Parte xxxiv

Studio per un sistema a 16 bit

«

Introduzione a os16 3031

- Organizzazione 3032
- Le directory 3034
- La struttura degli eseguibili 3035
- Caricamento del kernel 3037
- Informazioni diagnostiche 3037
- Tabelle 3037
- Guida di stile 3038
- Tipi derivati speciali 3043

Caricamento ed esecuzione del kernel 3047

- Dal file su disco alla copia in memoria 3047
- File «kernel/main/crt0.s» 3050
- File «kernel/main.h» e «kernel/main/*» 3055

Funzioni interne legate all'hardware 3063

- Libreria: «lib/sys/os16.h» e «lib/sys/os16/...» 3064
- Funzioni di basso livello dei file «kernel/ibm_i86/*» 3066
- Gestione della console 3072
- Gestione dei dischi 3075

Gestione della memoria 3081

- File «kernel/memory.h» e «kernel/memory/...» 3082

Scansione della mappa di memoria 3086
Gestione dei terminali virtuali 3089
Dispositivi 3093
File «lib/sys/os16.h» e directory «lib/sys/os16/» 3093
File «kernel/devices.h» e «kernel/devices/...» 3094
Numero primario e numero secondario 3098
Dispositivi previsti 3099
Gestione del file system 3105
File «kernel/fs/sb_...» 3106
File «kernel/fs/zone_...» 3111
File «kernel/fs/inode_...» 3113
Fasi dell'innesto di un file system 3124
File «kernel/fs/file_...» 3126
Descrittori di file 3129
File «kernel/fs/path_...» 3131
File «kernel/fs/fd_...» 3136
Gestione dei processi 3141
File «kernel/proc/_isr.s» e «kernel/proc/_ivt_load.s» 3142
La tabella dei processi 3154
Chiamate di sistema 3161
File «kernel/proc/...» 3162
Caricamento ed esecuzione delle applicazioni 3171
Caricamento in memoria 3171
Il codice iniziale dell'applicativo 3174

Introduzione a os16

Organizzazione 3032
Le directory 3034
La struttura degli eseguibili 3035
Caricamento del kernel 3037
Informazioni diagnostiche 3037
Tabelle 3037
Guida di stile 3038
Tipi derivati speciali 3043

`addr_t` 3043 `diag.h` 3037 `directory_t` 3043 `dsk_chs_t` 3043 `dsk_t` 3043 `fd_t` 3043 `file_t` 3043 `inode_t` 3043 `memory_t` 3043 `offset_t` 3043 `sb_t` 3043 `segment_t` 3043 `tty_t` 3043 `zno_t` 3043

os16 è uno studio che applica qualche rudimento relativo ai sistemi operativi, basandosi sull'architettura x86-16 del vecchio IBM PC, utilizzando come strumenti di sviluppo Bcc, As86 e Ld86 (oltre a GNU GCC per controllare meglio la sintassi del codice C), su un sistema GNU/Linux. Il risultato non è un sistema operativo utilizzabile, ma una struttura su cui poter fare esperimenti e di cui è possibile mostrare (in termini tipografici) ed eventualmente descrivere ogni riga di codice.

os16 contiene uno schedulatore banale e molto limitato, un'organizzazione dei processi ad albero e una funzionalità limitata di amministrazione dei segnali, una gestione del file system Minix 1

(ma di unità intere, senza partizioni), una shell banale e qualche programma di servizio di esempio.

Per poter giungere rapidamente a un risultato e comunque per semplificare il codice, os16 utilizza le funzioni del BIOS tradizionale, le quali hanno lo svantaggio di impegnare in modo esclusivo l'elaboratore nel momento del loro funzionamento (dato che non possono essere rientranti). È noto che un sistema operativo multiprogrammato dignitoso non può avvalersi di tali funzionalità; pertanto, se si vuole studiare os16, non va dimenticato questo principio, benché qui sia stato trascurato.

Il kernel di os16 è monolitico, nel senso che incorpora tutte le proprie funzioni in un solo programma. Purtroppo, la dimensione del codice del kernel (e di qualunque altro processo di os16) non può superare i 64 Kibyte, ma la sua dimensione è già molto vicina a tale valore. Pertanto, non è possibile aggiungere funzionalità a questo sistema. Si può osservare che anche ELKS (*http://elks.sourceforge.net/*) soffra dello stesso limite e, d'altra parte, si può apprezzare come Minix 2 (*http://minix1.woodhull.com/*) riesca a superarlo attraverso un'organizzazione a micro-kernel.

Organizzazione

«

Tutti i file di os16 dovrebbero essere disponibili a partire da *allegati/os16*. In particolare i file '`floppy.a`' e '`floppy.b`' sono le immagini di due dischetti da 1440 Kibyte, contenenti un file system Minix 1: il primo predisposto attraverso Bootblocks (sezione u0.5)

per l'avvio di un kernel denominato 'boot'; il secondo usato per essere innestato nella directory '/usr/' del primo.

Gli script preparati per lo sviluppo di os16 prevedono che i file-immagine dei dischetti vadano innestati nelle directory '/mnt/os16.a/' e '/mnt/os16.b/'. Pertanto, se si ricompila os16, tali directory vanno predisposte (oppure vanno modificati gli script con l'organizzazione che si preferisce attuare).

Per la verifica del funzionamento del sistema, è previsto l'uso equivalente di Bochs o di Qemu. Per questo scopo sono disponibili gli script 'bochs' e 'qemu' (rispettivamente i listati i160.1.1 e i160.1.2), con le opzioni necessarie a operare correttamente.

Per la compilazione del lavoro si usa lo script 'makeit' (listato i160.1.3), il quale ricrea ogni volta i file-make, basandosi sui file presenti effettivamente nelle varie directory previste. Questo script, alla fine della compilazione, copia il kernel nel file-immagine del primo dischetto (purché risulti innestato come previsto nella directory '/mnt/os16.a/') e con esso copia anche gli applicativi principali, mentre il resto viene copiato nel secondo.

Nello script 'makeit', la variabile di ambiente '**TAB**' deve contenere esattamente il carattere di tabulazione (<*HT*>), corrispondente al codice 09_{16}. Il file che viene distribuito contiene invece l'assegnamento di uno spazio puro e semplice, che va modificato a mano, sostituendolo con tale codice. La riga da modificare è quella in cui la variabile viene dichiarata:

```
local TAB=" "
```

Va osservato che il lavoro si basa su un file system Minix 1 (sezione 68.7) perché è molto semplice, ma soprattutto, la prima versione è quella che può essere utilizzata facilmente in un sistema operativo GNU/Linux (sul quale avviene lo sviluppo di os16). È bene sottolineare che si tratta della versione con nomi da 14 caratteri, ovvero quella tradizionale del sistema operativo Minix, mentre nei sistemi GNU/Linux, la creazione predefinita di un file system del genere produce una versione particolare, con nomi da 30 caratteri.

Le directory

«

Gli script descritti nella sezione precedente, si trovano all'inizio della gerarchia prevista per os16. Le directory successive dividono in modo molto semplice le varie componenti per la compilazione:

Directory	Contenuto
‘`applic/`’	File delle applicazioni da usare con os16.
‘`kernel/`’	File per la realizzazione del kernel, inclusi i file di intestazione specifici.
‘`lib/`’	File di intestazione generali, file della libreria C per le applicazioni e, per quanto possibile, anche per il kernel.
‘`ported/`’	Applicativi di altri autori, adattati per os16.
‘`skel/`’	Scheletro del file system complessivo, con i file di configurazione e le pagine di manuale.

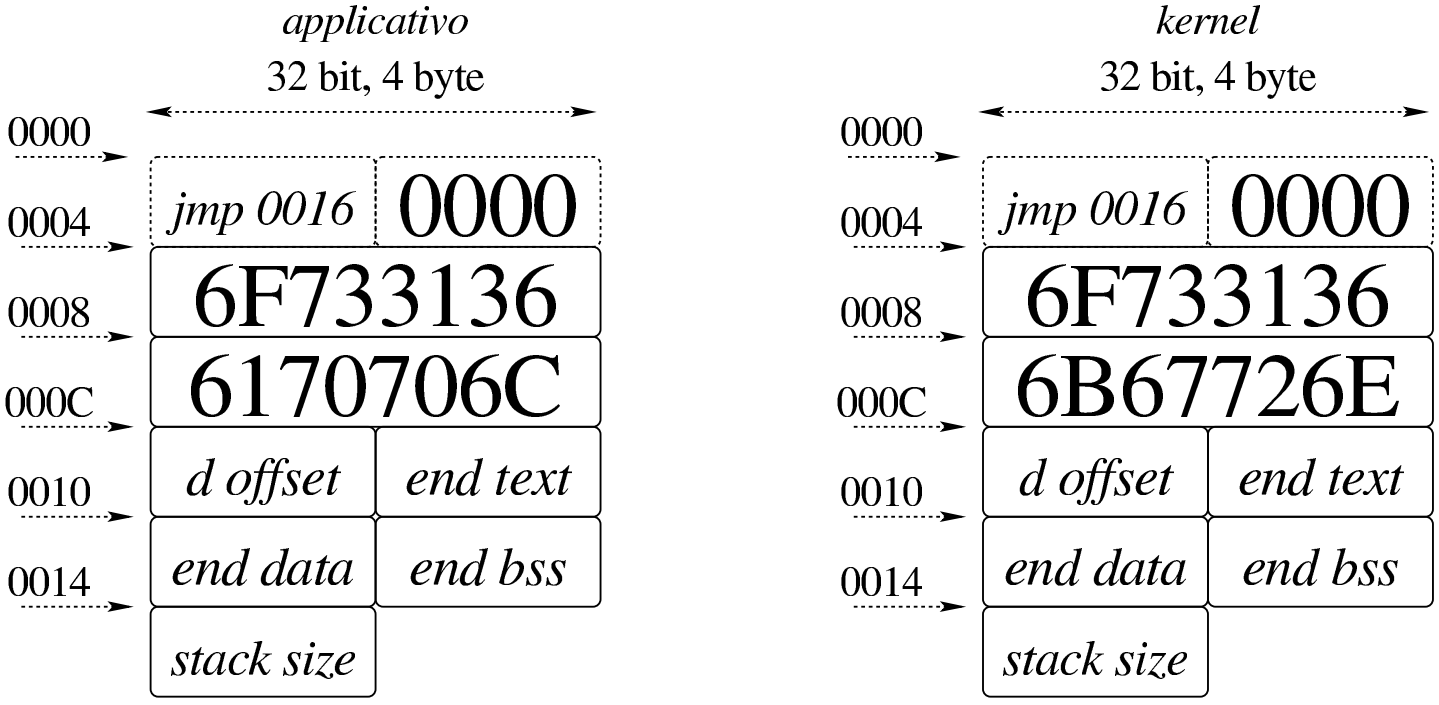
La libreria C non è completa, limitandosi a contenere ciò che serve per lo stato di avanzamento attuale del lavoro. Si osservi che nella directory ‘`lib/bcc/`’ si collocano file contenenti una libreria di funzioni in linguaggio assemblatore, necessaria al compilatore Bcc per

compiere il proprio lavoro correttamente con valori da 32 bit. Tale libreria è scritta dall'autore originale di Bcc, Bruce Evans, e viene inclusa in modo da garantire che la compilazione non richieda alcun file estraneo.

La struttura degli eseguibili

Nell'ottica della massima semplicità, gli eseguibili di os16 hanno un'intestazione propria, schematizzata dalla figura successiva. Tale intestazione viene ottenuta attraverso il file '`crt0.s`', che è comunque differente se si tratta di un applicativo o del kernel.

Figura u142.2. Struttura iniziale dei file eseguibili di os16.



Nella figura si mettono a confronto la parte iniziale dell'eseguibile di un applicativo con quella del kernel di os16. Nei primi quattro byte c'è un'istruzione di salto al codice che si trova subito dopo l'intestazione, quindi appare un'impronta di riconoscimento che occupa quattro byte. Tale impronta è la rappresentazione esadecimale della stringa «os16». Successivamente appare un'altra impronta, con cui

si distingue se si tratta di un applicativo o del kernel; si tratta in pratica della sequenza di «appl», oppure di «kern». Tuttavia, a causa dell'inversione dell'ordine dei byte, in pratica, se si visualizza il file binario si legge «61so», «lppa» e «nrek».

Dopo l'impronta di riconoscimento si trovano, rispettivamente, lo scostamento del segmento dati, espresso in multipli di 16 (in pratica, 1234_{16} rappresenterebbe uno scostamento di 12340_{16} byte, rispetto all'inizio del codice), gli indirizzi conclusivi dell'area del codice, dei dati inizializzati e di quelli non inizializzati. Alla fine viene indicata la dimensione richiesta per la pila dei dati. Ciò che appare dopo è il codice del programma.

Il kernel e gli applicativi di os16 sono compilati in modo da rendere indipendenti l'area del codice rispetto a quella dei dati (si dice che hanno aree «I&D separate»).

Nel caso del kernel, le informazioni successive alle impronte di riconoscimento non vengono utilizzate, perché il kernel viene collocato in uno spazio preciso in memoria: l'area dati va a trovarsi dall'indirizzo efficace 00500_{16} fino a $104FF_{16}$ incluso, mentre l'area del codice inizia da 10500_{16} fino alla fine.

Per fare in modo che il proprio sistema GNU possa riconoscere correttamente questi file, si può modificare la configurazione del file '`/etc/magic`', aggiungendo le righe seguenti:

```
4       quad        0x6B65726E6F733136      os16 kernel
4       quad        0x6170706C6F733136      os16 application
```

Caricamento del kernel

«

Il kernel è preparato per trovarsi inizialmente in memoria, tale e quale al file da cui viene caricato, a partire dall'indirizzo efficace 10000_{16}, così come avviene quando si utilizza Bootblocks (a cui si è già accennato nel capitolo). Successivamente il kernel stesso si sposta, copiandosi inizialmente a partire dall'indirizzo efficace 30000_{16}, quindi suddividendosi e mettendo la propria area dati a partire dall'indirizzo 00500_{16} e l'area codice da 10500_{16} (lo spazio di memoria che va da 00000_{16} a $004FF_{16}$ incluso, non può essere utilizzato, perché contiene la tabella IVT e l'area BDA, secondo l'architettura degli elaboratori IBM PC tradizionali).

Informazioni diagnostiche

«

Nel codice del kernel vengono usate, in varie occasioni, delle funzioni che hanno lo scopo di visualizzare delle informazioni diagnostiche. Queste funzioni sono raccolte in file della directory '`kernel/diag/`', a cui si abbina il file di intestazione '`kernel/diag.h`' (listato u0.3 e successivi). In generale, in questa documentazione, non viene dato molto spazio alla descrizione di queste funzioni, perché hanno un ruolo marginale e sono fatte per essere modificate in base alle esigenze di verifica del momento.

Tabelle

«

Nel codice del kernel si utilizzano spesso delle informazioni organizzate in memoria in forma di tabella. Si tratta precisamente di array, le cui celle sono costituite generalmente da variabili strutturate. Queste tabelle, ovvero gli array che le rappresentano, sono dichiarate

come variabili pubbliche; tuttavia, per facilitare l'accesso ai rispettivi elementi e per uniformità di comportamento, viene abbinata loro una funzione, con un nome terminante per '…**`_reference()`**', con cui si ottiene il puntatore a un certo elemento della tabella, fornendo gli argomenti appropriati. Per esempio, la tabella degli inode in corso di utilizzazione viene dichiarata così nel file '`kernel/inode/inode_table.c`':

```
inode_t inode_table[INODE_MAX_SLOTS];
```

Successivamente, la funzione ***inode_reference()*** offre il puntatore a un certo inode:

```
inode_t *inode_reference (dev_t device, ino_t ino);
```

Guida di stile

«

Per cercare di dare un po' di uniformità al codice del kernel e a quello della libreria, dove possibile, i nomi delle variabili seguono una certa logica, riassunta dalla tabella successiva.

