Sistemi Operativi A.A.2003/2004

Come realizzare un sistema operativo

Obiettivo del progetto:

Creare un sistema operativo multitask minimale

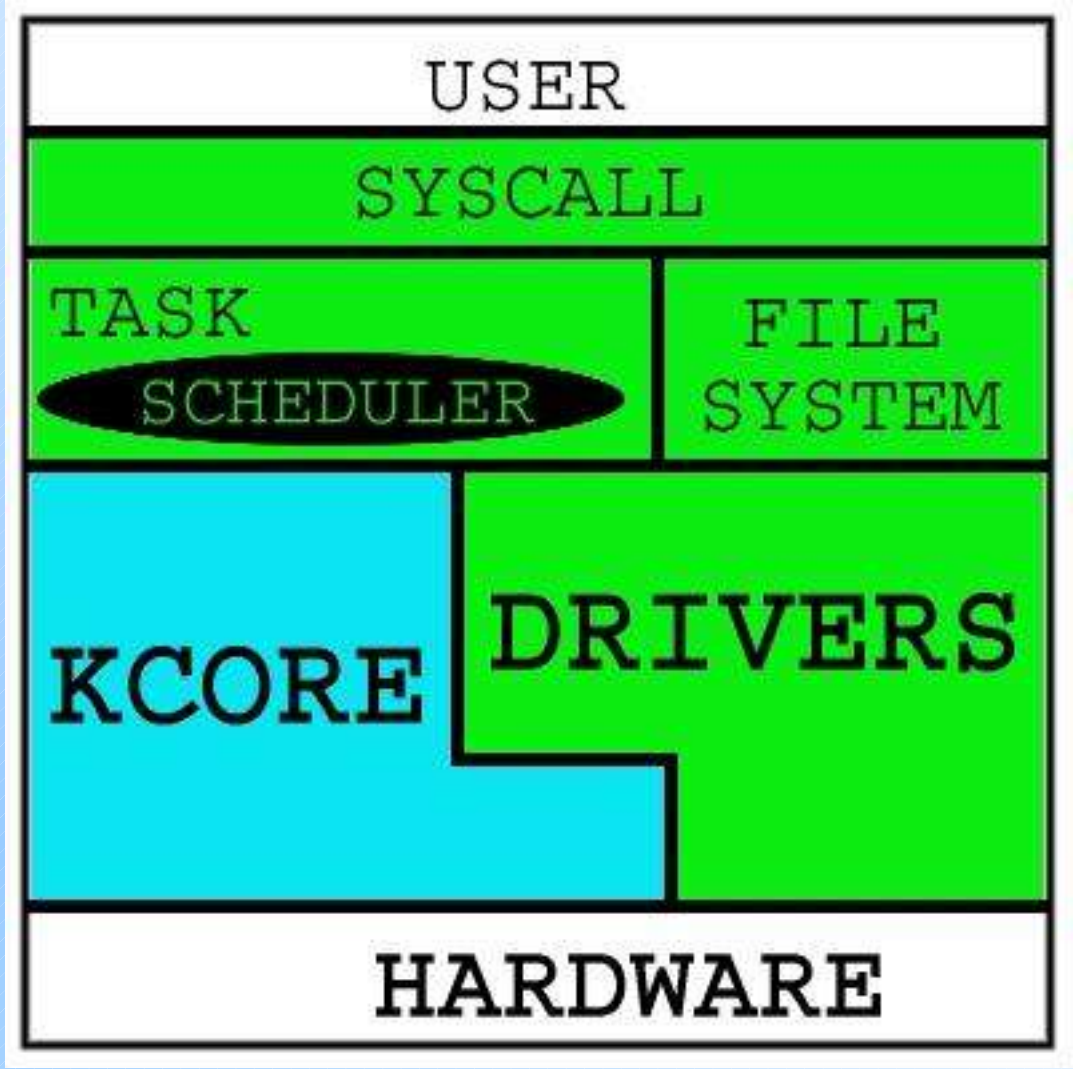
Fase 1: il progetto su carta...

Strumenti di sviluppo:
- sistema (Linux);
- linguaggio (C, compilatore gcc);
- un emulatore (Bochs);
- una macchina 'reale' per una prova 'fisica';
- un boot loader (GRUB).

Piattaforma:
- Intel 386 (per la sua grande diffusione).

Raccolta di informazioni:
- manuali (sia cartacei che on-line);
- progetti esistenti.

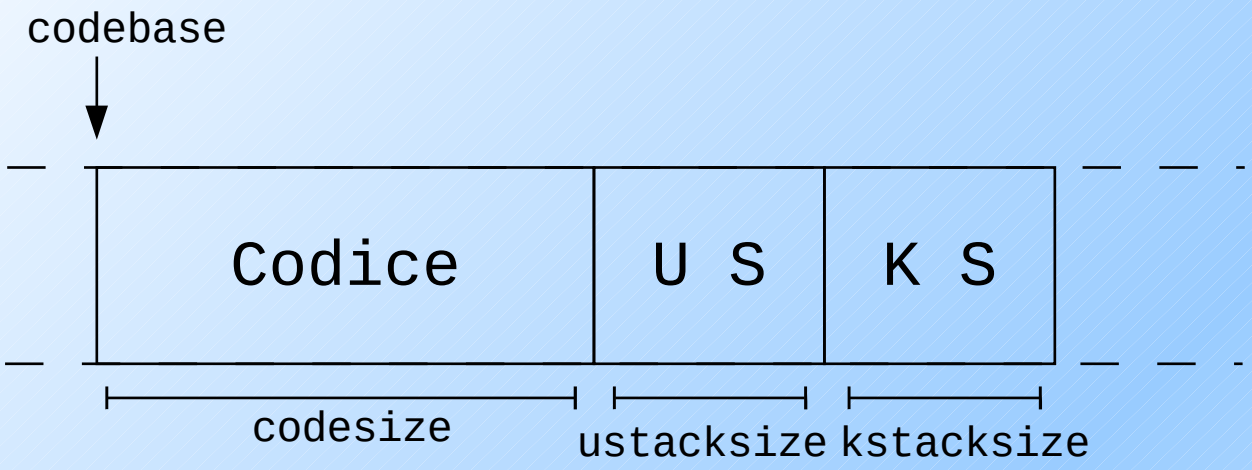
Il Kernel: organizzazione stratificata del codice



Kernel Core (nucleo)

- Il nucleo del Kernel lavora sui task
- Ad ogni livello ed ogni task e' associato uno STACK
- A livelli piu' ALTI (2,1) l'esecuzione del codice e' legata ad un particolare TASK
- A livelli piu' BASSI (0,1) viene eseguito solo codice “SICURO”
- Il passaggio di Livello e' associato ad un INTERRUPT

Come descrivere un Task?



Spazio in memoria:

- Codice da eseguire
- Stack Utente (L2)
- StackKernel

CPU

GP	eax	ebx	ecx	edx
	esi	edi	ebp	esp
FL	eip	eflags		
SG	esi	edi	esp	ebp

Stato del processore

- Contenuto dei registi(X2)

Creazione del kernel: i task

Struttura del task

```
struct   task_t
    {
    signed int pid;
    struct task_mem_t mem;
    unsigned int modo;
    struct task_regs_t userRegs;
    struct task_regs_t kernelRegs;
    };
```

Indipendente dalla piattaforma

Stato del processore

```
struct   task_regs_t
    {
    unsigned int eip,cs,eflags;
    unsigned int eax,ebx,ecx,edx;
    unsigned int esp,ebp,esi,edi;
    unsigned int ds,ss,es,fs,gs;
    };
```

Dipendente dalla piattaforma!!!

