Device driver per Miosix

28_peripherals_GPIO_raw.cpp
29_peripherals_quadrature_encoder.cpp
30_interrupts.cpp
Use case interrupts: SPI skyward
Lunga lista di cosa NON si può fare in un interrupt
31_interrupts_e20.cpp

Affidabilità — Stima occupazione RAM

Quando si sviluppa una applicazione per PC di solito non ci si preoccupa eccessivamente per l'uso della memoria:

- Al giorno d'oggi i PC hanno molti GByte di RAM
- Il sistema operativo supporta lo *swapping*
- Il sistema operativo condivide pagine di memoria tra più processi, rendendo difficile relazionare la memoria occupata dai singoli processi con quella totale
- Diversi PC hanno diverse quantità di RAM, quindi il massimo che si può fare è ridurre l'occupazione di RAM, ma non si può essere mai certi che l'applicazione funzionerà sempre (es: l'applicazione viene lanciata insieme ad altre che occupano tanta RAM, o su un PC molto vecchio con poca RAM)

Questo è accettabile solo in un contesto non realtime (swapping) o critico (nessuna garanzia).

Affidabilità — Stima occupazione RAM

Un sistema operativo realtime per microcontrollori è molto più semplice:

- Niente swapping, niente memoria virtuale, niente cache: accesso a memoria ha tempi deterministici
- Memoria non paginata, e niente condivisione delle pagine, la memoria occupata è la somma della memoria dei singoli task (facile da calcolare)
- Quando si scrive una applicazione realtime si consocno tutti i task di cui è composta e su che hardware verrà fatta funzionare, quindi si sa esattamente quanta RAM c'è e si può garantire che sia abbastanza
- Inoltre il fatto che la memoria è poca (da qualche KByte a qualche MByte) rende l'analisi dell'uso della memoria fondamentale

Ok, ma come si fanno questi conti sull'occupazione di RAM?

Affidabilità — Stima occupazione RAM

Ci sono due metodi complementari, analisi statica del codice e misurazione a runtime.

Analisi statica

- Può essere fatta a design time o a posteriori sul codice
- Tende a sottostimare la memoria utilizzata se non si considerano gli overhead dovuti all'allineamento delle strutture dati, dei chunk nello heap ecc, spilling di variabili sullo stack durante codegen...
- Può essere fatta a grana grossa (solo le strutture dati più grandi) o essere più dettagliata (richiede però molto tempo, non conviene)
- Consente di trattare tutti i code path, anche quelli che vengono eseguiti raramente

Misure a runtime

- Molto precise, considerano anche gli overhead
- Considerano solo i code path in cui entra il codice, ci potrebbero essere altre allocazioni non considerate

Affidabilità — Stima occupazione RAM — Memory layout

Cominciamo con il tipico memory layout in un microcontrollore.

Un micro ha in genere due memorie, una FLASH e una RAM

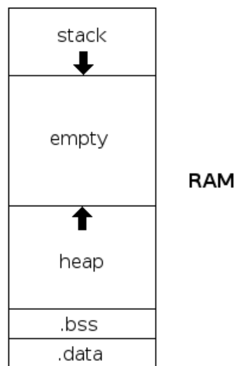
FLASH

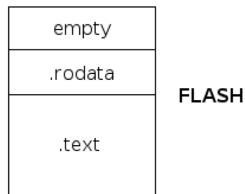
- Preserva il suo contenuto anche in assenza di corrente
- Quando il microcontrollore è operativo di solito è *read-only*, mentre è scrivibile solo durante quando la CPU del micro è ferma (salvo eccezioni)
- E' mappata nello spazio di indirizzamento del processore, che può quindi eseguire codice direttamente dalla FLASH

RAM

- Perde il contenuto ogni volta che si toglie corrente
- E' leggibile e scrivibile dalla CPU