Tipo	Nome	Utilizzo
`inode_t *`	`inode` `inode_…`	Puntatore a un inode (puntatore a un elemento della tabella di inode).
`ino_t`	`ino` `ino_…`	Numero di inode, nell'ambito di un certo super blocco (ammesso che sia abbinato effettivamente a un dispositivo).

Tipo	Nome	Utilizzo
`int`	`fdn` `fdn_…`	Numero del descrittore di un file (indice all'interno della tabella dei descrittori).
`fd_t *`	`fd` `fd_…`	Puntatore a un descrittore di file (puntatore a un elemento della tabella di descrittori).
`int`	`fno` `fno_…`	Numero del file di sistema (indice all'interno della tabella dei file di sistema).
`zno_t`	`zone` `zone_…`	Numero assoluto di una «zona» del file system Minix.
`zno_t`	`fzone` `fzone_…`	Numero relativo di una «zona» del file system Minix. In questo caso, il numero della zona è relativo al file, dove la prima zona del file ha il numero zero.

`off_t`	`offset`
	`offset_…`
	`off_…`
		Scostamento, secondo il significato del tipo derivato '**`off_t`**'.

`size_t` `ssize_t`	`size` `size_…`	Dimensione, secondo il significato dei tipi derivati '**`size_t`**' o '**`ssize_t`**'.

Tipo	Nome	Utilizzo

`size_t`
 	
`ssize_t`
	`count`
	
	`count_…`
		Quantità, quando il tipo '**`size_t`**' è appropriato.

`blkcnt_t`	`blkcnt`
	
	`blkcnt_…`
		Quantità espressa in blocchi del file system (in questo caso, trattandosi di un file system Minix 1, si intendono zone).

`blksize_t`	`blksize`
	
	`blksize_…`
		Dimensione del blocco del file system, espressa in byte (in questo caso, trattandosi di un file system Minix 1, si intende la dimensione della zona).

`int`	`fno`
	
	`fno_…`
		Numero di file system.

`int`	`oflags`
	
	`oflags_…`
		Opzioni relative all'apertura di un file, annotate nella tabella dei file di sistema: indicatori di sistema.

`int`	`status`
	
	`status_…`
		Valore intero restituito da una funzione, quando la risposta contiene solo l'indicazione di un successo o di un insuccesso.

Tipo	Nome	Utilizzo
`void *`	`pstatus`	Puntatore restituito da una funzione, quando interessa sapere solo se si tratta di un esito valido.
`char *`	`path` `path_…`	Percorso del file system.
`dev_t`	`device` `device_…`	Numero di dispositivo, contenente sia il numero primario, sia quello secondario (*major*, *minor*).
`int`	`n` `n_…`	Dimensione di qualcosa, di tipo '**`int`**'.
`char *`	`string` `string_…`	Area di memoria da considerare come stringa.
`void *`	`buffer` `buffer_…`	Area di memoria destinata ad accogliere un'informazione di tipo imprecisato.
`int`	`n` `n_…`	Dimensione o quantità di qualcosa, espressa attraverso il tipo '**`int`**'.
`int`	`c` `c_…`	Un carattere senza segno trasformato nel tipo '**`int`**'.

Tipo	Nome	Utilizzo
`struct stat`	`st` `st_…`	Variabile strutturata usata per rappresentare lo stato di un file, secondo il tipo '**`struct stat`**'.
`FILE *`	`fp` `fp_…`	Puntatore che rappresenta un flusso di file.
`DIR *`	`dp` `dp_…`	Puntatore che rappresenta un flusso relativo a una directory.
`struct dirent`	`dir` `dir_…`	Variabile strutturata contenente le informazioni su una voce di una directory.
`struct password`	`pws` `pws_…`	Variabile strutturata contenente le informazioni di una voce del file '`/etc/passwd`'.
`struct tm`	`tms` `tms_…`	Variabile strutturata contenente le componenti di un orario.
`struct tm *`	`timeptr` `timeptr_…`	Puntatore a una variabile strutturata contenente le componenti di un orario.

Tipi derivati speciali

«

Nel codice del kernel si usano dei tipi derivati speciali, riassunti nella tabella successiva.

File di intestazione	Tipo speciale	Descrizione
‘`kernel/memory.h`’	`addr_t`	Variabile scalare, in grado di rappresentare un indirizzo efficace di memoria (un indirizzo che vada da 00000_{16} a $FFFFF_{16}$).
‘`kernel/memory.h`’	`segment_t`	Variabile scalare, a 16 bit, usata per rappresentare il valore di un registro di segmento.
‘`kernel/memory.h`’	`offset_t`	Variabile scalare, a 16 bit, usata per rappresentare lo scostamento di memoria, a partire dall’inizio di un segmento. Questo tipo di variabile non va confuso con il tipo ‘**`off_t`**’, il quale è un valore con segno e di rango maggiore rispetto a questo ‘**`offset_t`**’.
‘`kernel/memory.h`’	`memory_t`	Variabile strutturata, adatta a contenere tutte le coordinate utili a individuare una certa area di memoria, secondo l’architettura prevista da os16.

File di intestazione	Tipo speciale	Descrizione
‘kernel/tty.h’	tty_t	Variabile strutturata, adatta a contenere le informazioni e lo stato di un terminale.
‘kernel/ibm_i86.h’	dsk_t	Variabile strutturata, adatta a contenere le informazioni hardware di una certa unità di memorizzazione.
‘kernel/ibm_i86.h’	dsk_chs_t	Variabile strutturata, adatta a contenere le coordinate di un certo settore (cilindro, testina e settore) in un’unità di memorizzazione.
‘kernel/fs.h’	zno_t	Variabile scalare, per rappresentare un numero di una zona, secondo la terminologia del file system Minix.
‘kernel/fs.h’	sb_t	Variabile strutturata, adatta a contenere tutte le informazioni di un super blocco, relativo a un dispositivo di memorizzazione innestato.
‘kernel/fs.h’	inode_t	Variabile strutturata, adatta a contenere tutte le informazioni di un inode aperto nel sistema.
‘kernel/fs.h’	file_t	Variabile strutturata, adatta a contenere i dati di un file di sistema.

File di intestazione	Tipo speciale	Descrizione
‘kernel/fs.h’	fd_t	Variabile strutturata, adatta a contenere i dati di un descrittore di file, ovvero del file di un certo processo elaborativo.
‘kernel/fs.h’	directory_t	Variabile strutturata, adatta a contenere una voce di una directory.

Caricamento ed esecuzione del kernel

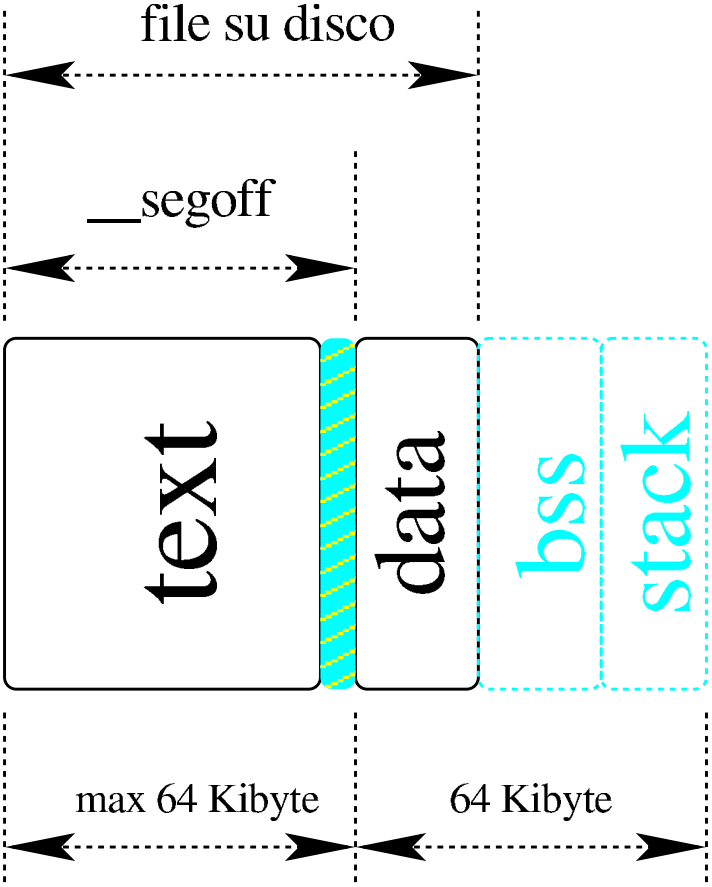
Dal file su disco alla copia in memoria 3047
File «kernel/main/crt0.s» 3050
File «kernel/main.h» e «kernel/main/*» 3055

`crt0.s` 3050 `main()` 3055

Il kernel di os16 (ma così vale anche per gli applicativi) viene compilato senza un'intestazione predefinita, pertanto questa viene costruita nel primo file: '`crt0.s`'. Questo file ha lo scopo di eseguire la funzione ***main()*** del kernel, in cui si sintetizza il funzionamento dello stesso.

Dal file su disco alla copia in memoria

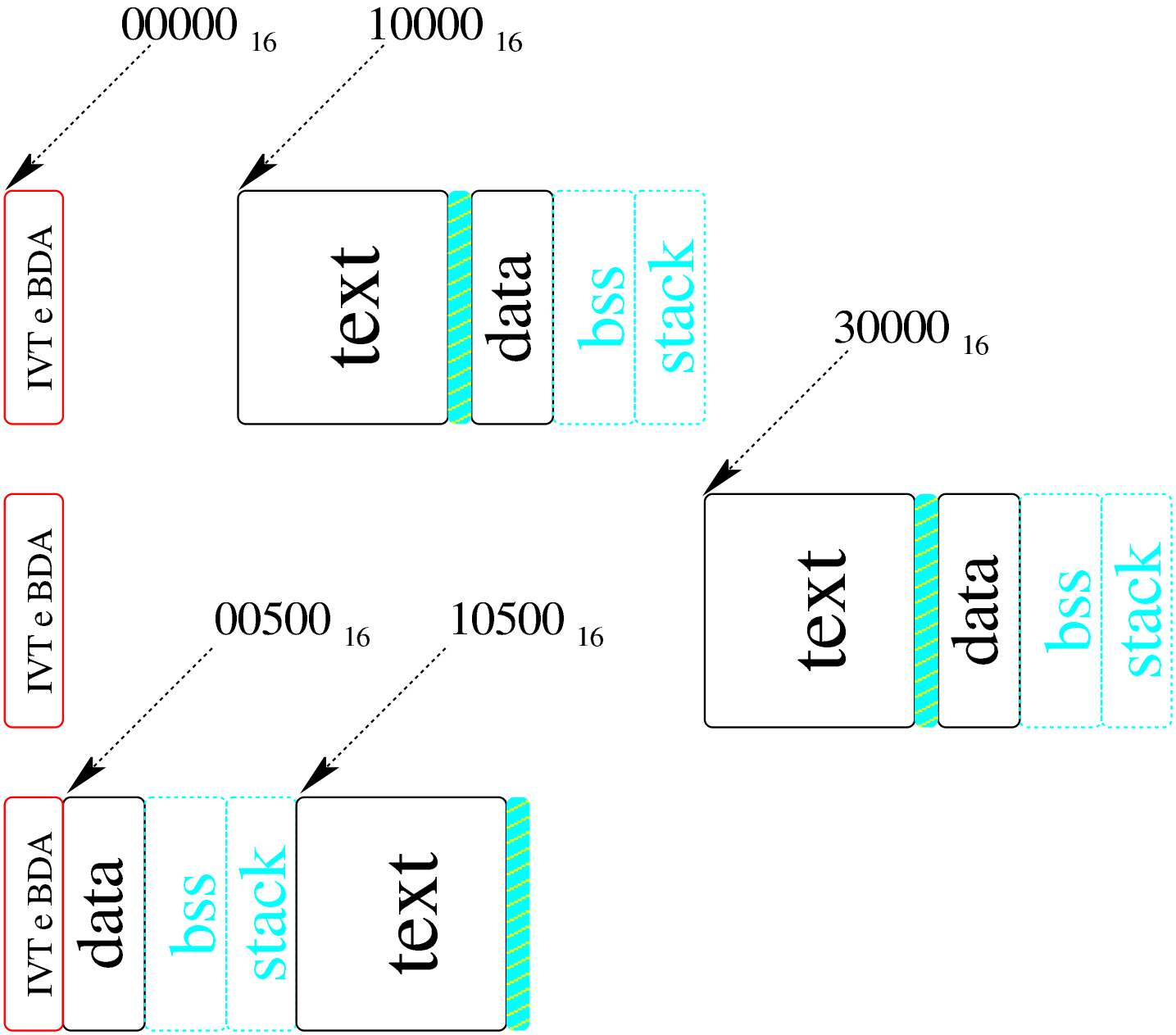
Il file del kernel prodotto dagli strumenti di sviluppo è strutturato come sintetizza il disegno seguente:



La prima parte del file è utilizzata dal codice (*text*), quindi ci può essere un piccolissimo spazio inutilizzato, seguito dalla porzione che riguarda i dati, tenendo conto che nel file ci sono solo i dati inizializzati, mentre gli altri non hanno bisogno di essere rappresentati, ma in memoria occupano comunque il loro spazio.

Il kernel è organizzato per tenere separate l'area delle istruzioni da quella dei dati, pertanto il compilatore (precisamente il «collegatore», ovvero il *linker*) offre il simbolo ***__segoff*** , con il quale si conosce la distanza del segmento dei dati dall'inizio del file. Il valore di questo scostamento è espresso in «paragrafi», ovvero in multipli di 16; in pratica si tratta dello scostamento da utilizzare in un registro di segmento. Dal momento che lo scostamento effettivo è costituito dalla dimensione dell'area del codice, approssimata per eccesso ai 16 byte successivi, tra la fine dell'area codice e l'inizio di quella dei

dati c'è quel piccolo spazio vuoto a cui già si è fatto riferimento.



Il kernel viene caricato in memoria, con l'ausilio di Bootblocks, all'indirizzo 10000_{16}. Da lì il kernel si mette in funzione e, prima si copia all'indirizzo 30000_{16}, quindi riprende a funzionare dal nuovo indirizzo, poi si copia mettendo i dati a partire dall'indirizzo 00500_{16} (dopo la tabella IVT e dopo l'area BDA) e il codice a partire dall'indirizzo 10500_{16}. Alla fine, riprende a funzionare dall'indirizzo 10500_{16}. La pila dei dati (*stack*) viene attivata solo quando il kernel

ha trovato la sua collocazione definitiva.

File «kernel/main/crt0.s»

«

Listato i160.7.2.