Segmento di memoria del task

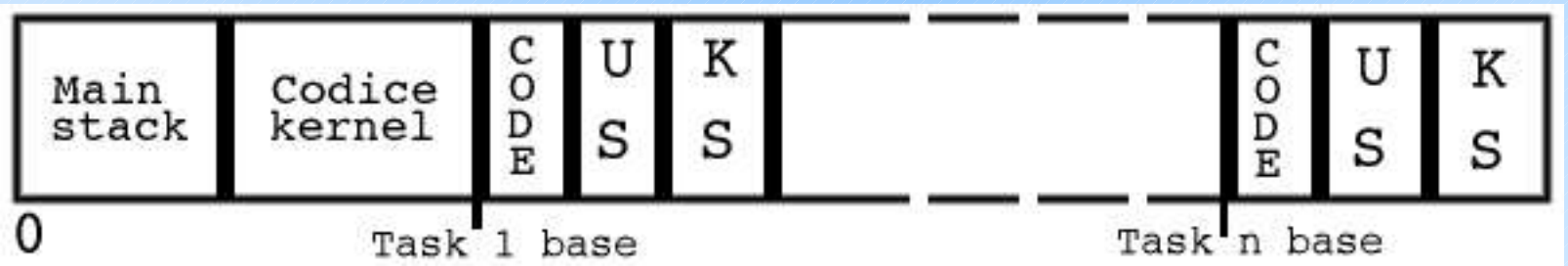
```
struct  task_mem_t
    {
    mem_t base;
    mem_t codesize, ustacksize, kstacksize;
    };
```

Indipendente dalla piattaforma

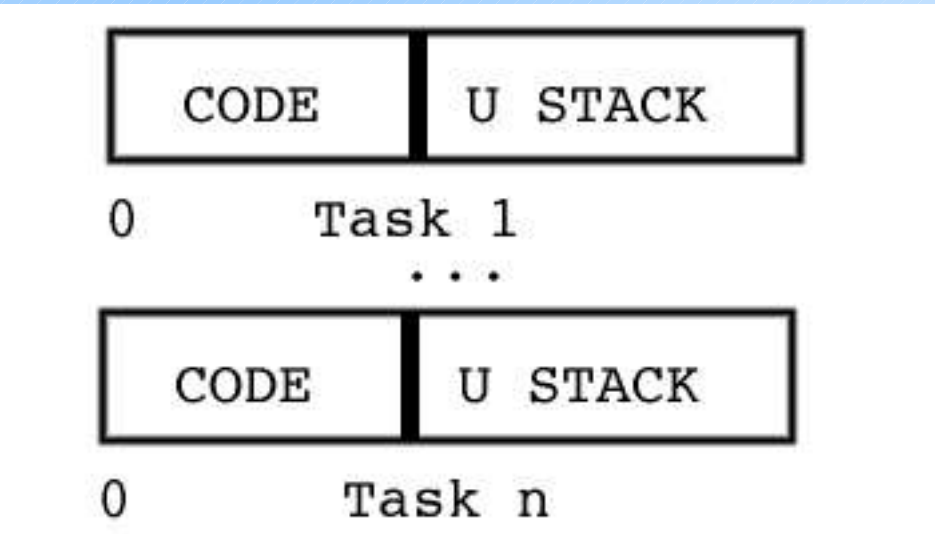
I Task

Schematizzazione dell'area di memoria:

La memoria vista dal kernel:



La memoria vista dai task:



Creazione del kernel: i task

L'indirizzo base *fisico* di ogni task
e' differente da quello *logico*.
Ogni task 'crede' di avere *base* zero!

Questo e' stato possibile sfruttando la MMU
(Memory Management Unit).

La MMU fornisce anche un certo livello di
protezione della memoria, anche se il sistema
usato attualmente dipende fortemente dalla
architettura del sistema!

Livelli di esecuzione

Il sistema si trova sempre in uno dei suoi 3 livelli di esecuzione.

Per ogni livello sono definiti

- **Privilegi**
  opcode riservati (cli,sti,in,out...)
  accesso a zone di memoria 'speciali'
- **Stack**
- **Insieme di Task**
  solo per livelli piu' alti!

Livelli di esecuzione
-
L2 user mode

- Ha i privilegi piu' bassi
- E' associato ad un task specifico
- Qui si trovano i task a livello utente

Livelli di esecuzione
-
L1 kernel mode (task)

- Modo di esecuzione intermedio
- Quasi tutti i privilegi del modo puro ma e' ancora legato al task
- Interrupt disabilitati ma modifcabili tramite 'CLI' e 'STI'
- Qui si trovano task che hanno chiamato delle syscall e alcuni task di sistema (come il task *idle*)

Livelli di esecuzione
-
L0 kernel mode (puro)

- Massimi privilegi
- Usa uno stack indipendente
- NON e' associato ad alcun task!
- Qui si trovano le routine di sistema

Livelli di esecuzione
-
L0 alcuni problemi

- I livelli non corrispondono ai livelli di privilegio del processore
- Tutto viene eseguito con privilegio massimo.
- Il livello 0 non deve essere interroto, ma non c'e' modo di controllare l'abilitazione delle interruzioni. Il sistema puo' crashare!
- Il livello 0 dovrebbe essere *nascosto* al resto del kernel e del sistema!!!