Affidabilità — Stima occupazione RAM — Memory layout

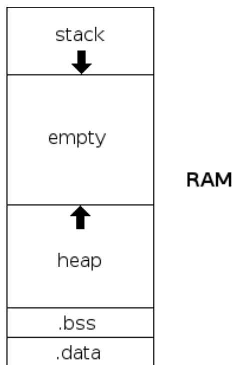


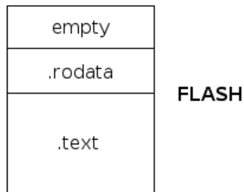


Cominciando dalla memoria FLASH, contiene

- Una prima area chiamata "sezione .text" con il codice (i.e, gli opcode delle istruzioni assembler che il compilatore C/C++ genera)
- Una seconda area, la .rodata con le costanti usate nel programma
- Una copia della sezione .data (non mostrata nel grafico per semplicità)
- La rimanente pare della FLASH è vuota

Affidabilità — Stima occupazione RAM — Memory layout



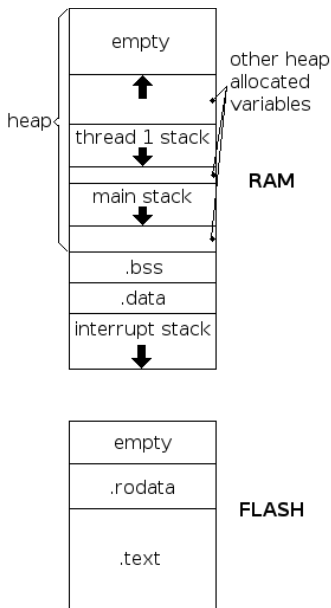


Passando alla RAM

- .data, sezione che raggruppa tutte le variabili globali inizializzate
- .bss, le variabili globali non inizializzate
- lo heap, area di memoria usata per malloc()/new
- lo stack, usato per i record di attivazione delle funzioni

La dimensione dello stack e dello heap **non** sono noti a compile-time, ma crescono durante l'esecuzione del programma. Possibile collisione e conseguente undefined behaviour.

Affidabilità — Stima occupazione RAM — Memory layout



Passiamo ora al memory layout in Miosix, che si differenzia nell'uso della RAM

- C'è uno stack dedicato agli interrupt (512 Byte), messo in basso in modo che un overflow causi un reboot invece che undefined behaviour
- .data e .bss come prima
- Il resto è tutto heap. Gli stack dei thread sono allocati nello heap (approccio stack-in-heap). Viene impiegato watermarking per rilevare stack overflow

Affidabilità — Stima occupazione RAM — What if?

Cosa succede in caso di memory overflow in Miosix?

- Heap full: malloc() ritorna NULL, e l'operator new lancia un eccezione di tipo std::bad_alloc, il programmatore può gestire la situazione. (Nota: se il kernel viene compilato con le eccezioni del C++ disattivate, uno heap full causa un reboot immediato, sia con malloc() che con new)
- Stack overflow in un thread: il kernel usa due metodi complementari per identificare a runtime stack overflow, watermarking e check dello stack pointer. Questi check vengono fatti a ogni context switch. Se viene rilevato uno stack overflow viene effettuato un reboot. I check non sono però in grado di identificare il 100% dei casi.
- Stack overflow nello stack degli interrupt: l'hardware rileva questa condizione e viene effettuato un reboot.

Affidabilità — Stima occupazione RAM — Static analysis

Analisi statica sul consumo della memoria, esempio sulla codebase di skyward

Affidabilità — Stima occupazione RAM — Runtime meas.

Miosix fornisce la classe *MemoryProfiling* in *miosix/util/util.h* che consente di sapere a runtime l'occupazione di stack e heap.
Le informazioni sullo heap sono globali, mentre le funzioni relative allo stack ritornano i valori dello stack del thread da cui vengono chiamate.