Dopo il preambolo in cui si dichiarano i simboli esterni e quelli interni da rendere pubblici, con l'istruzione '**`entry startup`**' si dichiara all'assemblatore che il punto di partenza è costituito dal simbolo '**`startup`**', ma in ogni caso questo deve essere all'inizio del codice, mancando un'intestazione precostituita. In pratica, la primissima cosa che si ottiene nel file eseguibile finale è un'istruzione di salto a una posizione più avanzata del codice, dove si colloca il simbolo '**`startup_code`**', e nello spazio intermedio (tra quell'istruzione di salto e il codice che si trova a partire da '**`startup_code`**') si collocano le impronte di riconoscimento, oltre ai dati sulla dislocazione dell'eseguibile in memoria.

```
…
entry startup
…
startup:
    jmp startup_code
…
startup_code:
…
```

Tra la prima istruzione di salto e le impronte di riconoscimento, introdotte dal simbolo '**`magic`**', c'è uno spazio vuoto (nullo), calcolato automaticamente in modo da garantire che la prima impronta inizi all'indirizzo relativo 0004_{16}. Di seguito vengono gli altri dati.

```
…
startup:
```

```
    jmp startup_code
filler:
    .space (0x0004 - (filler - startup))
magic:
    .data4 0x6F733136
    .data4 0x6B65726E
segoff:
    .data2 __segoff
etext:
    .data2 __etext
edata:
    .data2 __edata
ebss:
    .data2 __end
stack_size:
    .data2 0x0000
.align 2
startup_code:
…
```

A partire da '**startup_code**' viene analizzato il valore effettivo del registro ***CS***. Se questo è pari a 1000_{16}, significa che il kernel si trova in memoria a partire dall'indirizzo efficace 10000_{16}, ma in tal caso si salta a una procedura che copia il kernel in un'altra posizione di memoria (30000_{16}); se invece il valore di ***CS*** viene riconosciuto pari a quello della destinazione della prima copia, si passa a un'altra procedura che scompone l'area dati e l'area codice (testo) del kernel, in modo da collocare l'area dati a partire da 00500_{16} e l'area codice a partire da 10500_{16}. Quando si riconosce che il valore di ***CS*** è quello finale, si salta al simbolo '**main_code**' e da lì inizia il lavoro vero e proprio.

```
…
startup_code:
    mov cx, cs
    xor cx, #0x1000
    jcxz move_code_from_0x1000_to_0x3000
    mov cx, cs
    xor cx, #0x3000
    jcxz move_code_from_0x3000_to_0x0050
    mov cx, cs
    xor cx, #0x1050
    jcxz main_code
    hlt
    jmp startup_code
move_code_from_0x1000_to_0x3000:
    …
    jmp far #0x3000:#0x0000
move_code_from_0x3000_to_0x0050:
    …
    jmp far #0x1050:#0x0000
main_code:
…
```

Non si prevede che il kernel possa trovarsi in memoria in una collocazione differente da quelle stabilite nelle varie fasi di avvio, pertanto, in caso contrario, si crea semplicemente un circolo vizioso senza uscita.

Dal simbolo '**`main_code`**' inizia finalmente il lavoro e si procede con l'allineamento dei registri dei segmenti dei dati, in modo che siano tutti corrispondenti al valore previsto: 0050_{16} (il segmento in cui inizia l'area dati, secondo la collocazione prevista). Viene poi posizionato il valore del registro ***SP*** a zero, in modo che al primo inserimento questo punti esattamente all'indirizzo più grande che si

possa raggiungere nel segmento dati ($FFFE_{16}$, considerato che gli inserimenti nella pila sono a 16 bit).

```
…
main_code:
    mov  ax, #0x0050
    mov  ds, ax
    mov  ss, ax
    mov  es, ax
    mov sp, #0x0000
    …
```

Appena la pila diventa operativa, si inizializza anche il registro ***FLAGS***, verificando di disabilitare inizialmente le interruzioni.

```
…
main_code:
    …
    push #0
    popf
    cli
    …
```

A questo punto, si chiama la funzione ***main()***, fornendo come argomenti tre valori a zero.

```
…
main_code:
    …
    push #0
    push #0
    push #0
    call _main
    add  sp, #2
    add  sp, #2
    add  sp, #2
…
```

Nel caso la funzione dovesse terminare e restituire il controllo, si passerebbe al codice successivo al simbolo '**halt**', con cui si crea un ciclo senza uscita, corrispondente alla conclusione del funzionamento del kernel.

```
…
halt:
    hlt
    jmp halt
…
```

Utilizzando il compilatore Bcc per compilare ciò che descrive la funzione ***main()***, viene richiesta la presenza della funzione ***__mkargv()*** (il simbolo '**___mkargv**'), che in questo caso può limitarsi a non fare alcunché.

```
…
___mkargv:
    ret
…
```

File «kernel/main.h» e «kernel/main/*»

Listato u0.7 e successivi.

Tutto il lavoro del kernel di os16 si sintetizza nella funzione ***main()***, contenuta nel file ‘`kernel/main/main.c`’. Per poter dare un significato a ciò che vi appare al suo interno, occorre conoscere tutto il resto del codice, ma inizialmente è utile avere un’idea di ciò che succede, se poi si vuole compilare ed eseguire il sistema operativo.

La funzione ***main()*** viene dichiarata secondo la forma tradizionale di un programma per sistemi POSIX, ma gli argomenti che riceve dalla chiamata contenuta nel file ‘`kernel/main/crt0.s`’ sono nulli, perché nessuna informazione gli viene passata effettivamente.

```
…
int
main (int argc, char *argv[], char *envp[])
{
    …
    tty_init ();
    k_printf ("os16 build %s ram %i Kibyte\n", BUILD_DATE,
              int12 ());
    dsk_setup ();
    heap_clear ();
    proc_init ();
    menu ();
    …
```

Dopo la dichiarazione delle variabili si inizializza la gestione del video della console con la funzione ***tty_init()***, si mostra un messaggio iniziale, quindi si passa alla predisposizione di ciò che serve, prima di poter avviare dei processi. In particolare va osservata la funzione ***heap_clear()***, la quale inizializza con il codice $FFFF_{16}$ lo spazio di

memoria libero, tra la fine delle variabili «statiche» e il livello che ha raggiunto in quel momento la pila dei dati. Successivamente, avendo marcato in questo modo quello spazio, diventa possibile riconoscere empiricamente quanto spazio di quella porzione di memoria avrebbe potuto essere utilizzato, senza essere sovrascritto dalla pila dei dati. Il messaggio iniziale contiene la data di compilazione e la memoria libera (la macro-variabile ***BUILD_DATE*** viene definita dallo script '**`makeit`**', usato per la compilazione, creando il file '`kernel/main/build.h`' che viene poi incluso dal file '`kernel/main/main.c`').

L'attivazione della gestione dei processi (e delle interruzioni) con la funzione ***proc_init()***, comporta anche l'innesto del file system principale (chiamando da lì la funzione ***sb_mount()***).

```
…
int
main (int argc, char *argv[], char *envp[])
{
    …
    menu ();
    for (exit = 0; exit == 0;)
      {
        sys (SYS_0, NULL, 0);
        dev_io ((pid_t) 0, DEV_TTY, DEV_READ, 0L, &key, 1,
                NULL);
        …
        switch (key)
          {
            case 'h':
                menu ();
                break;
            …
```

```
            case 'x':
                exit = 1;
                break;
            case 'q':
                k_printf ("System halted!\n");
                return (0);
                break;
        }
    }
...
```

A questo punto il kernel ha concluso le sue attività preliminari e, per motivi diagnostici, mostra un menù, quindi inizia un ciclo in cui ogni volta esegue una chiamata di sistema nulla e poi legge un carattere dalla tastiera: se risulta premuto un tasto previsto, fa quanto richiesto e riprende il ciclo. La chiamata di sistema nulla serve a far sì che lo schedulatore ceda il controllo a un altro processo, ammesso che questo esista, consentendo l'avvio di processi ancor prima di avere messo in funzione quel processo che deve svolgere il ruolo di '**`init`**'.

In generale le chiamate di sistema sono fatte per essere usate solo dalle applicazioni; tuttavia, in pochi casi speciali il kernel le deve utilizzare come se fosse proprio un'applicazione. Qui si rende necessario l'uso della chiamata nulla, perché quando è in funzione il codice del kernel non ci possono essere interruzioni esterne e quindi nessun altro processo verrebbe messo in condizione di funzionare.

Le funzioni principali disponibili in questa modalità diagnostica

sono riassunte nella tabella successiva:

Tasto	Risultato
[*h*]	Mostra il menù di funzioni disponibili.
[*l*]	Invia il segnale '**SIGKILL**' al processo numero uno.
[*2*]...[*9*]	Invia il segnale '**SIGTERM**' al processo con il numero corrispondente.
[*A*]...[*F*]	Invia il segnale '**SIGTERM**' al processo con il numero da 10 a 15.
[*a*], [*b*], [*c*]	Avvia il programma '`/bin/aaa`', '`/bin/bbb`' o '`/bin/ccc`'.
[*f*]	Mostra l'elenco dei file aperti nel sistema.
[*m*], [*M*]	Innesta o stacca il secondo dischetto dalla directory '`/usr/`'.
[*n*], [*N*]	Mostra l'elenco degli inode aperti: l'elenco è composto da due parti.
[*l*]	Invia il segnale '**SIGCHLD**' al processo numero uno.
[*p*]	Mostra la situazione dei processi e altre informazioni.
[*x*]	Termina il ciclo e successivamente si passa all'avvio di '`/bin/init`'.
[*q*]	Ferma il sistema.

Premendo [*x*], il ciclo termina e il kernel avvia '`/bin/init`'. Quindi si mette in un altro ciclo, dove si limita a passare ogni volta il controllo allo schedulatore, attraverso la chiamata di sistema nulla.

```
…
int
main (int argc, char *argv[], char *envp[])
{
    …
    menu ();
    for (exit = 0; exit == 0;)
      {
        …
```

```
      }
    exec_argv[0] = "/bin/init";
    exec_argv[1] = NULL;
    pid = run ("/bin/init", exec_argv, NULL);
    while (1)
      {
        sys (SYS_0, NULL, 0);
      }
    …
}
```

Figura u143.15. Aspetto di os16 in funzione, con il menù in evidenza.

```
os16 build 20YY.MM.DD HH:MM:SS ram 639 Kibyte
.-------------------------------------------------------------.
| [h]      show this menu                                     |
| [p]      process status and memory map                      |
| [1]..[9] kill process  1 to 9                               |
| [A]..[F] kill process 10 to 15                              |
| [l]      send SIGCHLD to process 1                          |
| [a]..[c] run programs ‘/bin/aaa’ to ‘/bin/ccc’ in parallel  |
| [f]      system file status                                 |
| [n], [N] list of active inodes                              |
| [m], [M] mount/umount ‘/dev/dsk1’ at ‘/usr/’                |
| [x]      exit interaction with kernel and start ‘/bin/init’ |
| [q]      quit kernel                                        |
‘-------------------------------------------------------------’
```

Figura u143.16. Aspetto di os16 in funzione mentre visualizza anche la tabella dei processi avviati (tasto [*p*]). *http://www.youtube.com/watch?v=0gbgNpRrXBU*

```
ababaaababaaababaabbaaababaaababap
pp   p pg
id id rp   tty  uid euid suid usage s iaddr isiz daddr dsiz sp   name
 0  0  0 0000    0    0    0 00.35 R 10500 eb7c 00500 0000 ffc8 os16 kernel
 0  1  0 0000    0    0    0 00.33 r 2f100 0600 2f700 aa00 a8e8 /bin/ccc
 0  2  0 0000    0    0    0 00.01 r 1f100 0600 84300 aa00 a8e8 /bin/ccc
 2  3  0 0000   10   10   10 00.01 r 1f100 0600 44b00 aa00 a8e8 /bin/ccc
 0  4  0 0000    0    0    0 00.17 r 21600 0600 3a100 aa00 a8e8 /bin/ccc
 4  5  0 0000   10   10   10 00.02 r 21c00 2400 6f100 a900 a86c /bin/aaa
 4  6  0 0000   11   11   11 00.02 s 24000 2500 59d00 a900 a8b6 /bin/bbb
 0  7  0 0000    0    0    0 00.13 r 26500 0600 64600 aa00 a8e8 /bin/ccc
 7  8  0 0000   10   10   10 00.02 r 26b00 2400 8ee00 a900 981e /bin/aaa
 7  9  0 0000   11   11   11 00.02 s 2bf00 2500 79a00 a900 a8b6 /bin/bbb
CS=1050 DS=0050 SS=0050 ES=0050 BP=ffe4 SP=ffe4 heap_min=878c etext=eb7c edata=1
b3c ebss=4c34 ksp=ffc8 clock=0000084b, time elapsed=00:01:57
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffe00000003ffffffffffffffffffffffffff
fffe000000000001fffffffff0007fffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff800
0000000000000000000000000000000000000007ffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffff7ffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffbfffffffffffffffffffffffffffffffffffffffe0000000000000000000000000000f

abbaaaaababaaababaaabbaabbaabababbaabbbbbbbbbbbbb
```

Figura u143.17. Aspetto di os16 in funzione con il menù in evidenza, dopo aver premuto il tasto [*x*] per avviare '**init**'. *http://www.youtube.com/watch?v=epql4EhgWPU*

```
os16 build 20YY.MM.DD HH:MM:SS ram 639 Kibyte
.-------------------------------------------------------------------.
| [h]      show this menu                                           |
| [p]      process status and memory map                            |
| [1]..[9] kill process  1 to 9                                     |
| [A]..[F] kill process 10 to 15                                    |
| [l]      send SIGCHLD to process 1                                |
| [a]..[c] run programs `/bin/aaa' to `/bin/ccc' in parallel        |
| [f]      system file status                                       |
| [n], [N] list of active inodes                                    |
| [m], [M] mount/umount `/dev/dsk1' at `/usr/'                      |
| [x]      exit interaction with kernel and start `/bin/init'       |
| [q]      quit kernel                                              |
`-------------------------------------------------------------------'
init
os16: a basic os. [Ctrl q], [Ctrl r], [Ctrl s], [Ctrl t] to change
console.
This is terminal /dev/console0
Log in as "root" or "user" with password "ciao" :-)
login:
```

Funzioni interne legate all'hardware

Libreria: «lib/sys/os16.h» e «lib/sys/os16/...» 3064
Funzioni di basso livello dei file «kernel/ibm_i86/*» 3066
Gestione della console 3072
Gestione dei dischi 3075

bp() 3065 cli() 3067 con_char_read() 3073
con_char_ready() 3073 con_char_wait() 3073
con_init() 3073 con_putc() 3073 con_scroll() 3073
con_select() 3073 cs() 3065 ds() 3065 dsk_chs_t 3075
dsk_read_bytes() 3077 dsk_read_sectors() 3077
dsk_reset() 3077 dsk_sector_to_chs() 3077
dsk_setup() 3077 dsk_t 3075 dsk_write_bytes() 3077
dsk_write_sectors() 3077 es() 3065 ibm_i86.h 3063
int10_00() 3067 int10_02() 3067 int10_05() 3067
int12() 3067 int13_00() 3067 int13_02() 3067
int13_03() 3067 int16_00() 3067 int16_01() 3067
int16_02() 3067 in_16() 3067 in_8() 3067 irq_off()
3067 irq_on() 3067 os16.h 3064 out_16() 3067 out_8()
3067 ram_copy() 3067 seg_d() 3065 seg_i() 3065 sp()
3065 ss() 3065 sti() 3067 _bp() 3065 _cs() 3065 _ds()
3065 _es() 3065 _int10_00() 3067 _int10_02() 3067
_int10_05() 3067 _int12() 3067 _int13_00() 3067
_int13_02() 3067 _int13_03() 3067 _int16_00() 3067
_int16_01() 3067 _int16_02() 3067 _in_16() 3067
_in_8() 3067 _out_16() 3067 _out_8() 3067

_ram_copy() 3067 _seg_d() 3065 _seg_i() 3065 _sp() 3065 _ss() 3065

Il file ‘kernel/ibm_i86.h’ e quelli contenuti nella directory ‘kernel/ibm_i86/’, raccolgono il codice del kernel che è legato strettamente all’hardware; a questi file vanno anche aggiunti ‘lib/sys/os16.h’ e la directory ‘lib/sys/os16/’, della libreria, utilizzati anche dalle applicazioni, a vario titolo. In generale si può osservare la presenza di funzioni che si avvalgono direttamente di alcune interruzioni del BIOS (fondamentalmente per la gestione del video e per l’accesso ai dischi); funzioni che permettono di leggere il valore di alcuni registri; funzioni per leggere e scrivere la memoria, in posizioni arbitrarie; funzioni per facilitare la lettura e la scrittura nei dischi, anche a livello di byte.