Level switch e ISR

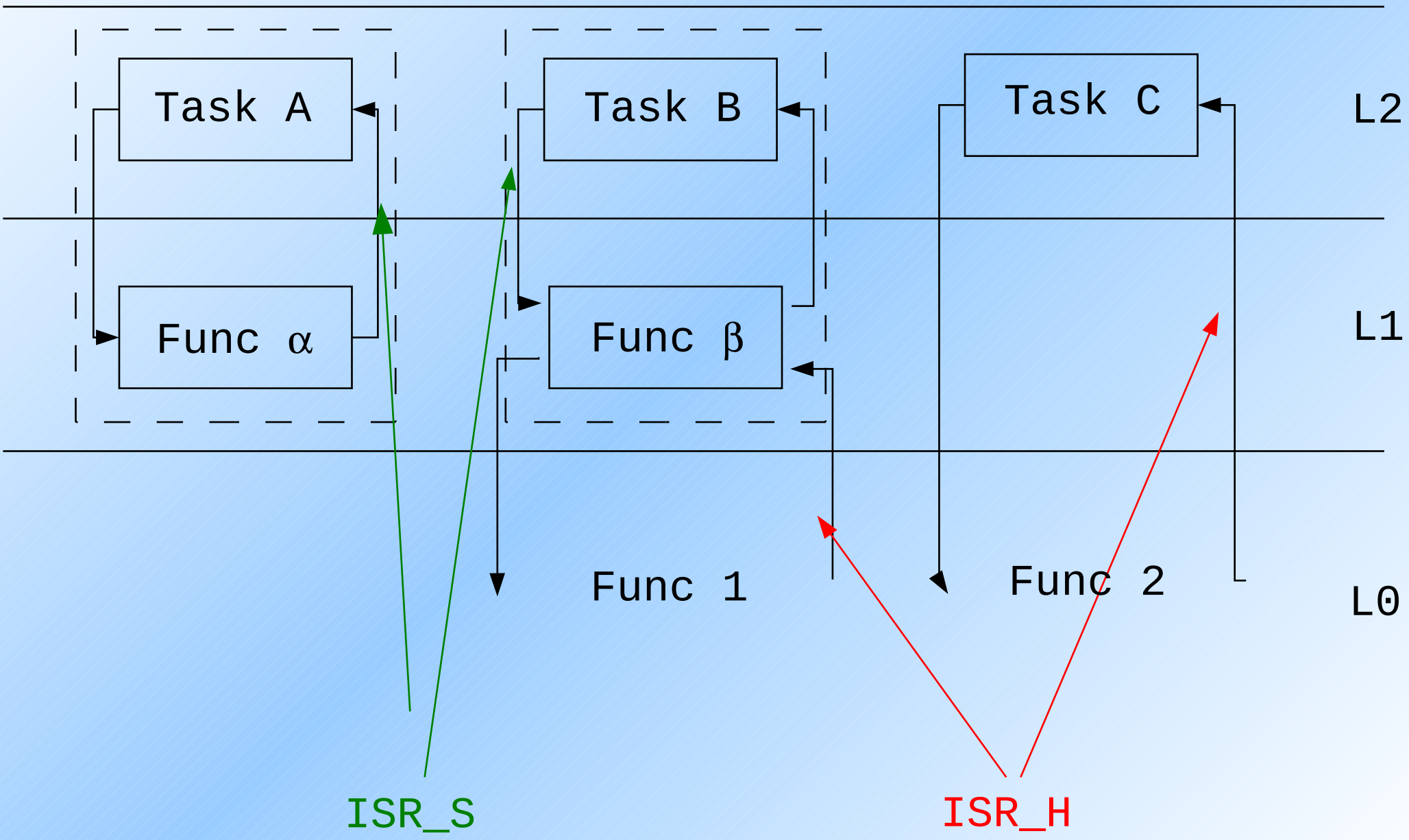
ISR = Interrupt Service Routine

Un passaggio di livello (level switch) avviene in uno dei seguenti casi:

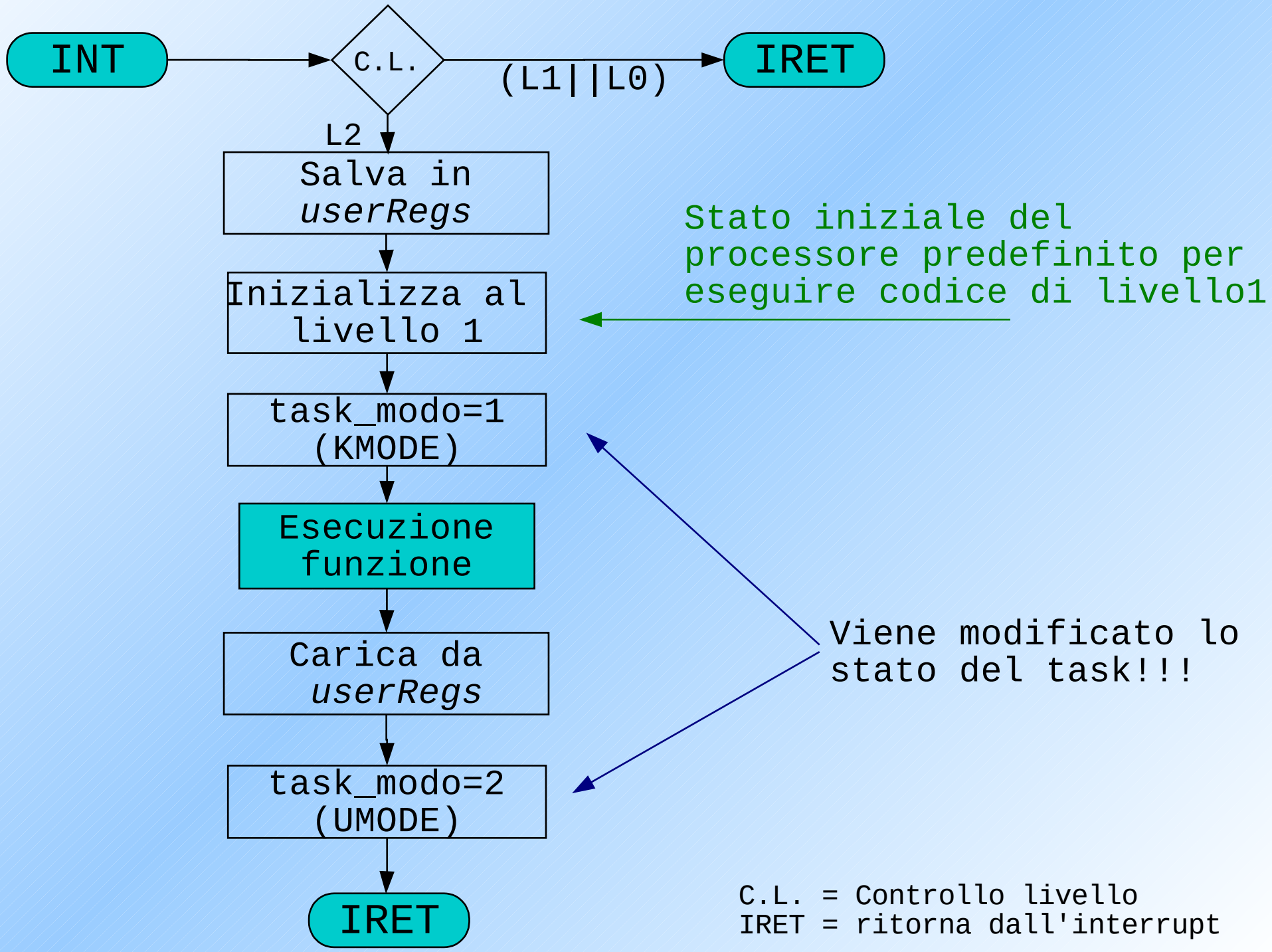
Durante un interrupt (quando viene chiamata una ISR)