- Le funzioni *current* ritornano la memoria libera nel momento in cui vengono chiamate
- Le funzioni *absolute* ritornano la minima memoria libera a partire dal boot

```
class MemoryProfiling
{
public:
    static void print(); //Print memory summary to stdout
    static unsigned int getStackSize();
    static unsigned int getAbsoluteFreeStack();
    static unsigned int getCurrentFreeStack();
    static unsigned int getHeapSize();
    static unsigned int getAbsoluteFreeHeap();
    static unsigned int getCurrentFreeHeap();
};
```

Affidabilità — Stima occupazione RAM — Best practices

- E' opportuno fare periodicamente analisi statica sulla codebase (code review) sommando le strutture dati "grandi" (buffer, stack dei thread)
- Ogni thread deve avere una modalità debug che periodicamente logga/stampa via printf quanto stack libero ha.
- Lo stack di ogni thread deve essere dimensionato in modo che ci sia un margine di almeno ∼1KB rispetto al massimo uso di stack visto durante il logging
- La modalità di debug deve anche stampare periodicamente l'uso dello heap
- Ci deve essere un sufficiente margine di heap libero

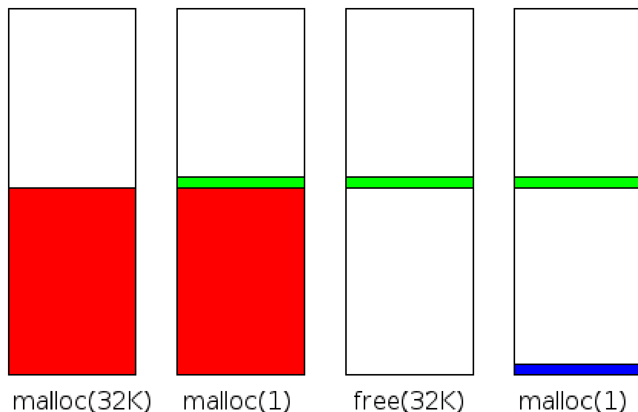
Affidabilità — Heap fragmentation

Considerate una architettura con uno heap da 64KB. Immaginate di eseguire questa sequenza di chiamate.

```
void *a=malloc(32*1024);
//Total allocated RAM=32KB
void *b=malloc(1);
//Total allocated RAM=32KB+1
free(a);
//Total allocated RAM=1
void *c=malloc(1);
//Total allocated RAM=2
void *c=malloc(32*1024);
```

L'ultima malloc() fallisce, perchè?

Affidabilità — Heap fragmentation



Anche se ci sono 64K-2 bytes liberi nello heap (senza contare l'overhead), non c'è un blocco **contiguo** di 32KB, quindi l'ultima malloc fallisce!

Affidabilità — Heap fragmentation — Best practices

I blocchi di memoria "grandi" dovrebbero essere allocati tutti all'inizio, quando l'heap non è frammentato.

Esempio: pipeline di scrittura su disco di skyward alloca due buffer da 32KB appena avviata e non li dealloca più fino a quando il file di log viene chiuso, invece di continuare a fare malloc/free di questi buffer.

Quando un blocco di memoria è da considerarsi "grande"? Quando è dell'ordine di 1/10 .. 1/20 o più della dimensione dello heap.

Affidabilità — Consumo di CPU

In un sistema realtime ci sono task che devono essere eseguiti entro una deadline.

In molti casi i task sono periodici, e devono essere completati prima del prossimo periodo.

E' importante stimare la percentuale di utilizo della CPU e verificare che il carico sia accettabile.

Esempio: se un task impiega 10ms e deve essere eseguito 1000 volte al secondo evidentemente il sistema non funzionarà!

Anche in questo caso è possibile effettuare sia una analisi statica che un profiling a runtime.

Affidabilità — Consumo di CPU — Analisi statica

Analisi statica sull'utilizzo di CPU, esempio sulla codebase di skyward

Affidabilità — Consumo di CPU — Profiling

E' possibile utilizzare un GPIO libero per misurare la percentuale di CPU usata da un task.

```
typedef Gpio<...> profileGpio;

void thread(void *)
{
    for(long long t=getTick();;t+=sleepTime)
    {
        profileGpio::high();
        doPeriodicTask();
        profileGpio::low();
        Thread::sleepUntil(t);
    }
}
```

Connettendo il GPIO a un oscilloscopio, si ottiene un'onda quadra il cui periodo è quello del task, e il cui duty cycle è la percentuale di CPU utilizzata.