Salvo poche eccezioni, le funzioni scritte in linguaggio assemblatore hanno nomi che iniziano con un trattino basso, ma a fianco di queste sono anche disponibili delle macroistruzioni, con nomi equivalenti, senza il trattino basso iniziale, per garantire che gli argomenti della chiamata abbiano il tipo corretto, restituendo un valore intero «normale», quando qualcosa deve essere restituito.

Libreria: «lib/sys/os16.h» e «lib/sys/os16/...»

«

Listato u0.12 e successivi.

Nel file ‘lib/sys/os16.h’ e in quelli della directory ‘lib/sys/os16/’ si raccolgono, tra le altre, delle funzioni di basso livello che possono essere utili per il kernel e per le applicazioni. Si tratta di ***_seg_i()*** e ***_seg_d()*** (ovvero le macroistruzioni ***seg_i()*** e ***seg_d()***), con cui si ottiene, rispettivamente, il numero del segmento codice (istruzioni) e il numero del segmento dati. Inoltre, per poter verifi-

care gli altri registri di segmento e i registri di gestione della pila, si aggiungono le funzioni ***_cs()***, ***_ds()***, ***_ss()***, ***_es()***, ***_sp()*** e ***_bp()***; le quali, rispettivamente, consentono di leggere il valore dei registri ***CS***, ***DS***, ***SS***, ***ES***, ***SP*** e ***BP*** (le macroistruzioni equivalenti sono ***cs()***, ***ds()***, ***ss()***, ***es()***, ***sp()*** e ***bp()***).

Tabella u144.1. Funzioni e macroistruzioni legate strettamente all'hardware, dichiarate nel file di intestazione '`lib/sys/os16.h`'. Tali funzioni e macroistruzioni possono essere utilizzate sia dal kernel, sia dalle applicazioni.

Funzione o macroistruzione	Descrizione

`uint16_t _seg_i (void);`
 	
`unsigned int seg_i (void);`
	Restituisce il numero del segmento codice (*instruction*).
	Listati u0.12 e i161.12.6.

`uint16_t _seg_d (void);`
 	
`unsigned int seg_d (void);`
	Restituisce il numero del segmento dati.
	Listati u0.12 e i161.12.5.

`uint16_t _cs (void);`
 	
`unsigned int cs (void);`
	Restituisce il valore del registro ***CS***.
	Listati u0.12 e i161.12.2.

`uint16_t _ds (void);`
 	
`unsigned int ds (void);`
	Restituisce il valore del registro ***DS***.
	Listati u0.12 e i161.12.3.

`uint16_t _ss (void);`
 	
`unsigned int ss (void);`
	Restituisce il valore del registro ***SS***.
	Listati u0.12 e i161.12.8.

Funzione o macroistruzione	Descrizione

`uint16_t _es (void);`
 	
`unsigned int es (void);`
	Restituisce il valore del registro ***ES***.
	Listati u0.12 e i161.12.4.

`uint16_t _sp (void);`
 	
`unsigned int sp (void);`
	Restituisce il valore del registro ***SP***. Il valore che si ottiene si riferisce allo stato del registro, prima di chiamare la funzione.
	Listati u0.12 e i161.12.7.

`uint16_t _bp (void);`
 	
`unsigned int bp (void);`
	Restituisce il valore del registro ***BP***. Il valore che si ottiene si riferisce allo stato del registro, prima di chiamare la funzione.
	Listati u0.12 e i161.12.1.

Funzioni di basso livello dei file «kernel/ibm_i86/*»

« Listato u0.5 e successivi.

Le funzioni con nomi che iniziano per '**_int*nn*...()**', dove ***nn*** è un numero di due cifre, in base sedici, consentono l'accesso all'interruzione ***nn*** del BIOS dal codice in linguaggio C.

Le funzioni con nomi del tipo '**_in_*n*()**' e '**_out_*n*()**' consentono di leggere e di scrivere un valore di ***n*** bit in una certa porta.

La funzione ***cli()*** disabilita le interruzioni hardware, mentre ***sti()*** le riabilita. Queste due funzioni vengono usate pochissimo nel codice del kernel. A loro si aggiungono le funzioni ***irq_on()*** e ***irq_off()***, per abilitare o escludere selettivamente un tipo di interruzione hardware. Queste funzioni vengono usate in una sola occasione, quan-

do si predispone la tabella IVT e poi si abilitano esclusivamente le interruzioni utili.

La funzione ***_ram_copy()*** si occupa di copiare una quantità stabilita di byte da una posizione della memoria a un'altra, entrambe indicate con segmento e scostamento (la funzione ***mem_copy()*** elencata in '`kernel/memory.h`' si avvale in pratica di questa).

Per agevolare l'uso di queste funzioni, senza costringere a convertire i valori numerici, sono disponibili diverse macroistruzioni con nomi equivalenti, ma privi del trattino basso iniziale.

Tabella u144.2. Funzioni e macroistruzioni di basso livello, dichiarate nel file di intestazione '`kernel/ibm_i86.h`' e descritte nei file della directory '`kernel/ibm_i860/`'. Le macroistruzioni hanno argomenti di tipo numerico non precisato, purché in grado di rappresentare il valore necessario.

Funzione o macroistruzione	Descrizione

`void _int10_00 (uint16_t` ***video_mode*** `);`
 	
`void int10_00 (`***video_mode***`);`
	Imposta la modalità video della console. Questa funzione viene usata solo da ***con_init()***, per inizializzare la console; la modalità video è stabilita dalla macro-variabile ***IBM_I86_VIDEO_MODE***, dichiarata nel file '`kernel/ibm_i86.h`'.

Funzione o macroistruzione	Descrizione

`void _int10_02 (uint16_t` ***page*****`,`**
`uint16_t` ***position*****`);`
`void int10_02 (`***page*****`,` ***position*****`);`
	Colloca il cursore in una posizione determinata dello schermo, relativo a una certa pagina video.
	Questa funzione viene usata solo da ***con_putc()***.

`void _int10_05 (uint16_t` ***page*****`);` `void int10_05 (`***page*****`);`	Seleziona la pagina attiva del video. Questa funzione viene usata solo da ***con_init()*** e ***con_select()***.
`void _int12 (void);` `void int12 (void);`	Restituisce la quantità di memoria disponibile, in multipli di 1 024 byte.
`void _int13_00 (uint16_t` ***drive*****`);` `void int13_00 (`***drive*****`);`	Azzera lo stato dell'unità a disco indicata, rappresentata da un numero secondo le convenzioni del BIOS. Viene usata solo dalle funzioni '**`dsk_...()`**' che si occupano dell'accesso alle unità a disco.

Funzione o macroistruzione	Descrizione
`uint16_t _int13_02 (uint16_t` ***drive*****`,`** `uint16_t` ***sectors*****`,`** `uint16_t` ***cylinder*****`,`** `uint16_t` ***head*****`,`** `uint16_t` ***sector*****`,`** `void *`***buffer***`);` `void int13_02 (`***drive*****`,`** ***sectors*****`,`** ***cylinder*****`,`** ***head*****`,`** ***sector*****`,`** ***buffer***`);`	Legge dei settori da un'unità a disco. Questa funzione viene usata soltanto da ***dsk_read_sectors()***.
`uint16_t _int13_03 (uint16_t` ***drive*****`,`** `uint16_t` ***sectors*****`,`** `uint16_t` ***cylinder*****`,`** `uint16_t` ***head*****`,`** `uint16_t` ***sector*****`,`** `void *`***buffer***`);` `void int13_03 (`***drive*****`,`** ***sectors*****`,`** ***cylinder*****`,`** ***head*****`,`** ***sector*****`,`** ***buffer***`);`	Scrive dei settori in un'unità a disco. Questa funzione viene usata solo da ***dsk_write_sectors()***.
`uint16_t _int16_00 (void);` `void int16_00 (void);`	Legge un carattere dalla tastiera, rimuovendolo dalla memoria tampone relativa. Viene usata solo in alcune funzioni di controllo della console, denominate '**`con_...()`**'.

Funzione o macroistruzione	Descrizione

`uint16_t _int16_01 (void);`
 	
`void int16_01 (void);`
	Verifica se è disponibile un carattere dalla tastiera: se c'è ne restituisce il valore, ma senza rimuoverlo dalla memoria tampone relativa, altrimenti restituisce zero. Viene usata solo dalle funzioni di gestione della console, denominate '**`con_...()`**'.

`void _int16_02 (void);`
 	
`void int16_02 (void);`
	Restituisce un valore con cui è possibile determinare quali funzioni speciali della tastiera risultano inserite (inserimento, fissamaiuscole, blocco numerico, ecc.). Al momento la funzione non viene usata.

`uint16_t _in_8 (uint16_t` ***port*****`);`** `void in_8 (`***port*****`);`**	Legge un byte dalla porta di I/O indicata. Questa funzione viene usata da ***irq_on()***, ***irq_off()*** e ***dev_mem()***.
`uint16_t _in_16 (uint16_t` ***port*****`);`** `void in_16 (`***port*****`);`**	Legge un valore a 16 bit dalla porta di I/O indicata. Questa funzione viene usata solo da ***dev_mem()***.

Funzione o macroistruzione	Descrizione

`void _out_8 (uint16_t` ***port***`,`
`uint16_t` ***value***`);`
`void out_8 (`***port***`,` ***value***`);`
	Scrive un byte nella porta di I/O indicata. Questa funzione viene usata da ***irq_on()***, ***irq_off()*** e ***dev_mem()***.

`void _out_16 (uint16_t` ***port***`,`
`uint16_t` ***value***`);`
`void out_16 (`***port***`,` ***value***`);`
	Scrive un valore a 16 bit nella porta indicata. Questa funzione viene usata solo da ***dev_mem()***.

`void cli (void);`	Azzera l'indicatore delle interruzioni, nel registro ***FLAGS***. La funzione serve a permettere l'uso dell'istruzione '**`CLI`**' dal codice in linguaggio C, ma in questa veste, viene usata solo dalla funzione ***proc_init()***.
`void sti (void);`	Attiva l'indicatore delle interruzioni, nel registro ***FLAGS***. La funzione serve a permettere l'uso dell'istruzione '**`STI`**' dal codice in linguaggio C, ma in questa veste, viene usata solo dalla funzione ***proc_init()***.

Funzione o macroistruzione	Descrizione
`void irq_on (unsigned int` ***irq*** `);`	Abilita l'interruzione hardware indicata. Questa funzione viene usata solo da ***proc_init()***.
`void irq_off (unsigned int` ***irq*** `);`	Disabilita l'interruzione hardware indicata. Questa funzione viene usata solo da ***proc_init()***.

`void _ram_copy (segment_t` ***org_seg*** `,`
`offset_t` ***org_off*** `,`
`segment_t` ***dst_seg*** `,`
`offset_t` ***dst_off*** `,`
`uint16_t` ***size*** `);`
`void ram_copy (`***org_seg*** `,`
org_off `,`
dst_seg `,`
dst_off `,`
size `);`
	Copia una certa quantità di byte, da una posizione di memoria all'altra, specificando segmento e scostamento di origine e destinazione. Viene usata solo dalle funzioni '**`mem_...()`**'.

Gestione della console

«

Listato u0.5 e successivi.

La console offre solo funzionalità elementari, dove è possibile scrivere o leggere un carattere alla volta, sequenzialmente. Ci sono al massimo quattro console virtuali, selezionabili attraverso le combinazioni di tasti [*Ctrl q*], [*Ctrl r*], [*Ctrl s*] e [*Ctrl t*] (ma nella configurazione predefinita vengono attivate solo le prime due) e non è possibile controllare i colori o la posizione del testo che si va a espor-

re; in pratica si opera come su una telescrivente. Le funzioni di livello più basso, relative alla console hanno nomi che iniziano per '**`con_...()`**'.

Nel codice del kernel si vede usata frequentemente la funzione ***k_printf()***, la quale va a utilizzare la funzione ***k_vprintf()***, dove poi, attraverso altri passaggi, si arriva a utilizzare la funzione ***con_putc()***.

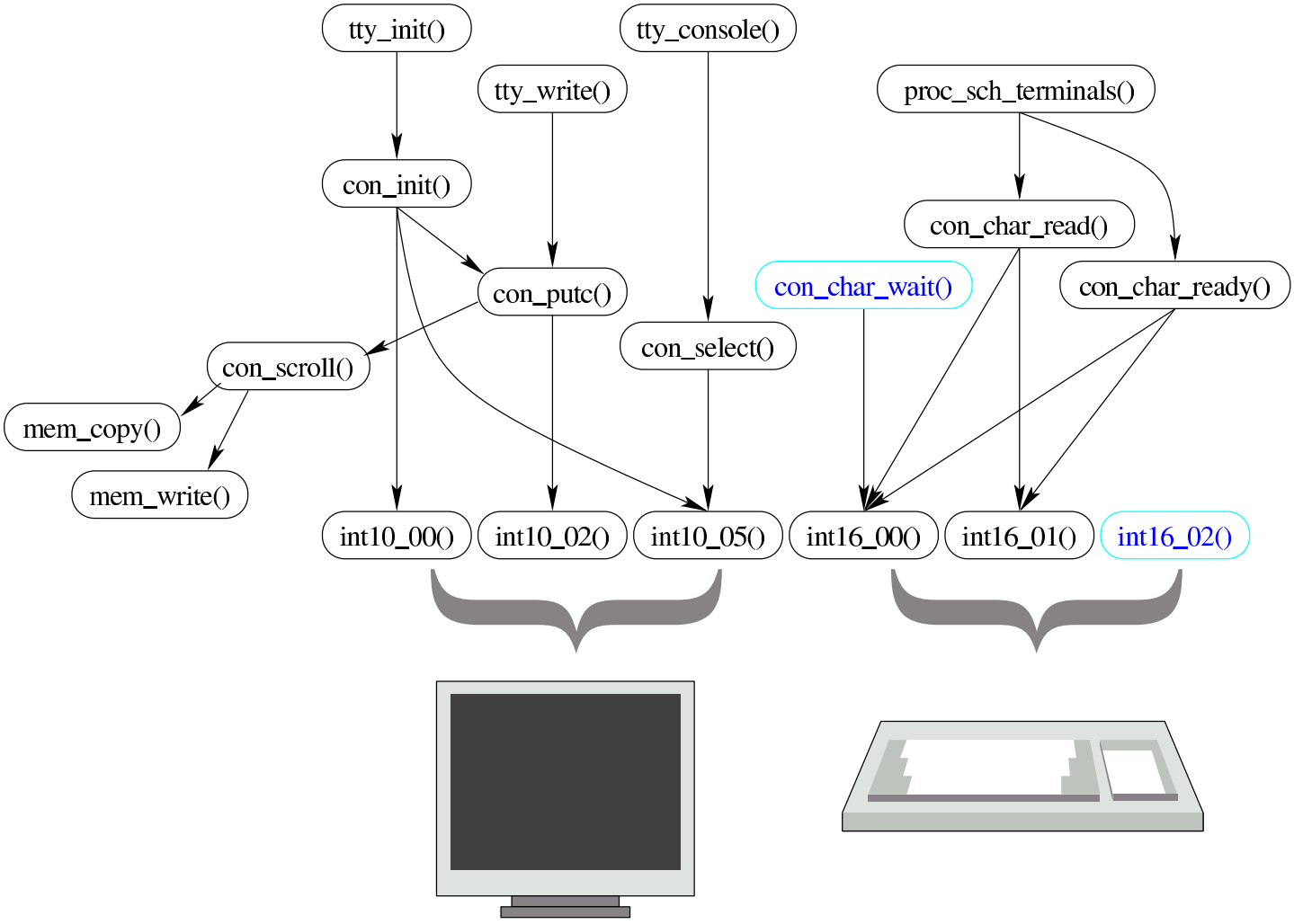
Tabella u144.3. Funzioni per l'accesso alla console, dichiarate nel file di intestazione '`kernel/ibm_i86.h`' e descritte nei file contenuti nella directory '`kernel/ibm_i86/`'.