Quando una ISR ritorna

ISR - Schema generico

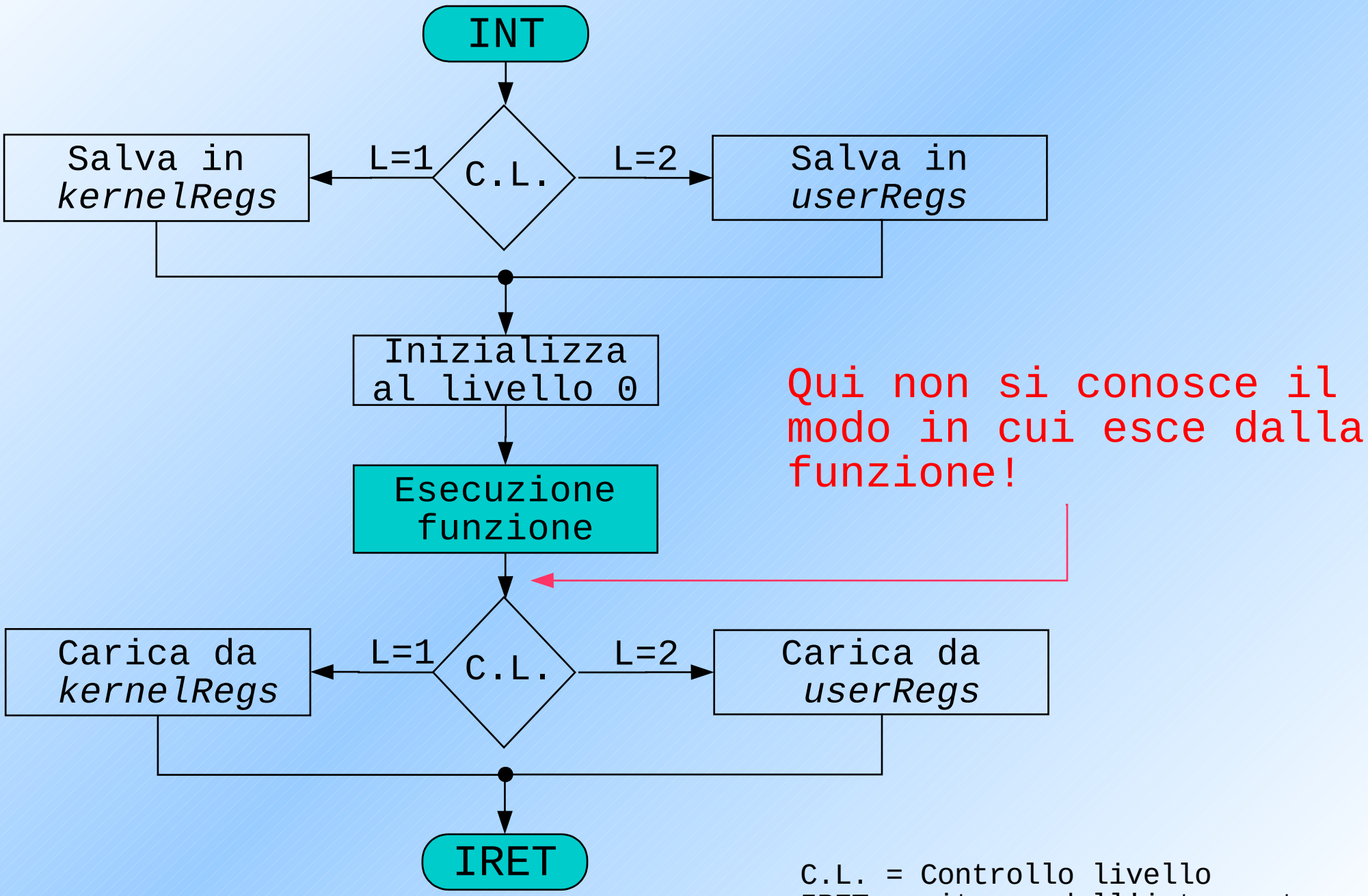


ISR_S - Schema a blocchi



C.L. = Controllo livello
IRET = ritorna dall'interrupt

ISR_H - Schema a blocchi



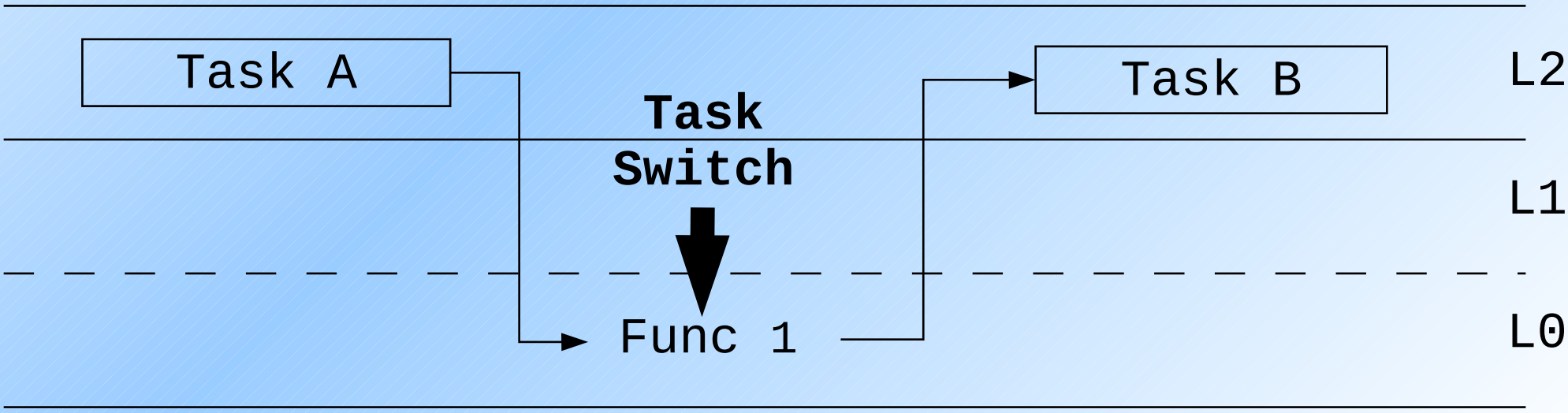
C.L. = Controllo livello
IRET = ritorna dall'interrupt

Task switch

Si vuole eseguire piu' task condividendo il tempo di CPU (Multitasking).

Come 'sostituire' il task corrente con un altro?

- Si passa ad un contesto indipendente dal task corrente, cioe' ad L0 (ISR_H)
- Si sostituisce il task corrente;
  viene salvata **struct** task_t current_task;
  si carica la struttura con i dati del nuovo task
- 'Ritorno' al nuovo task (return)

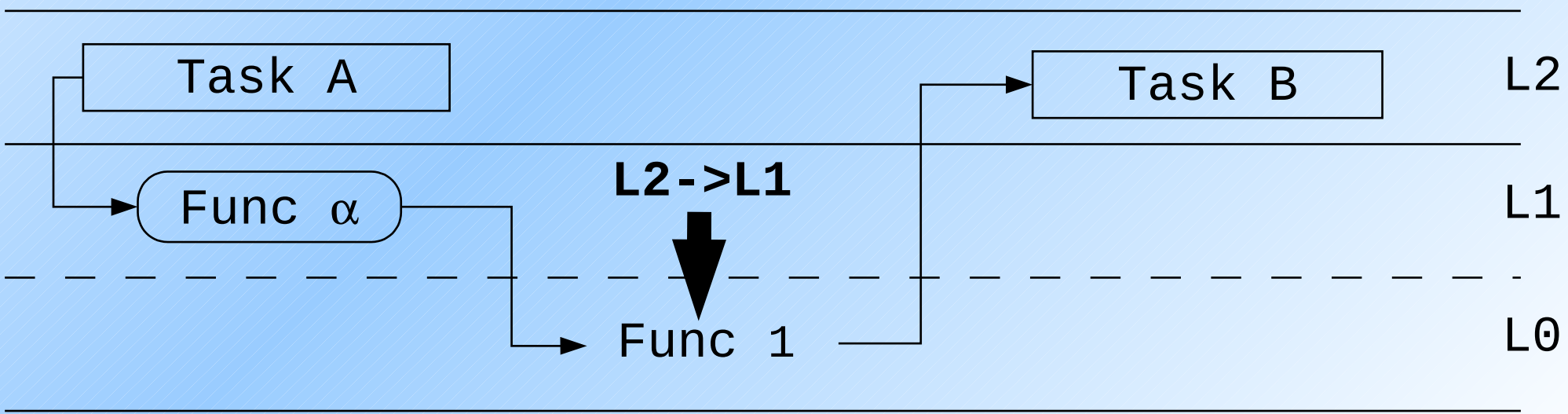


Task switch

Associamo ISR_H *Func1* all'INT del clock,
ma si potrebbe usarne uno qualunque.
(es. quello della tastiera, ma sarebbe un po' scomodo!!!)

Il Multitask non e' necessario; KernelCore puo' essere usato per implementare un SO monotask (DOS).

All'uscita da L0 controlliamo il livello a cui si ritorna (puo' essere cambiato).