Funzione	Descrizione
`int con_char_read (void);`	Legge un carattere dalla console, se questo è disponibile, altrimenti restituisce il valore zero. Questa funzione viene usata solo da ***proc_sch_terminals()***.
`int con_char_wait (void);`	Legge un carattere dalla console, ma se questo non è ancora disponibile, rimane in attesa, bloccando tutto il sistema operativo. Questa funzione non è utilizzata.
`int con_char_ready (void);`	Verifica se è disponibile un carattere dalla console: se è così, restituisce un valore diverso da zero, corrispondente al carattere in attesa di essere prelevato. Questa funzione viene usata solo da ***proc_sch_terminals()***.
`void con_init (void);`	Inizializza la gestione della console. Questa funzione viene usata solo da ***tty_init()***.

Funzione	Descrizione
`void con_select (int` ***console*** `);`	Seleziona la console desiderata, dove la prima si individua con lo zero. Questa funzione viene usata solo da ***tty_console()***.
`void con_putc (int` ***console*** `,` `int` ***c*** `);`	Visualizza il carattere indicato sullo schermo della console specificata, sulla posizione in cui si trova il cursore, facendolo avanzare di conseguenza e facendo scorrere il testo in alto, se necessario. Questa funzione viene usata solo da ***tty_write()***.
`void con_scroll (int` ***console*** `);`	Fa avanzare in alto il testo della console selezionata. Viene usata internamente, solo dalla funzione ***con_putc()***.

Nella figura successiva si vede l'interdipendenza tra le funzioni relative alla gestione di basso livello della console. In un altro capitolo si descrivono le funzioni '**`tty_...()`**', con le quali si gestiscono i terminali in forma più astratta. Nello schema successivo si può vedere che la funzione ***con_scroll()*** si avvale di funzioni per la gestione della memoria: infatti, lo scorrimento del testo dello schermo si ottiene intervenendo direttamente nella memoria utilizzata per la rappresentazione del testo sullo schermo.

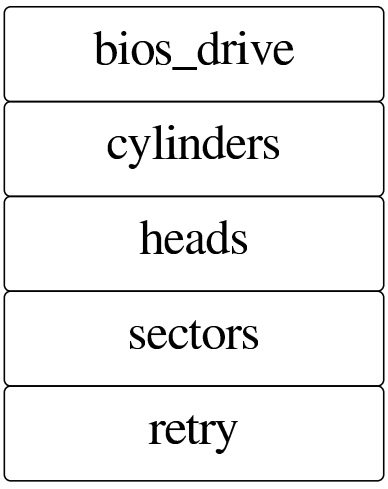
Figura u144.4. Interdipendenza tra le funzioni relative alla gestione della console.



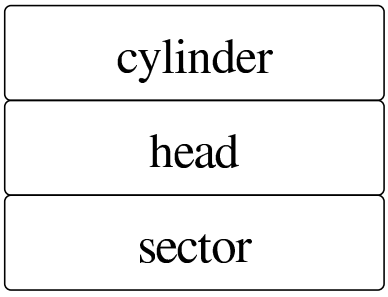
Gestione dei dischi

Listato u0.5 e successivi.

Nel file ‘`ibm_i86.h`’ vengono definiti due tipi derivati: ‘**`dsk_t`**’ per annotare le caratteristiche di un disco; ‘**`dsk_chs_t`**’ per annotare simultaneamente le coordinate di accesso a un disco, formate dal numero del cilindro, della testina e del settore.



dsk_t



dsk_chs_t

Le funzioni con nomi del tipo '**dsk_...()**' riguardano l'accesso ai dischi, a livello di settore o di byte, e utilizzano le informazioni annotate nell'array ***dsk_table[]***, composto da elementi di tipo '**dsk_t**'. In pratica, l'array ***dsk_table[]*** viene creato con '**DSK_MAX**' elementi (pertanto solo quattro), uno per ogni disco che si intende gestire. Quando le funzioni '**dsk_...()**' richiedono l'indicazione di un numero di unità (*drive*), si riferiscono all'indice dell'array ***dsk_table[]*** (al contrario, le funzioni '**_int13_...()**' hanno come riferimento il codice usato dal BIOS).

La funzione ***dsk_setup()*** compila l'array ***dsk_table[]*** con i dati relativi ai dischi che si utilizzano; la funzione ***dsk_reset()*** azzera la funzionalità di una certa unità; la funzione ***dsk_sector_to_chs()*** converte il numero assoluto di un settore nelle coordinate corrispondenti (cilindro, testina e settore).

Le funzioni ***dsk_read_sectors()*** e ***dsk_write_sectors()*** servono a leggere o scrivere una quantità stabilita di settori, usando come appoggio un'area di memoria individuata da un puntatore generico. Le funzioni ***dsk_read_bytes()*** e ***dsk_write_bytes()*** svolgono un compito equivalente, ma usando come riferimento il byte; in questo caso,

restituiscono la quantità di byte letti o scritti rispettivamente.

Tabella u144.6. Funzioni per l'accesso ai dischi, dichiarate nel file di intestazione '`kernel/ibm_i86.h`'.

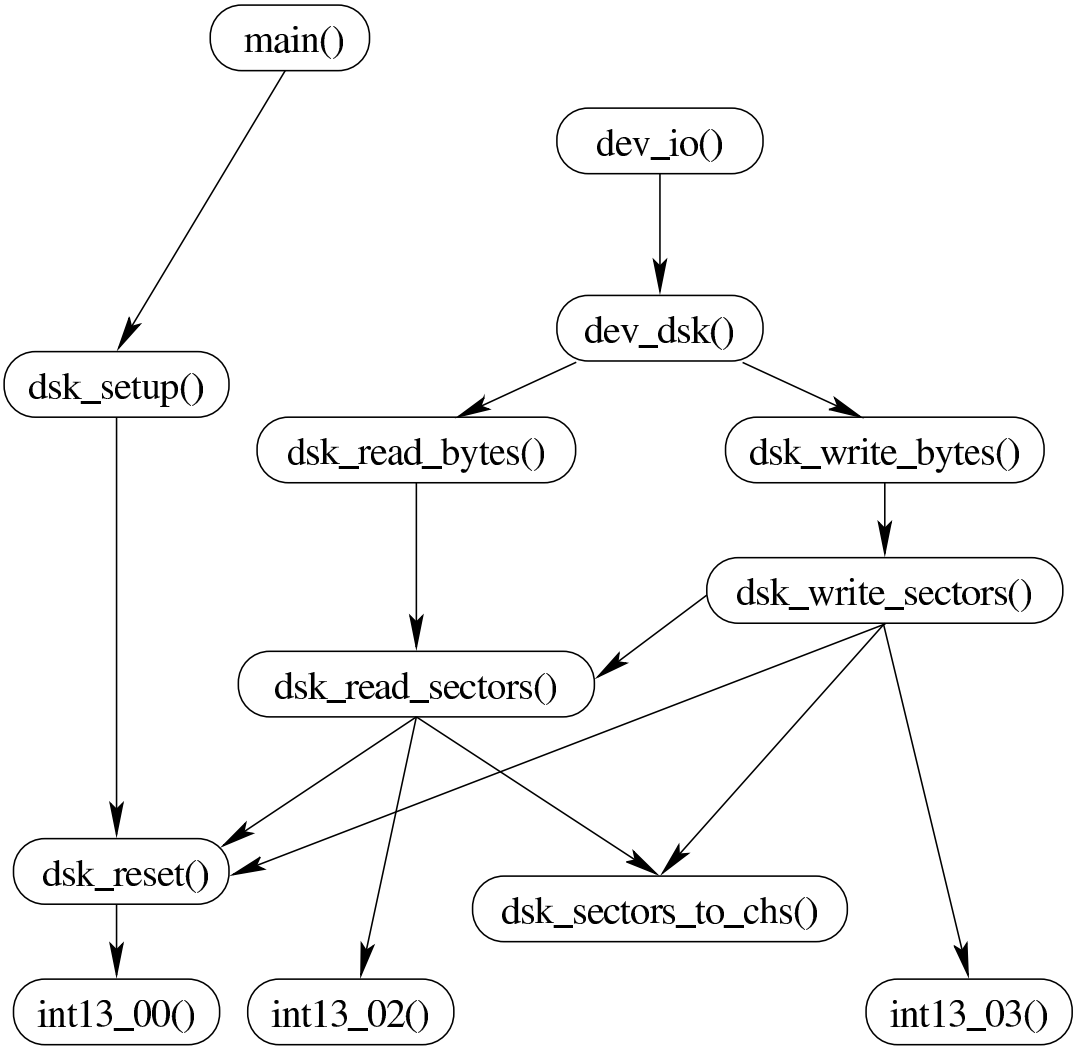
Funzione	Descrizione
`void dsk_setup (void);`	Predispone il contenuto dell'array ***dsk_table[]***. Questa funzione viene usata soltanto da ***main()***.
`int dsk_reset (int` ***drive*** `);`	Azzera lo stato dell'unità corrispondente a ***dsk_table[drive].bios_drive***. Viene usata solo internamente, dalle altre funzioni '**`dsk_...()`**'.

`void dsk_sector_to_chs`
`(int` ***drive*** `,`
`unsigned int` ***sector*** `,`
`dsk_chs_t *`***chs***`);`
	Modifica le coordinate della variabile strutturata a cui punta l'ultimo parametro, con le coordinate corrispondenti al numero di settore fornito. Viene usata solo internamente, dalle altre funzioni '**`dsk_...()`**'.

Funzione	Descrizione
`int dsk_read_sectors (int` ***drive*`,` `unsigned int` ***start_sector*`,` `void *`***buffer*`,` `unsigned int` ***n_sectors*`);`	Legge una sequenza di settori da un disco, mettendo i dati in memoria, a partire dalla posizione espressa da un puntatore generico. La funzione è ricorsiva, ma oltre che da se stessa, viene usata internamente da ***dsk_read_bytes()*** e da ***dsk_write_bytes()***.
`int dsk_write_sectors (int` ***drive*`,` `unsigned int` ***start_sector*`,` `void *`***buffer*`,` `unsigned int` ***n_sectors*`);`	Scrive una sequenza di settori in un disco, traendo i dati da un puntatore a una certa posizione della memoria. La funzione è ricorsiva, ma oltre che da se stessa, viene usata solo internamente da ***dsk_write_bytes()***.
`size_t dsk_read_bytes (int` ***drive*`,` `off_t` ***offset*`,` `void *`***buffer*`,` `size_t count);`	Legge da una certa unità a disco una quantità specificata di byte, a partire dallo scostamento indicato (nel disco), il quale deve essere un valore positivo. Questa funzione viene usata solo da ***dev_dsk()***.

Funzione	Descrizione
`size_t dsk_write_bytes (int` ***drive*****,** `off_t` ***offset*****,** `void *`***buffer*****,** `size_t count);`	Scrive su una certa unità a disco una quantità specificata di byte, a partire dallo scostamento indicato (nel disco), il quale deve essere un valore positivo. Questa funzione viene usata solo da ***dev_dsk()***.

Figura u144.7. Interdipendenza tra le funzioni relative all'accesso ai dischi.



Gestione della memoria

File «kernel/memory.h» e «kernel/memory/...» 3082

Scansione della mappa di memoria 3086

addr_t 3082 kernel/memory.c 3082 mb_alloc() 3084 mb_alloc_size() 3084 mb_free() 3084 mb_reference() 3084 memory.h 3082 memory_t 3082 MEM_BLOCK_SIZE 3082 mem_copy() 3086 MEM_MAX_BLOCKS 3082 mem_read() 3086 mem_write() 3086 offset_t 3082 segment_t 3082

Dal punto di vista del kernel di os16, l'allocazione della memoria riguarda la collocazione dei processi elaborativi nella stessa. Disponendo di una quantità di memoria esigua, si utilizza una mappa di bit per indicare lo stato dei blocchi di memoria, dove un bit a uno indica un blocco di memoria occupato.

Nel file 'memory.h' viene definita la dimensione di un blocco di memoria e, di conseguenza, la quantità massima che possa essere gestita. Attualmente i blocchi sono da 256 byte, pertanto, sapendo che la memoria può arrivare solo fino a 640 Kibyte, si gestiscono al massimo 2560 blocchi.

Per la scansione della mappa si utilizzano interi da 16 bit, pertanto tutta la mappa si riduce a 40 di questi interi, ovvero 80 byte. Nell'ambito di ogni intero da 16 bit, il bit più significativo rappresenta il primo blocco di memoria di sua competenza. Per esempio, per indicare che si stanno utilizzando i primi 1280 byte, pari ai primi cinque blocchi di memoria, si rappresenta la mappa della memoria come «F80000000...».

Il fatto che la mappa della memoria vada scandito a ranghi di 16 bit va tenuto in considerazione, perché se invece si andasse con ranghi differenti, si incapperebbe nel problema dell'inversione dei byte.

Quando possibile, si fa riferimento a indirizzi di memoria efficaci, nel senso che, con un solo valore, si rappresentano le posizioni da 00000_{16} a $FFFFF_{16}$. Per questo viene predisposto il tipo derivato '**`addr_t`**' nel file '`memory.h`'.

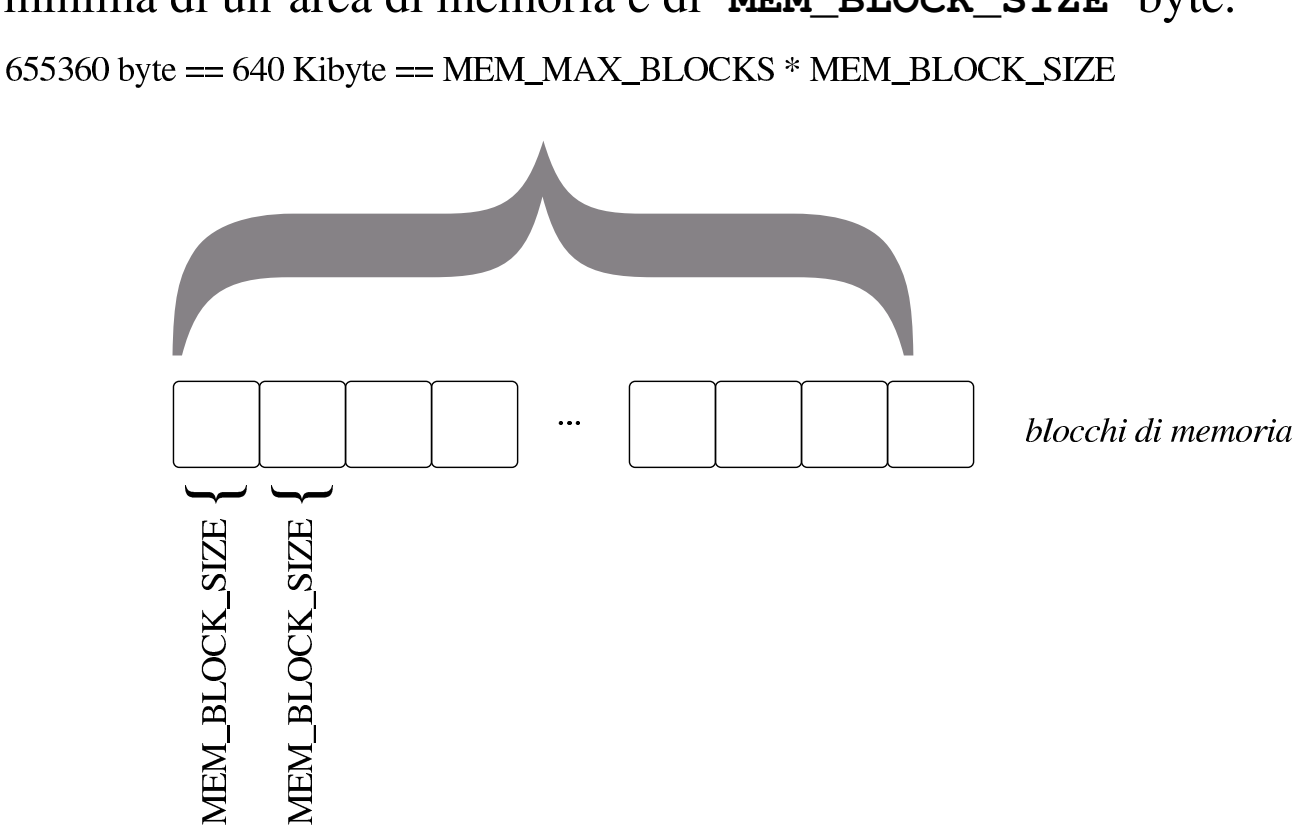
File «kernel/memory.h» e «kernel/memory/...»

«

Listato u0.8 e successivi.

Il file '`kernel/memory.h`', oltre ai prototipi delle funzioni usate per la gestione della memoria, definisce la dimensione del blocco minimo di memoria e la quantità massima di questi, rispettivamente con le macro-variabili ***MEM_BLOCK_SIZE*** e ***MEM_MAX_BLOCKS***; inoltre predispone i tipi derivati '**`memory_t`**', '**`segment_t`**', '**`offset_t`**' e '**`addr_t`**', corrispondenti, rispettivamente, a una variabile strutturata che consente di rappresentare un'area di memoria in modo relativamente comodo (indirizzo efficace, segmento e dimensione), un indirizzo di segmento, uno scostamento dall'inizio di un segmento e un indirizzo efficace.

Figura u145.1. Mappa della memoria in blocchi: la dimensione minima di un'area di memoria è di '**`MEM_BLOCK_SIZE`**' byte.



Nei file della directory '`kernel/memory/`' viene dichiarata la mappa della memoria, corrispondente a un array di interi a 16 bit, denominato ***mb_table[]***. L'array è pubblico, tuttavia è disponibile anche una funzione che ne restituisce il puntatore: ***mb_reference()***. Tale funzione sarebbe perfettamente inutile, ma rimane per uniformità rispetto alla gestione delle altre tabelle.

Nelle funzioni che riguardano l'allocazione della memoria, quando si indica la dimensione di questa, spesso si considera il valore zero equivalente a 10000_{16}, ovvero la dimensione massima di un segmento secondo l'architettura.

Tabella u145.2. Funzioni per la gestione della mappa della memoria, dichiarate nel file di intestazione ‘`kernel/memory.h`’ e realizzate nella directory ‘`kernel/memory/`’.

Funzione	Descrizione
`uint16_t *mb_reference (void);`	Restituisce il puntatore alla tabella dei blocchi di memoria, per uniformare l’accesso alla tabella dalle funzioni che non fanno parte del gruppo contenuto nella directory ‘`kernel/memory/`’.
`ssize_t mb_alloc (addr_t` ***address*****`,`** `size_t` ***size*****`);`**	Alloca la memoria a partire dall’indirizzo efficace indicato, per la quantità di byte richiesta (zero corrisponde a 10000_{16} byte). L’allocazione ha termine anticipatamente se si incontra un blocco già utilizzato. La funzione restituisce la dimensione allocata effettivamente.

Funzione	Descrizione
`ssize_t mb_free (addr_t` ***address*****`,`** `size_t` ***size*****`);`	Libera la memoria a partire dall'indirizzo efficace indicato, per la quantità di byte richiesta (zero corrisponde a 10000_{16} byte). Lo spazio viene liberato in ogni caso, anche se risulta già libero; tuttavia viene prodotto un avvertimento a video se si verifica tale ipotesi.

`int mb_alloc_size`
`(size_t` ***size*****`,`**
`memory_t *`***allocated***`);`
	Cerca e alloca un'area di memoria della dimensione richiesta, modificando la variabile strutturata di cui viene fornito il puntatore come secondo parametro. In pratica, l'indirizzo e l'estensione della memoria allocata effettivamente si trovano nella variabile strutturata in questione, mentre la funzione restituisce zero (se va tutto bene) o -1 se non è disponibile la memoria libera richiesta.

Tabella u145.3. Funzioni per le operazioni di lettura e scrittura in memoria, dichiarate nel file di intestazione ‘`kernel/memory.h`’ e realizzate nella directory ‘`kernel/memory/`’.

Funzione	Descrizione

`void mem_copy (addr_t` ***orig*`,`
`addr_t` ***dest*`,`
`size_t` ***size*`);`
	Copia la quantità richiesta di byte, dall’indirizzo di origine a quello di destinazione, espressi in modo efficace.

`size_t mem_read (addr_t` ***start*`,`
`void *`***buffer*`,`
`size_t` ***size*`);`
	Legge dalla memoria, a partire dall’indirizzo indicato come primo parametro, la quantità di byte indicata come ultimo parametro. Ciò che viene letto va poi copiato nella memoria tampone corrispondente al puntatore generico indicato come secondo parametro.

`size_t mem_write (addr_t` ***start*`,`
`void *`***buffer*`,`
`size_t` ***size*`);`
	Scrive, in memoria, a partire dall’indirizzo indicato come primo parametro, la quantità di byte indicata come ultimo parametro. Ciò che viene scritto proviene dalla memoria tampone corrispondente al puntatore generico indicato come secondo parametro.

Scansione della mappa di memoria

«

Listato u0.8 e successivi.

La mappa della memoria si rappresenta (a sua volta in memoria), con un array di interi a 16 bit, dove ogni bit individua un blocco di

memoria. Pertanto, l'array si compone di una quantità di elementi pari al valore di '**`MEM_MAX_BLOCKS`**' diviso 16.

Il primo elemento di questo array, ovvero ***mb_table[0]***, individua i primi 16 blocchi di memoria, dove il bit più significativo si riferisce precisamente al primo blocco. Per esempio, se ***mb_table[0]*** contiene il valore $F800_{16}$, ovvero 1111100000000000_2, significa che i primi cinque blocchi di memoria sono occupati, mentre i blocchi dal sesto al sedicesimo sono liberi.

Dal momento che i calcoli per individuare i blocchi di memoria e per intervenire nella mappa relativa, possono creare confusione, queste operazioni sono raccolte in funzioni statiche separate, anche se sono utili esclusivamente all'interno del file in cui si trovano. Tali funzioni statiche hanno una sintassi comune:

```
int mb_block_set1   (int block)
int mb_block_set0   (int block)
int mb_block_status (int block)
```

Le funzioni ***mb_block_set1()*** e ***mb_block_set0()*** servono rispettivamente a impegnare o liberare un certo blocco di memoria, individuato dal valore dell'argomento. La funzione ***mb_block_status()*** restituisce uno nel caso il blocco indicato risulti allocato, oppure zero in caso contrario.

Queste tre funzioni usano un metodo comune per scandire la mappa della memoria: il valore che rappresenta il blocco a cui si vuole fare riferimento, viene diviso per 16, ovvero il rango degli elementi dell'array che rappresenta la mappa della memoria. Il risultato intero della divisione serve per trovare quale elemento dell'array conside-

rare, mentre il resto della divisione serve per determinare quale bit dell'elemento trovato rappresenta il blocco desiderato. Trovato ciò, si deve costruire una maschera, nella quale si mette a uno il bit che rappresenta il blocco; per farlo, si pone inizialmente a uno il bit più significativo della maschera, quindi lo si fa scorrere verso destra di un valore pari al resto della divisione.

Per esempio, volendo individuare il terzo blocco di memoria, pari al numero 2 (il primo blocco corrisponderebbe allo zero), si avrebbe che questo è descritto dal primo elemento dell'array (in quanto 2/16 dà zero, come risultato intero), mentre la maschera necessaria a trovare il bit corrispondente è 0010000000000000_2, la quale si ottiene spostando per due volte verso destra il bit più significativo (due volte, pari al resto della divisione).

Una volta determinata la maschera, per segnare come occupato un blocco di memoria, basta utilizzare l'operatore OR binario:

```
        mb_table[i] = mb_table[i] | mask;
```

Se invece si vuole liberare un blocco di memoria, si utilizza un AND binario, invertendo però il contenuto della maschera:

```
        mb_table[i] = mb_table[i] & ~mask;
```

Va osservato che la rappresentazione dei blocchi nella mappa è invertita rispetto ad altri sistemi operativi, in quanto non sarebbe tanto logico il fatto che il bit più significativo si riferisca invece alla parte più bassa del proprio insieme di blocchi di memoria. La scelta è dovuta al fatto che, volendo rappresentare la mappa numericamente, la lettura di questa sarebbe più vicina a quella che è la percezione umana del problema.

tty.h 3089 tty_console() 3090 tty_init() 3090 tty_read() 3090 tty_reference() 3090 tty_write() 3090

Listato u0.10 e successivi.

os16 offre esclusivamente un utilizzo operativo tramite console. Alcune funzioni con prefisso '**con_...()**', dichiarate nel file 'kernel/ibm_i86.h', si occupano di tale gestione, ma per distinguere tra terminali virtuali (o console virtuali), associati a processi differenti, si rende necessario un livello ulteriore di astrazione, costituito dal codice contenuto nel file 'kernel/tty.h' e in quelli della directory 'kernel/tty/'.

I terminali virtuali gestibili sono rappresentati da un array di variabili strutturate, ognuna delle quali contiene tutte le informazioni del contesto operativo di un certo terminale. L'array in questione è ***tty_table[]*** (a cui però si accede tramite una funzione che ne restituisce il puntatore) e vi si annotano, per ogni terminale attivo, il numero del dispositivo corrispondente, il numero del gruppo di processi a cui si associa, l'ultimo codice digitato (e non ancora letto), lo stato di funzionamento (se sono stati persi dei dati o meno).

Va osservata la differenza sostanziale che c'è tra le operazioni di scrittura e quelle di lettura. Infatti, la scrittura sul terminale implica la chiamata della funzione ***con_putc()***, del file 'kernel/ibm_i86.h'; al contrario, la lettura avviene in forma passiva, limitandosi ad acquisire il valore già disponibile nella variabile strutturata che rap-

presenta il terminale virtuale. Come può essere verificato successivamente, è compito del sistema di gestione delle interruzioni la fornitura del valore digitato al terminale virtuale competente, tramite l'appoggio della variabile strutturata che lo rappresenta.

Come per tutte le tabelle di os16 che non fanno parte di uno standard, anche quella che contiene le informazioni dei terminali virtuali è accessibile preferibilmente con l'ausilio di una funzione che ne restituisce il puntatore. In questo caso, la funzione ***tty_reference()*** consente di ottenere il puntatore all'elemento corrispondente della tabella dei terminali, fornendo come argomento il numero del dispositivo cercato.

Tabella u146.1. Funzioni per la gestione dei terminali, dichiarate nel file di intestazione '`kernel/tty.h`'.

Funzione	Descrizione
`void tty_init (void);`	Inizializza la gestione dei terminali. Viene usata una volta sola nella funzione ***main()*** del kernel.
`tty_t *tty_reference (dev_t` ***device*** `);`	Restituisce il puntatore a un elemento della tabella dei terminali. Se come numero di dispositivo si indica lo zero, si ottiene il riferimento a tutta la tabella; se non viene trovato il numero di dispositivo cercato, si ottiene il puntatore nullo.

Funzione	Descrizione
`dev_t tty_console (dev_t` ***device*** `);`	Seleziona la console indicata attraverso il numero di dispositivo che costituisce l'unico parametro. Se viene dato un valore a zero, si ottiene solo di conoscere qual è la console attiva. La console selezionata viene anche memorizzata in una variabile statica, per le chiamate successive della funzione. Se viene indicato un numero di dispositivo non valido, si seleziona implicitamente la prima console.
`int tty_read (dev_t` ***device*** `);`	Legge un carattere dal terminale specificato attraverso il numero di dispositivo. Per la precisione, il carattere viene tratto dal campo relativo contenuto nella tabella dei terminali. Il carattere viene restituito dalla funzione come valore intero comune; se si ottiene zero significa che non è disponibile alcun carattere.

Funzione	Descrizione
`void tty_write (dev_t` ***device*****,** `int` ***c*****);**	Scrive sullo schermo del terminale rappresentato dal numero di dispositivo, il carattere fornito come secondo parametro.

File «lib/sys/os16.h» e directory «lib/sys/os16/» 3093
File «kernel/devices.h» e «kernel/devices/...» 3094
Numero primario e numero secondario 3098
Dispositivi previsti 3099

devices.h 3094 devices/dev_tty.c 3094 DEV_CONSOLE 3099 DEV_CONSOLE*n* 3099 dev_dsk.c 3094 DEV_DSK*n* 3099 dev_io() 3094 dev_io.c 3094 dev_kmem.c 3094 DEV_KMEM_FILE 3099 DEV_KMEM_INODE 3099 DEV_KMEM_MMP 3099 DEV_KMEM_PS 3099 DEV_KMEM_SB 3099 DEV_MEM 3099 DEV_NULL 3099 DEV_PORT 3099 DEV_TTY 3099 DEV_ZERO 3099 os16.h 3093

La gestione dei dispositivi fisici, da parte di os16, è molto limitata, in quanto ci si avvale prevalentemente delle funzionalità già offerte dal BIOS. In ogni caso, tutte le operazioni di lettura e scrittura di dispositivi, passano attraverso la gestione comune della funzione ***dev_io()***.

File «lib/sys/os16.h» e directory «lib/sys/os16/»

Listato u0.12 e altri.

Una parte delle definizioni che riguardano la gestione dei dispositivi, necessaria al kernel, è disponibile anche alle applicazioni attraverso il file ‘`lib/sys/os16.h`’. Al suo interno, tra le altre cose, è definito un insieme di macro-variabili, con prefisso ***DEV_***..., con cui si distinguono i numeri attribuiti ai dispositivi. Per esempio,

DEV_DSK_MAJOR corrisponde al numero primario (*major*) per tutte le unità di memorizzazione, mentre ***DEV_DSK1*** corrisponde al numero primario e secondario (*major* e *minor*), in un valore unico, della seconda unità a disco.

File «kernel/devices.h» e «kernel/devices/...»

«

Listati u0.2 e altri.

Il file ‘`kernel/devices.h`’ incorpora il file ‘`lib/sys/os16/os16.h`’, per acquisire le funzionalità legate alla gestione dei dispositivi che sono disponibili anche agli applicativi. Successivamente dichiara la funzione ***dev_io()***, la quale sintetizza tutta la gestione dei dispositivi. Questa funzione utilizza il parametro ***rw***, per specificare l'azione da svolgere (lettura o scrittura). Per questo parametro vanno usate le macro-variabili ***DEV_READ*** e ***DEV_WRITE***, così da non dover ricordare quale valore numerico corrisponde alla lettura e quale alla scrittura.

```
ssize_t dev_io (pid_t pid, dev_t device, int rw, off_t offset,
                void *buffer, size_t size, int *eof);
```

Sono comunque descritte anche altre funzioni, ma utilizzate esclusivamente da ***dev_io()***.

La funzione ***dev_io()*** si limita a estrapolare il numero primario dal numero del dispositivo complessivo, quindi lo confronta con i vari tipi gestibili. A seconda del numero primario seleziona una funzione appropriata per la gestione di quel tipo di dispositivo, passando praticamente gli stessi argomenti già ricevuti.

Va osservato il caso particolare dei dispositivi '**DEV_KMEM_...**'. In un sistema operativo Unix comune, attraverso ciò che fa capo al file di dispositivo '`/dev/kmem`', si ha la possibilità di accedere all'immagine in memoria del kernel, lasciando a un programma con privilegi adeguati la facoltà di interpretare i simboli che consentono di individuare i dati esistenti. Nel caso di os16, non ci sono simboli nel risultato della compilazione, quindi non è possibile ricostruire la collocazione dei dati. Per questa ragione, le informazioni che devono essere pubblicate, vengono controllate attraverso un dispositivo specifico. Quindi, il dispositivo '**DEV_KMEM_PS**' consente di leggere la tabella dei processi, '**DEV_KMEM_MMAP**' consente di leggere la mappa della memoria, e così vale anche per altre tabelle.