Gestione del clock

Il kernel conta il tempo in *ticks* (1 tick = 10ms)

All'avvio la variabile globale *ticks* viene impostata a zero

Il clock e' costituito da una ISR_H associata ad un interrupt

L'ISR compie queste azioni:

Incrementa la variabile globale *ticks*

Scorre una lista di funzioni associate ad un periodo di tempo e ad un contatore

Gestione del clock (2)

```
struct time_table_t
    {
    time_job_t func;
    unsigned long  tickperiod;
    unsigned long  tickcount;
    unsigned int volte;
    };
```

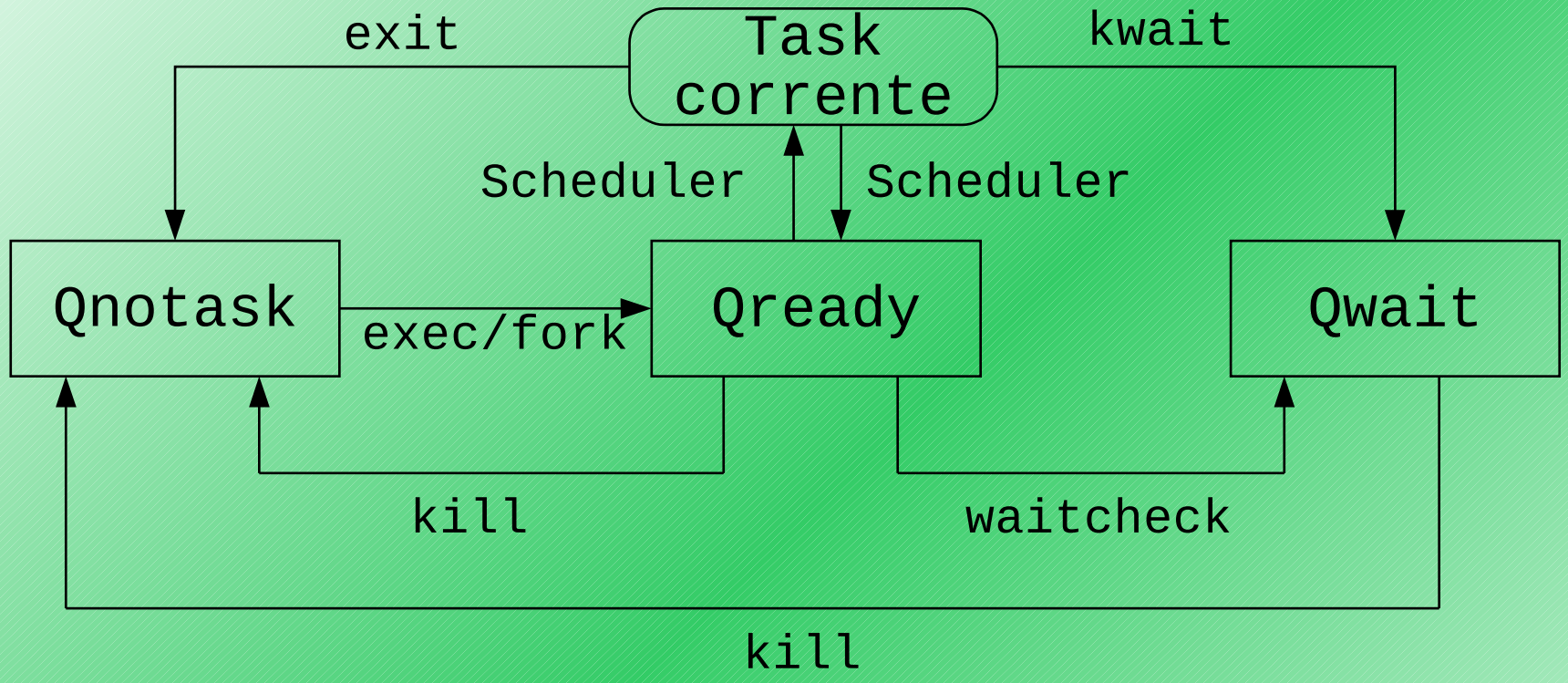
Quando *tickperiod = tickcount*
viene eseguita la funzione *func*

Possiamo mandare in esecuzione del codice
ogni *n* ticks

Queste funzioni sono chiamate time-job

PROBLEMA: il codice dei time-job viene eseguito a livello 0, con le conseguenze e le restrizioni gia' viste in precedenza

Scheduler




```
lista_task[MAXTASK]      : array di struct task_t
          Qnotask    : lista dei pid 'liberi'
          Qready     : coda di READY
          Qwait      : coda di WAIT

  Le code sono di pid(int) e non di strutture !
```

Scheduler (2)

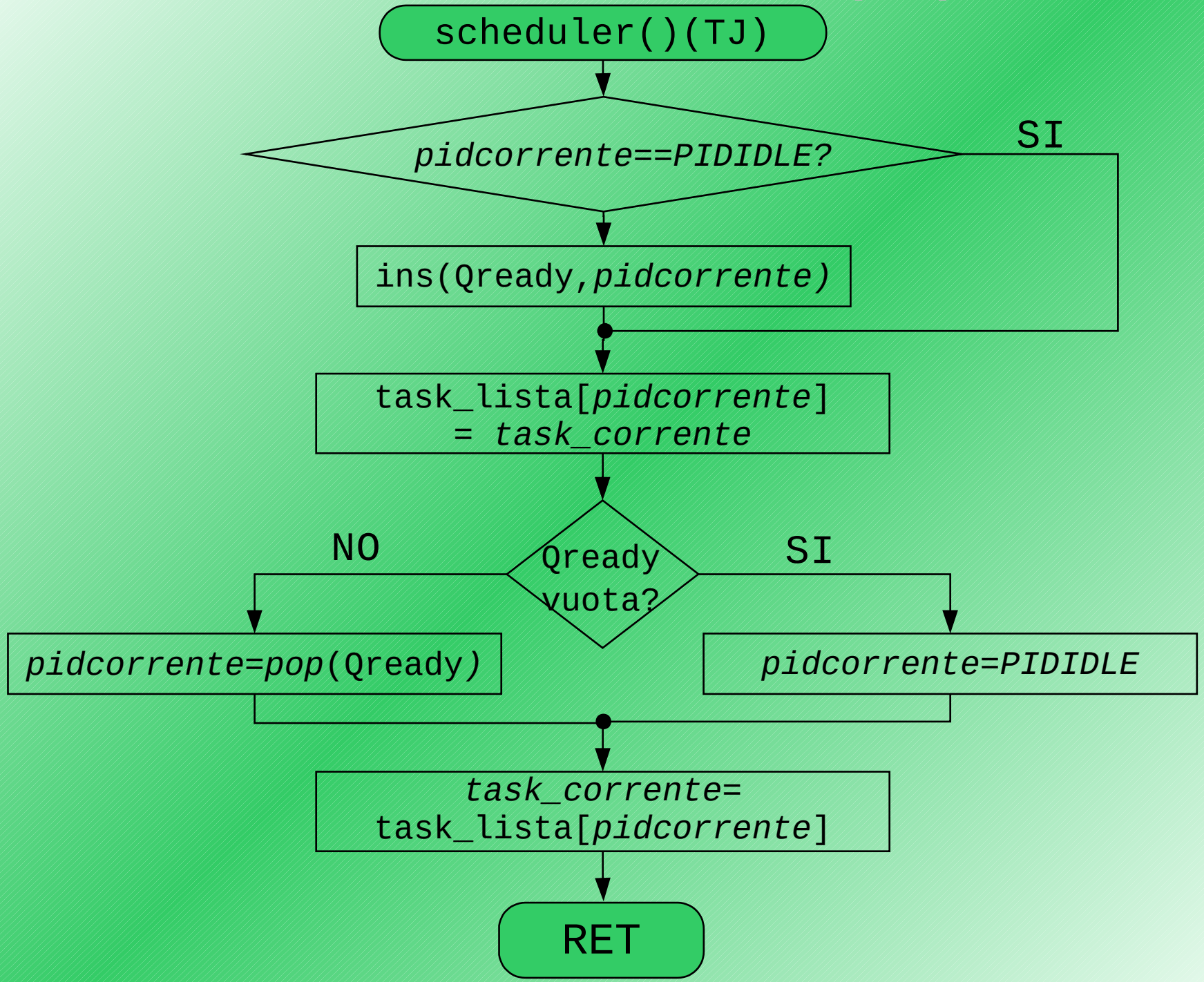
Attualmente e' implementata una Round Robin semplice con coda di wait (i processi in attesa non consumano tempo di processore).

Non fa parte del KCORE, quindi puo' essere sostituito per implementarne uno di tipo differente.

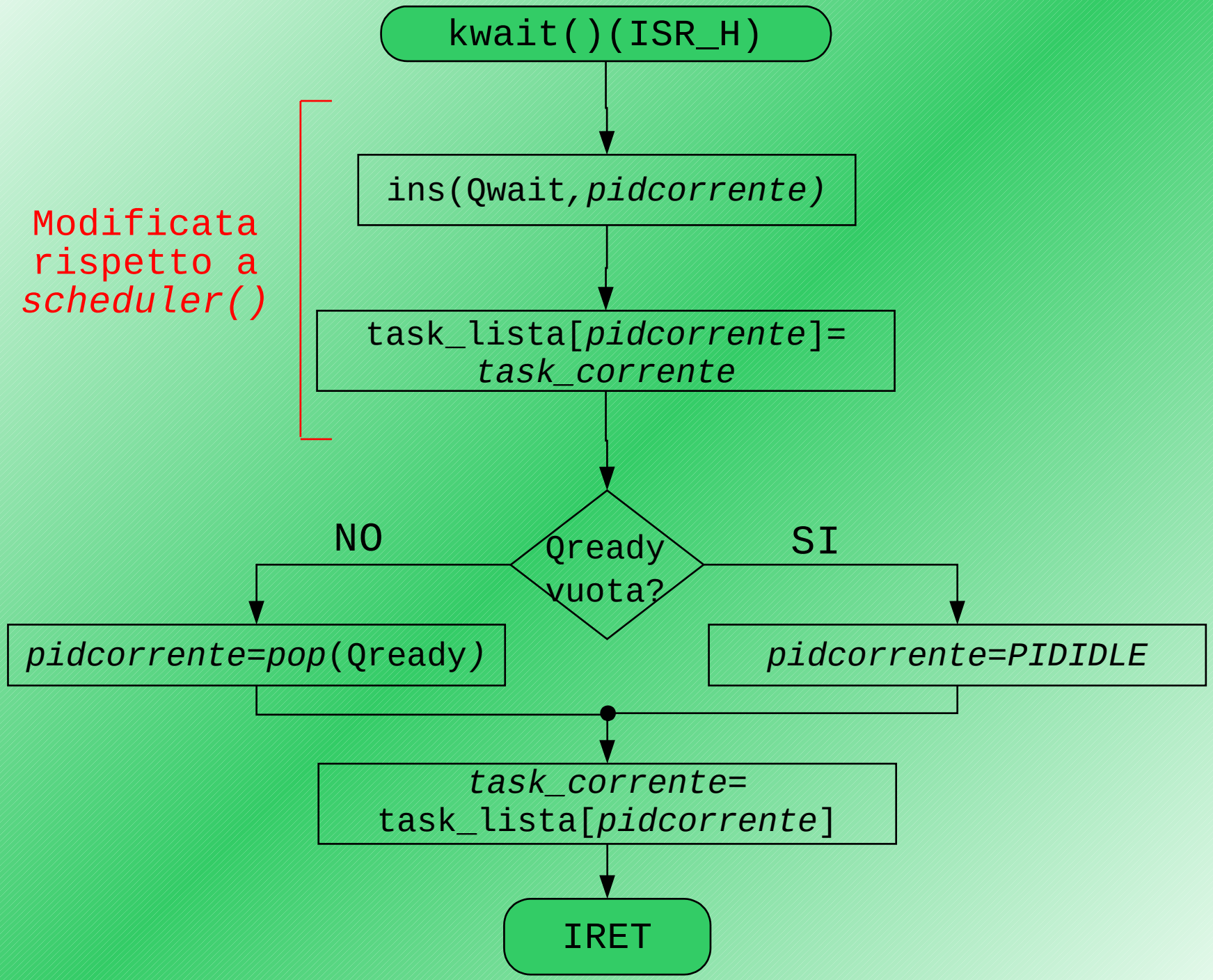
Formato da:
scheduler (TJ)
kwait (int 0x42, ISR_H)
kwait_check (TJ)

Sono state implementate delle syscall apposite
fork exec
exit wait
waitpid getpid
kill

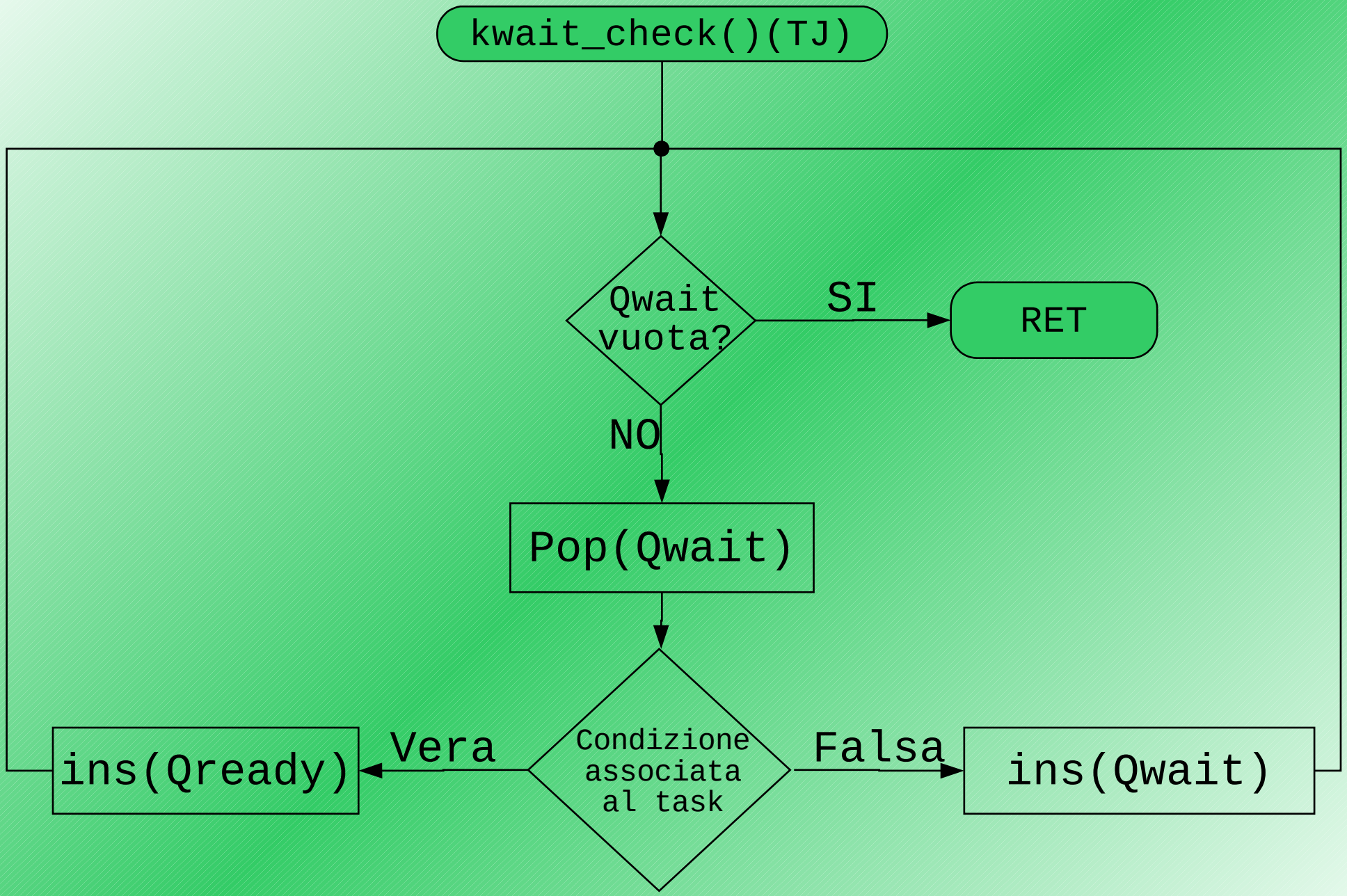
Scheduler (3)


scheduler()(TJ)
pidcorrente==PIDIDLE?
SI
ins(Qready,pidcorrente)
task_lista[pidcorrente]
= task_corrente
Qready
vuota?
NO
SI
pidcorrente=pop(Qready)
pidcorrente=PIDIDLE
task_corrente=
task_lista[pidcorrente]
RET

Scheduler (4)


kwait()(ISR_H)
ins(Qwait,pidcorrente)
task_lista[pidcorrente]=
task_corrente
Modificata
rispetto a
scheduler()
Qready
vuota?
NO
SI
pidcorrente=pop(Qready)
pidcorrente=PIDIDLE
task_corrente=
task_lista[pidcorrente]
IRET

Scheduler (5)


kwait_check()(TJ)
Qwait vuota?
SI
RET
NO
Pop(Qwait)
Condizione associata al task
Vera
ins(Qready)
Falsa
ins(Qwait)

Il Filesystem

Perche' una gestione generica del filesystem?