Per quanto riguarda la gestione dei terminali, attraverso la funzione ***dev_tty()***, quando un processo vuole leggere dal terminale, ma non risulta disponibile un carattere, questo viene messo in pausa, in attesa di un evento legato ai terminali.

os16, non disponendo di un sistema di trattenimento dei dati in memoria (*cache*), esegue le operazioni di lettura e scrittura dei dispositivi in modo immediato. Per questo motivo, la distinzione tra file di dispositivo a blocchi e a caratteri, rimane puramente estetica, ovvero priva di un'utilità concreta.

Figura u147.1. Interdipendenza tra la funzione ***dev_io()*** e le altre. I collegamenti con le funzioni ***major()*** e ***minor()*** sono omesse.

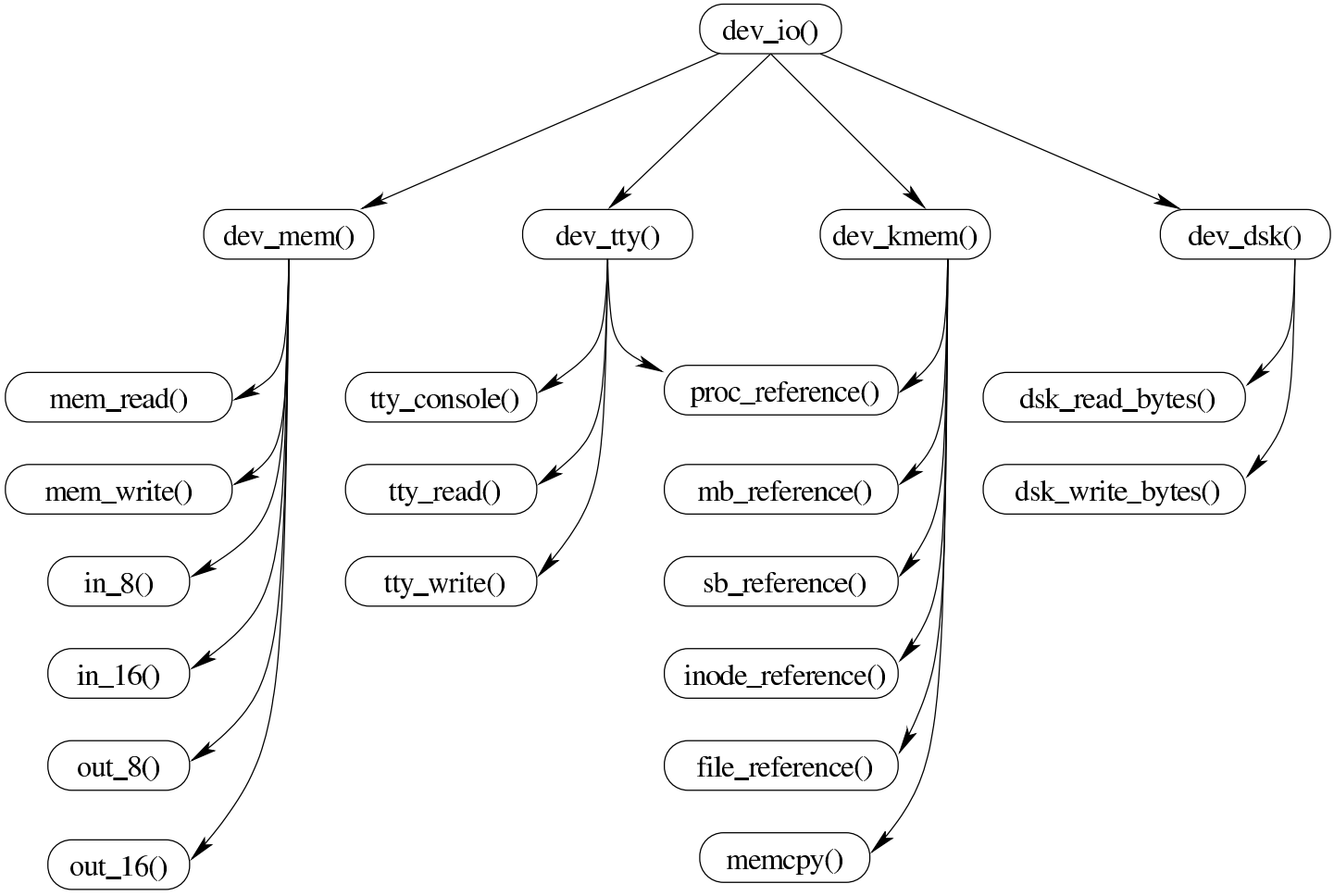
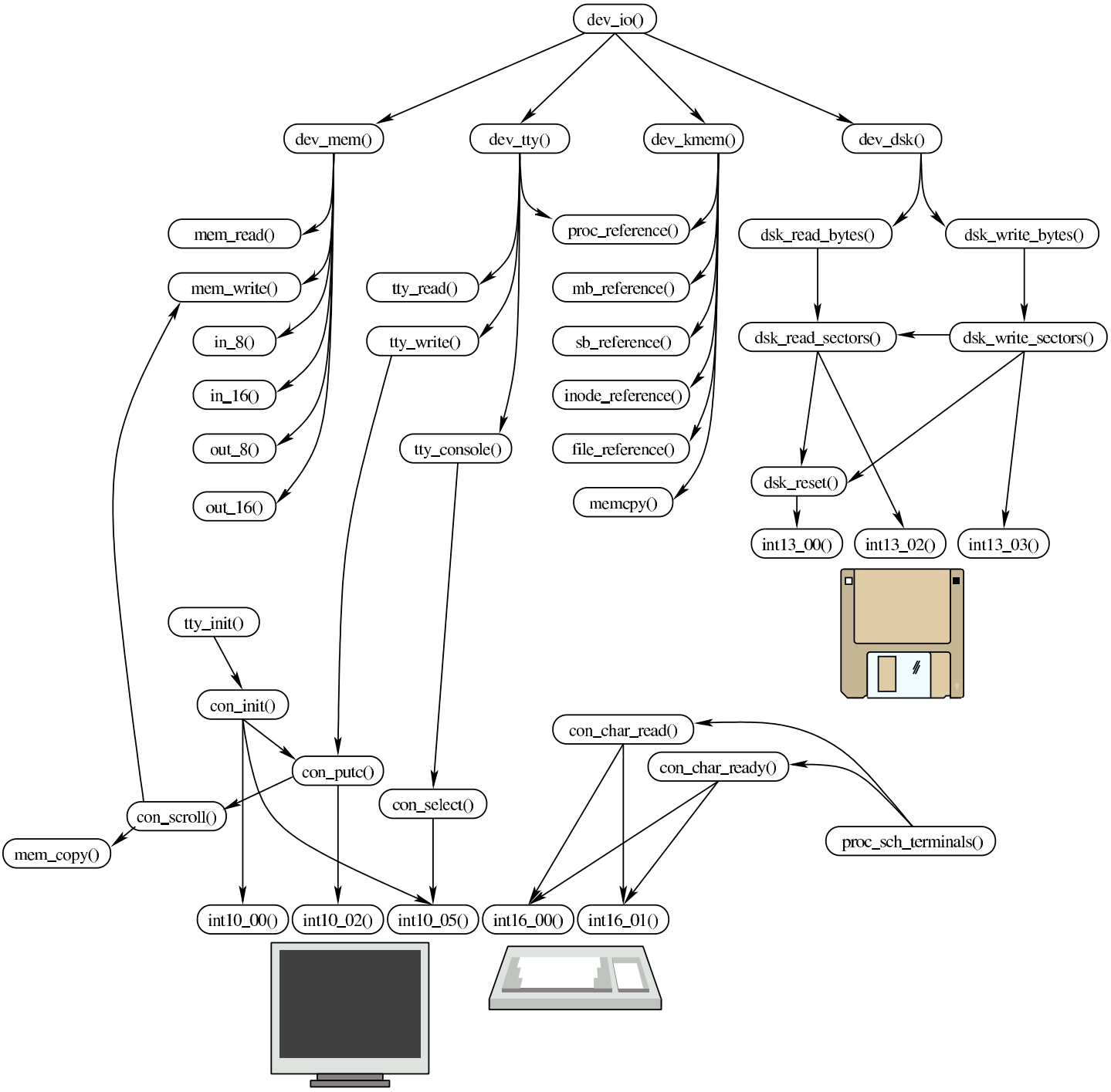


Figura u147.2. Schema più ampio delle dipendenze che hanno origine dalla funzione ***dev_io()***.



Numero primario e numero secondario

«

I dispositivi, secondo la tradizione dei sistemi Unix, sono rappresentati dal punto di vista logico attraverso un numero intero, senza segno, a 16 bit. Tuttavia, per organizzare questa numerazione in modo ordinato, tale numero viene diviso in due parti: la prima parte, nota come *major*, ovvero «numero primario», si utilizza per individuare il tipo di dispositivo; la seconda, nota come *minor*, ovvero «numero secondario», si utilizza per individuare precisamente il dispositivo, nell'ambito del tipo a cui appartiene.

In pratica, il numero complessivo a 16 bit si divide in due, dove gli 8 bit più significativi individuano il numero primario, mentre quelli meno significativi danno il numero secondario. L'esempio seguente si riferisce al dispositivo che genera il valore zero, il quale appartiene al gruppo dei dispositivi relativi alla memoria:

`DEV_MEM_MAJOR`	01_{16}
`DEV_ZERO`	0104_{16}

In questo caso, il valore che rappresenta complessivamente il dispositivo è 0104_{16} (pari a 260_{10}), ma si compone di numero primario 01_{16} e di numero secondario 04_{16} (che coincidono nella rappresentazione in base dieci). Per estrarre il numero primario si deve dividere il numero complessivo per 256 (0100_{16}), trattenendo soltanto il risultato intero; per filtrare il numero secondario si può fare la stessa divisione, ma trattenendo soltanto il resto della stessa. Al contrario, per produrre il numero del dispositivo, partendo dai numeri primario e secondario separati, occorre moltiplicare il numero primario per

256, sommando poi il risultato al numero secondario.

Dispositivi previsti

L'astrazione della gestione dei dispositivi, consente di trattare tutti i componenti che hanno a che fare con ingresso e uscita di dati, in modo sostanzialmente omogeneo; tuttavia, le caratteristiche effettive di tali componenti può comportare delle limitazioni o delle peculiarità. Ci sono alcune questioni fondamentali da considerare: un tipo di dispositivo potrebbe consentire l'accesso in un solo verso (lettura o scrittura); l'accesso al dispositivo potrebbe essere ammesso solo in modo sequenziale, rendendo inutile l'indicazione di un indirizzo; la dimensione dell'informazione da trasferire potrebbe assumere un significato differente rispetto a quello comune.

Tabella u147.4. Classificazione dei dispositivi di os16.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_MEM`	r/w	diretto	Permette l'accesso alla memoria, in modo indiscriminato, perché os16 non offre alcun tipo di protezione al riguardo.
`DEV_NULL`	r/w	nessuno	Consente la lettura e la scrittura, ma non si legge e non si scrive alcunché.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_PORT`	r/w	sequenziale	Consente di leggere e scrivere da o verso una porta di I/O, individuata attraverso l'indirizzo di accesso (l'indirizzo, o meglio lo scostamento, viene trattato come la porta a cui si vuole accedere). Tuttavia, la dimensione dell'informazione da trasferire è valida solo se si tratta di uno o di due byte: per la dimensione di un byte si usano le funzioni ***in_8()*** e ***out_8()***; per due byte si usano le funzioni ***in_16()*** e ***out_16()***. Per dimensioni differenti la lettura o la scrittura non ha effetto.
`DEV_ZERO`	r	sequenziale	Consente solo la lettura di valori a zero (zero inteso in senso binario).
`DEV_TTY`	r/w	sequenziale	Rappresenta il terminale virtuale del processo attivo.
`DEV_DSK`***n***	r/w	diretto	Rappresenta l'unità a dischi ***n***. os16 non gestisce le partizioni.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_KMEM_PS`	r	diretto	Rappresenta la tabella contenente le informazioni sui processi. L'indirizzo di accesso indica il numero del processo di partenza; la dimensione da leggere dovrebbe essere abbastanza grande da contenere un processo, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_KMEM_MMP`	r	sequenziale	Rappresenta la mappa della memoria, alla quale si può accedere solo dal suo principio. In pratica, l'indirizzo di accesso viene ignorato, mentre conta solo la quantità di byte richiesta.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_KMEM_SB`	r	diretto	Rappresenta la tabella dei super blocchi (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il super blocco; la dimensione richiesta dovrebbe essere abbastanza grande da contenere un super blocco, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_KMEM_INODE`	r	diretto	Rappresenta la tabella degli inode (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare l'inode; la dimensione richiesta dovrebbe essere abbastanza grande da contenere un inode, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_KMEM_FILE`	r	diretto	Rappresenta la tabella dei file (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il file; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di un file, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
`DEV_CONSOLE`	r/w	sequenziale	Legge o scrive relativamente alla console attiva la quantità di byte richiesta, ignorando l'indirizzo di accesso.
`DEV_CONSOLE`***n***	r/w	sequenziale	Legge o scrive relativamente alla console ***n*** la quantità di byte richiesta, ignorando l'indirizzo di accesso.

Gestione del file system

«

File «kernel/fs/sb_...» 3106

File «kernel/fs/zone_...» 3111

File «kernel/fs/inode_...» 3113

Fasi dell'innesto di un file system 3124

File «kernel/fs/file_...» 3126

Descrittori di file 3129

File «kernel/fs/path_...» 3131

File «kernel/fs/fd_...» 3136

fd_chmod() 3137 fd_chown() 3137 fd_close() 3137 fd_dup() 3137 fd_dup2() 3137 fd_fcntl() 3137 fd_lseek() 3137 fd_open() 3137 fd_read() 3137 fd_reference() 3137 fd_stat() 3137 fd_write() 3137 file_reference() 3128 file_stdio_dev_make() 3128 file_t 3126 fs.h 3105 inode_alloc() 3116 inode_check() 3116 inode_dir_empty() 3116 inode_file_read() 3116 inode_file_write() 3116 inode_fzones_read() 3116 inode_get() 3116 inode_put() 3116 inode_reference() 3116 inode_save() 3116 inode_t 3113 inode_truncate() 3116 inode_zone() 3116 path_chdir() 3132 path_chmod() 3132 path_chown() 3132 path_device() 3132 path_fix() 3131 path_full() 3131 path_inode() 3132 path_inode_link() 3132 path_link() 3132 path_mkdir() 3132 path_mknod() 3132 path_mount()

3132 path_stat() 3132 path_umount() 3132 path_unlink() 3132 sb_inode_status() 3109 sb_mount() 3109 sb_reference() 3109 sb_save() 3109 sb_t 3106 sb_zone_status() 3109 zone_alloc() 3112 zone_free() 3112 zone_read() 3112 zone_write() 3112