Ogni file e' identificato da una stringa standard

<VOLUME>:<PATH><FILE/DIR>

A monte, un sistema di controllo per l'accesso concorrente ai vari files

Operazioni fondamentali:

- Apertura
- Chiusura
- Lettura
- Scrittura
- Ricerca
- Creazione
- Rimozione

```
struct fsi_t
    {
        char nome[4];

        fsi_init_t init;

        fsi_open_t    open;
        fsi_close_t   close;
        fsi_seek_t    seek;
        fsi_read_t    read;
        fsi_write_t   write;
        fsi_touch_t   touch;
        fsi_remv_t    remv;
    };
```

Il Filesystem (2)

All'apertura di una directory otteniamo un file virtuale con il suo contenuto (equivalente all'uscita del comando *ls*)

Uno stesso file puo' essere aperto da un solo task alla volta e per non piu' di una volta

I task figli non ereditano nulla dai padri

Se un task termina senza aver chiuso i file aperti, ci pensa il sistema operativo a chiuderli!!!

NOTA: per adesso il meccanismo non e' perfetto! Non fa differenza tra accesso in lettura e accesso in scrittura

Il Filesystem (3)

Sono supportati piu' volumi contemporaneamente:

- KER: [Contiene informazioni sul kernel]
- CON: [le consoles (I/O)]
- FAT: [Fat 12 per il floppy]

CON: e FAT: sono collegati ai dispositivi di I/O ed e' questo l'unico modo che i task hanno per comunicare con l'esterno (e con l'utente)

Implementare un nuovo filesystem:
basta creare una nuova struttura di tipo fsi_t simile a quella vista nella slide precedente e scrivere le funzioni fondamentali

Il Filesystem (4)

Come funziona?

La prima parte del PATH serve per riconoscere il filesystem da usare

Si basa su una tabella che tiene conto di quale task opera su quale file.

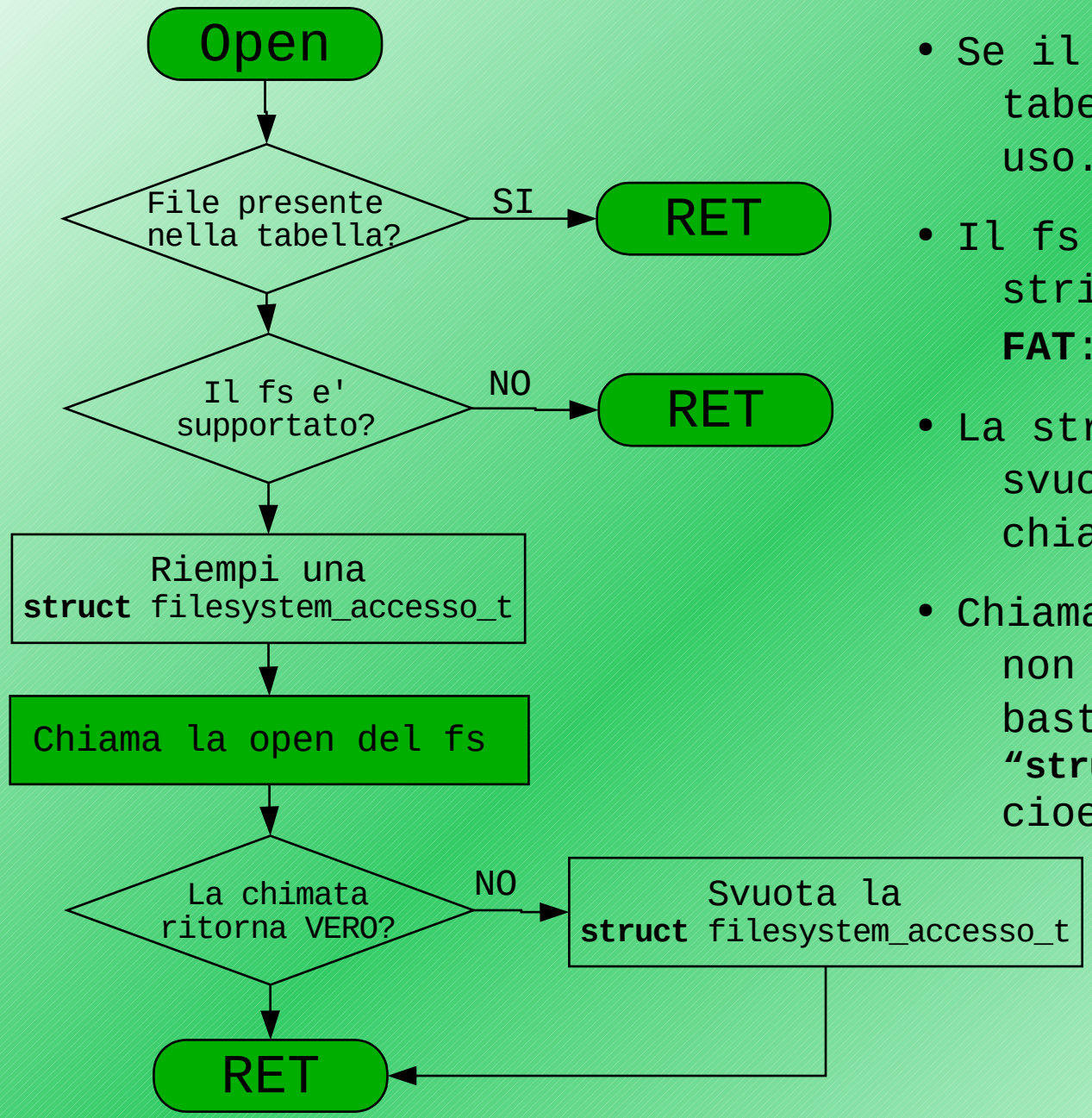
La tabella e' composta da '**struct** filesystem_accesso_t'

```
struct filesystem_accesso_t
    {
    char nomefile[KSTRNG_LUNG];
    int pid;
    int fs;
    void* session;
    };
```

- Nome del file su cui si opera (anche se non esiste!)
- Pid del processo che opera
- Puntatore al tipo indefinito (link ad informazioni specifiche della sessione)
- L'indice dell'array e' FD !

Il Filesystem (5)

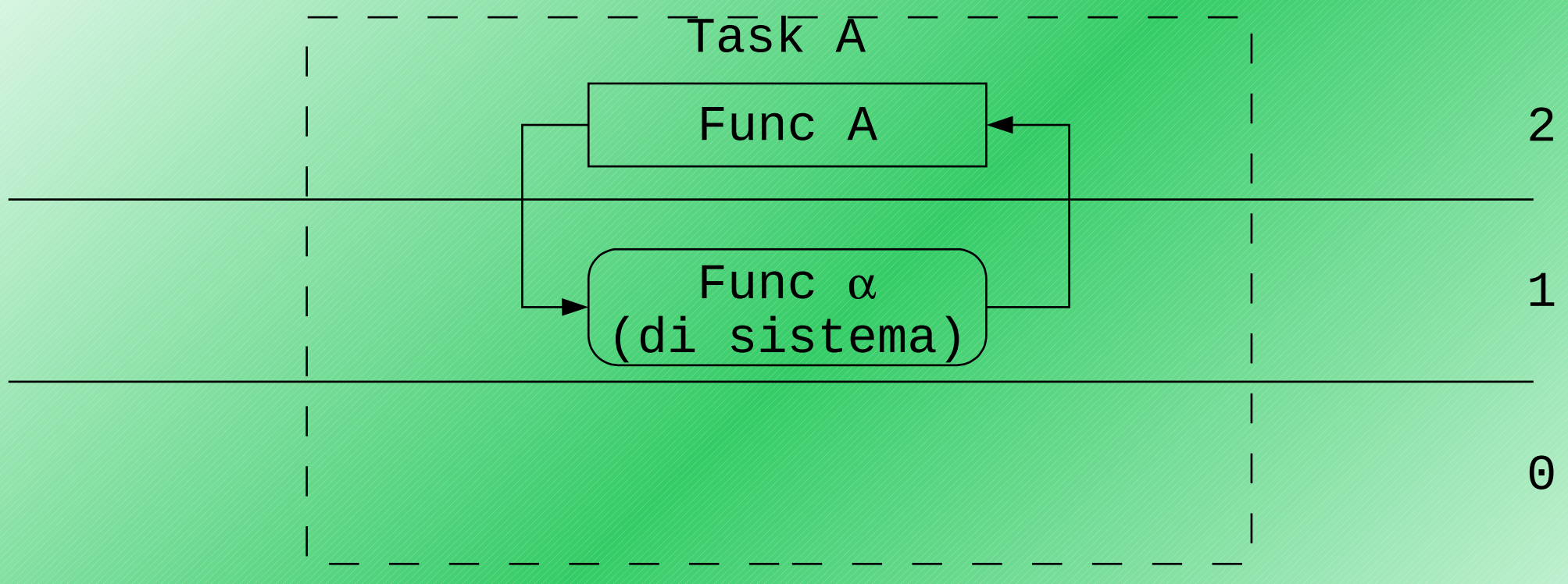
Facciamo un esempio : open



- Se il file e' presente nella tabella allora e' gia' in uso.
- Il fs viene riconoscito dalla stringa del file **FAT**:/BOOT/GRUB/MENU.LST
- La struttura non viene svuotata!(ci pensera' la chiamata close)
- Chiamate di read write seek non specificano il file; basta l'indice della “**struct** filesystem_accesso_t” cioe' il FD

Le syscall

Chiamate fatte da un task ad una funzione di sistema



Durante la syscall avviene il passaggio al livello 1

Si esegue la funzione richiesta dal task
(con privilegio kernel controllato e non arbitrario)

Si ritorna al livello 2

Le syscall (2)

Per passare i parametri da un livello all'altro si usa uno standard comune:

Registro	16 bit alti	16 bit bassi
eax	Gruppo di syscall	Numero syscall
ebx	Valore di ritorno (in uscita) / Parametro C (in ingresso)	
ecx	Parametro A	
edx	Parametro B	

Problema: per passare i parametri vengono utilizzati i registri, e questo rende le syscall dipendenti dall'architettura!!!

Sviluppi futuri

Indipendenza dalla piattaforma

- Devono essere raggruppate tutte le parti dipendenti dall'architettura.
- Il level switch non dovra' usare i registri per passare i parametri
- La MMU dovra' essere gestita a parte
- I parametri dovranno essere passati attraverso uno stack invece che attraverso i registri

Sviluppi futuri

Modularita'

- Caricare i moduli dinamicamente (usando GRUB per caricare i moduli fin dall'inizio).

- Aggiungere una interfaccia comune a tutti i devices.

- Rendere il S.O. utilizzabile anche su sistemi privi di MMU (tipo Intel 8086). Nei sistemi con MMU, questa potra' essere caricata come modulo (attenzione! Si perde la protezione della memoria. Ogni task vede tutta la memoria disponibile).

- Sfruttare il punto precedente per implementare il paging della memoria

Sviluppi futuri

Modularita'

Puo' essere ottenuta tramite l'uso del formato ELF rilocabile (ELF = Executable & Linkable Format).

La rilocazione permette di far funzionare il sistema su macchine prive di MMU, e il linking permette di caricare in posizioni arbitrarie di memoria le parti (moduli) del kernel.

Un ulteriore vantaggio: lo sviluppo del sistema puo' essere distribuito tra piu' persone; ognuno realizza un modulo che verra' unito dinamicamente agli altri.

Alcuni link utili

http://osdev.org
[Sviluppo sistemi operativi]
http://osdev.net
[Sviluppo sistemi operativi]
http://www.intel.com
[Manualistica]
http://www.nondot.net/sabre/os/articles
[The Operative Sistem resource center]
http://www.osdever.net
[Bona Fide O.S. Development]
http://www.bsdg.org
[Boise Software Developer Group]

http://unitutor.unisi.it/~sulion
[Versione scaricabile del nostro S.O. con documentazione]

Domande???

Ringraziamenti

Desideriamo ringraziare le seguenti persone:
Andrea Righi (per alcune 'consulenze' tecniche),
METTERE NOME 'L'omino del floppy' METTERE COGNOME
(per aver scritto 30 anni fa driver per floppy usati da tutti),
la Ceres e la Lowenbrau (che ci hanno alleviato la sete durante la
creazione di queste slides), il succo ai frutti di bosco (per lo stesso
motivo dei precedenti), Loris - o meglio L'Oris - (che ci fa i panini quasi
tutti i giorni), Oreste (il grande Oreste!), Feller (per aver fatto da βtester
per il giochino del Master Mind), Dalamar (che ci ha sempre dato fiducia dicendoci
che non eravamo capaci), R0y_Jones (per l'avatar fantastico che ha sul forum), Oiccic
(che tutte le mattine ci veniva a trovare in laboratorio infamandoci), Lazzaro (per il suo
amore per gli animali), Gigi non lo ringraziamo perche' non ci ha fatto ancora vedere il portatile
che si e' comprato, G3 (perche' invece il portatile ce lo ha pure lasciato... chissa' se ha apprezzato
lo sfondo che gli abbiamo messo?), tutte le belle bimbe che sono venute a trovarci in laboratorio (questo
si che e' supporto morale!), l'ideatore dei giochini (in Flash) del pinguino, il creatore del gioco del trial,
Mr.George Lucas (per avermi ispirato questo bellissimo layout per i ringraziamenti), Leo Ortolani (per aver creato
Rat-Man, volevo quasi iniziare questa slide con le stesse parole che usa lui per l'inizio di Star Rats, ma poi sembrava
di copiare!), i Pink Floyd , i Queen, Simon & Garfunkel e i cantanti delle mitiche sigle dei cartoni animati anni ottanta
(per la musica che ascoltavamo mentre scrivevamo le slides), l'Enel (senza la cui corrente non avremmo potuto usare i
computere senza la quale non avremmo imparato nuovi improperi quando la corrente saltava, buttando via il lavoro non
salvato ogni dieciminuti!!!), la stampante del laboratorio (che ci ha stampato la stupenda documentazione del progetto),
le sedie del laboratorio(che ci hanno affinato i riflessi visto che tendono a cadere a pezzi quando ti ci siedi su!), i
mouse rotti (che hanno temprato lanostra pazienza) e le tastiere senza un piedino, il temperamatite (che serviva appunto
da piedino della tastiera di prima) e un bel grazie anche a tutti quelli che ci siamo scordati di menzionare . Parole e
tanti puntini inutili per allineare bene.....

FINE