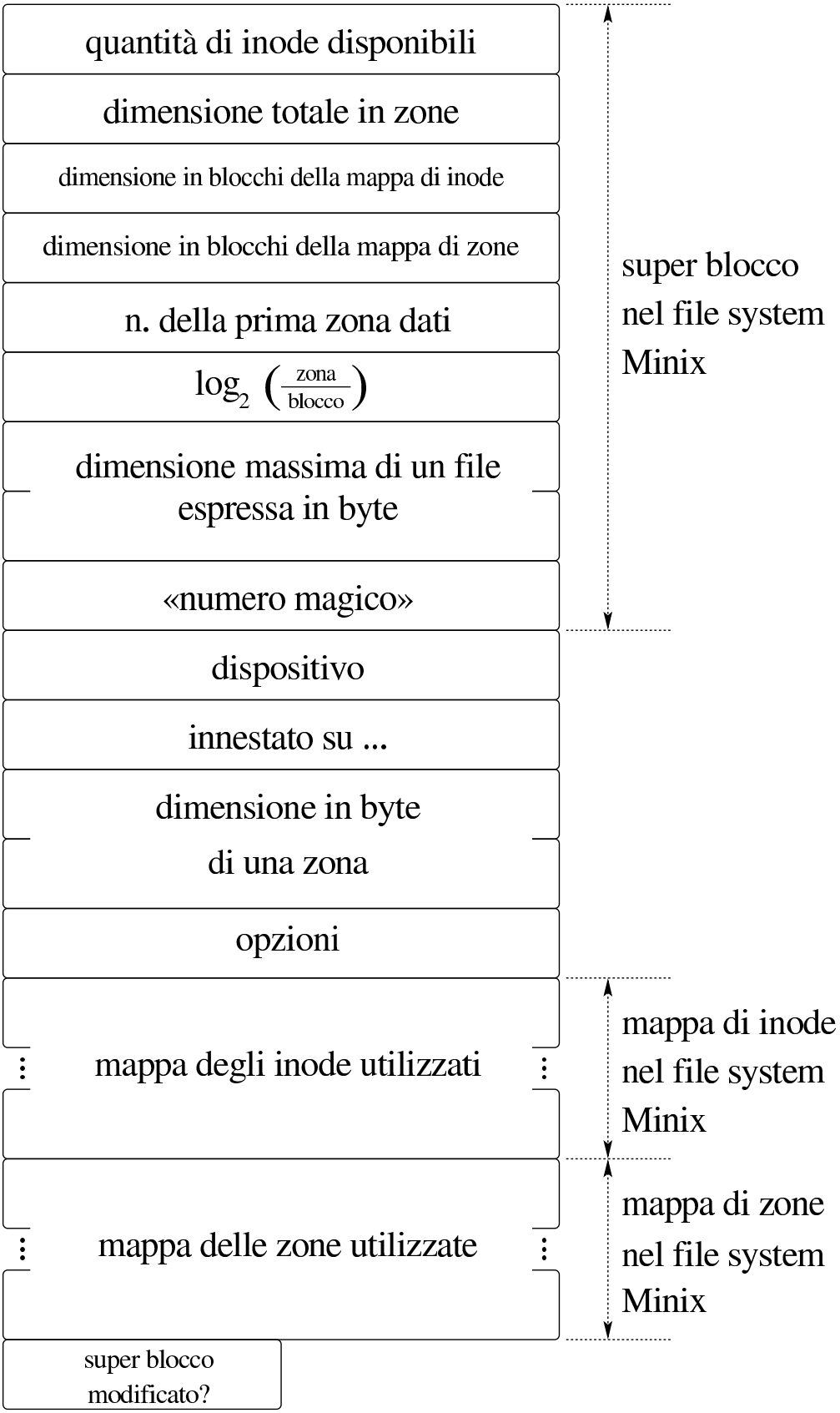
La gestione del file system è suddivisa in diversi file contenuti nella directory ‘kernel/fs/’, facenti capo al file di intestazione ‘kernel/fs.h’.

File «kernel/fs/sb_...»

«

I file ‘kernel/fs/sb_...’ descrivono le funzioni per la gestione dei super blocchi, distinguibili perché iniziano tutte con il prefisso ‘**sb_**’. Tra questi file si dichiara l’array ***sb_table[]***, il quale rappresenta una tabella le cui righe sono rappresentate da elementi di tipo ‘**sb_t**’ (il tipo ‘**sb_t**’ è definito nel file ‘kernel/fs.h’). Per uniformare l’accesso alla tabella, la funzione ***sb_reference()*** permette di ottenere il puntatore a un elemento dell’array ***sb_table[]***, specificando il numero del dispositivo cercato.

Figura u148.1. Struttura del tipo '**`sb_t`**', corrispondente agli elementi dell'array ***sb_table[]***.



Listato u148.2. Struttura del tipo '**sb_t**', corrispondente agli elementi dell'array ***sb_table[]***.

```
typedef struct sb        sb_t;

struct sb {
  uint16_t  inodes;
  uint16_t  zones;
  uint16_t  map_inode_blocks;
  uint16_t  map_zone_blocks;
  uint16_t  first_data_zone;
  uint16_t  log2_size_zone;
  uint32_t  max_file_size;
  uint16_t  magic_number;
  //------------------------
  dev_t     device;
  inode_t  *inode_mounted_on;
  blksize_t blksize;
  int       options;
  uint16_t  map_inode[SB_MAP_INODE_SIZE];
  uint16_t  map_zone[SB_MAP_ZONE_SIZE];
  char      changed;
};
```

Il super blocco rappresentato dal tipo '**sb_t**' include anche le mappe delle zone e degli inode impegnati. Queste mappe hanno una dimensione fissa in memoria, mentre nel file system reale possono essere di dimensione minore. La tabella di super blocchi, contiene le informazioni dei dispositivi di memorizzazione innestati nel sistema. L'innesto si concretizza nel riferimento a un inode, contenuto nella tabella degli inode (descritta in un altro capitolo), il quale rappresenta la directory di un'altra unità, su cui tale innesto è avvenuto. Naturalmente, l'innesto del file system principale rappresenta un caso particolare.

Tabella u148.3. Funzioni per la gestione dei dispositivi di memorizzazione di massa, a livello di super blocco, definite nei file 'kernel/fs/sb_...'.

Funzione	Descrizione
`sb_t *sb_reference (dev_t` ***device*** `);`	Restituisce il riferimento a un elemento della tabella dei super blocchi, in base al numero del dispositivo di memorizzazione. Se il dispositivo cercato non risulta già innestato, si ottiene il puntatore nullo; se si chiede il dispositivo zero, si ottiene il puntatore al primo elemento della tabella.

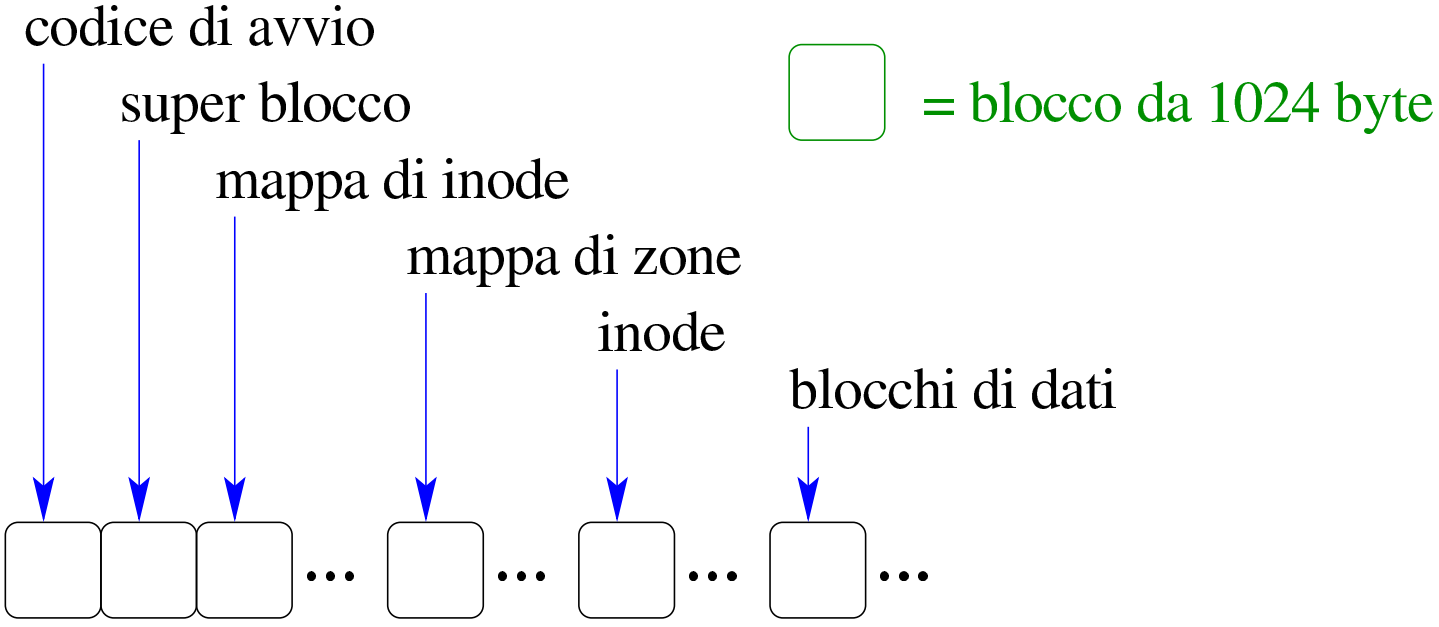
Funzione	Descrizione
`sb_t *sb_mount (dev_t` ***device*****,** `inode_t **`***inode_mnt*****,** `int` ***options*****);**	Innesta il dispositivo rappresentato numericamente dal primo parametro, sulla directory corrispondente all'inode a cui punta il secondo parametro, con le opzioni del terzo parametro. Quando si tratta del primo innesto del file system principale, la directory è quella dello stesso file system, pertanto, in tal caso, ****inode_mnt*** è inizialmente un puntatore nullo e deve essere modificato dalla funzione stessa.
`int sb_save (sb_t *`***sb*****);**	Salva il super blocco nella sua unità di memorizzazione, se questo risulta modificato. In questo caso, il super blocco include anche le mappe degli inode e delle zone.
`int sb_zone_status (sb_t *`***sb*****,** `zno_t` ***zone*****);**	Restituisce uno se la zona rappresentata dal secondo parametro è impegnata nel super blocco a cui si riferisce il primo parametro; diversamente restituisce zero.

Funzione	Descrizione
`int sb_inode_status (sb_t *sb, ino_t ino);`	Restituisce uno se l'inode rappresentato dal secondo parametro è impegnato nel super blocco a cui si riferisce il primo parametro; diversamente restituisce zero.

File «kernel/fs/zone_...»

«

Nel file system Minix 1, si distinguono i concetti di blocco e zona di dati, con il vincolo che la zona ha una dimensione multipla del blocco. Il contenuto del file system, dopo tutte le informazioni amministrative, è organizzato in zone; in altri termini, i blocchi di dati si raggiungono in qualità di zone.



La zona rimane comunque un tipo di blocco, potenzialmente più grande (ma sempre multiplo) del blocco vero e proprio, che si numera a partire dall'inizio dello spazio disponibile, con la differenza che

è utile solo per raggiungere i blocchi di dati. Nel super blocco del file system si trova l'informazione del numero della prima zona che contiene dati, in modo da non dover ricalcolare questa informazione ogni volta.

I file '`kernel/fs/zone_...`' descrivono le funzioni per la gestione del file system a zone.

Tabella u148.5. Funzioni per la gestione delle zone, definite nei file '`kernel/fs/zone_...`'.

Funzione	Descrizione
`zno_t zone_alloc (sb_t *`***sb***`);`	Alloca una zona, restituendo il numero della stessa. In pratica, cerca la prima zona libera nel file system a cui si riferisce il super blocco ****sb*** e la segna come impegnata, restituendone il numero.
`int zone_free (sb_t *`***sb***`,` `zno_t` ***zone***`);`	Libera una zona, impegnata precedentemente.

`int zone_read (sb_t *`***sb***`,`
`zno_t` ***zone***`,`
`void *`***buffer***`);`
	Legge il contenuto di una zona, memorizzandolo a partire dalla posizione di memoria rappresentato da ***buffer***.

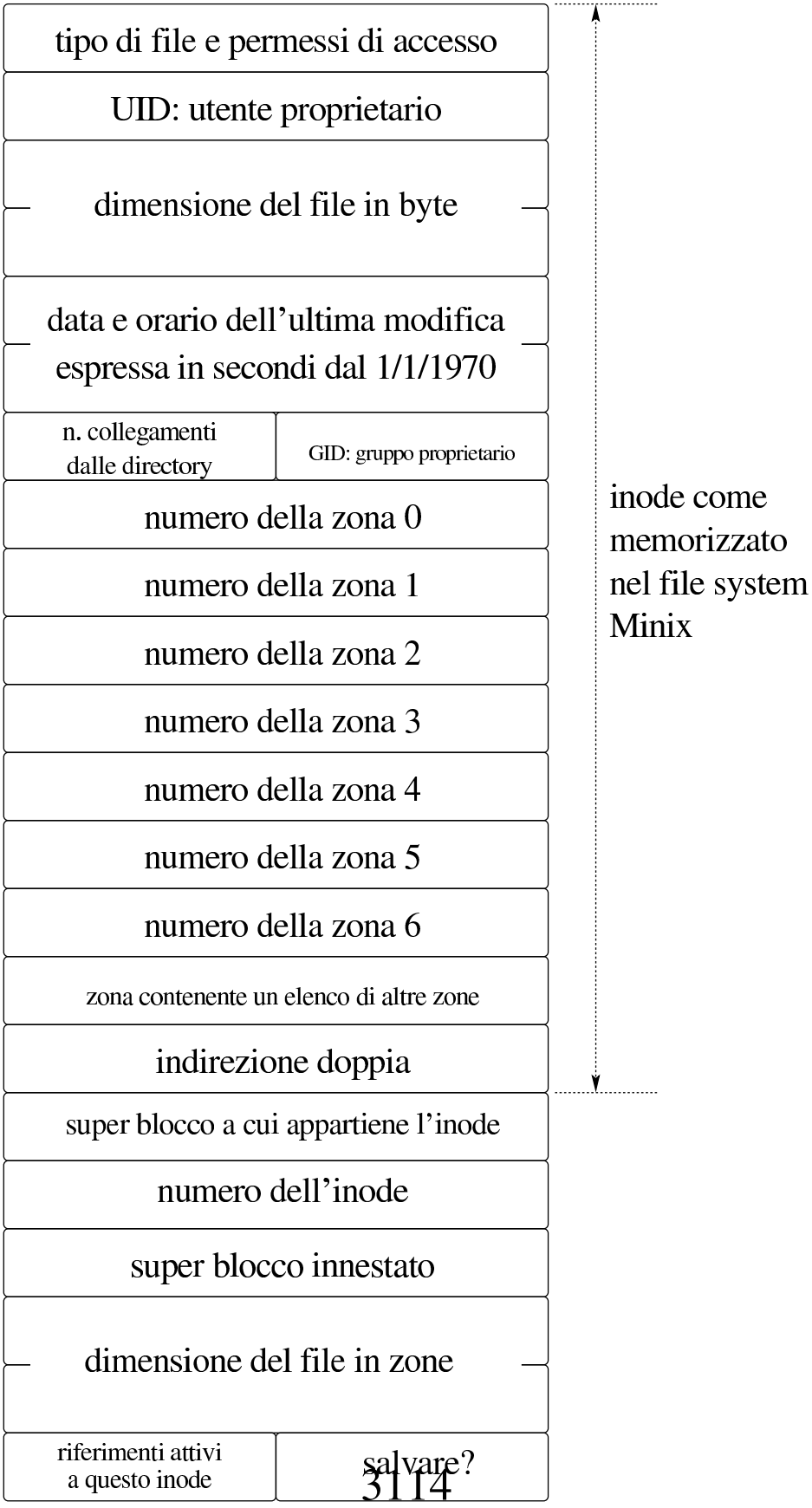
`int zone_write (sb_t *`***sb***`,`
`zno_t` ***zone***`,`
`void *`***buffer***`);`
	Sovrascrive una zona, utilizzando il contenuto della memoria a partire dalla posizione rappresentata da ***buffer***.

File «kernel/fs/inode_...»

«

I file ‘`kernel/fs/inode_...`’ descrivono le funzioni per la gestione dei file, in forma di inode. In uno di questi file viene dichiarata la tabella degli inode in uso nel sistema, rappresentata dall’array ***inode_table[]*** e per individuare un certo elemento dell’array si usa preferibilmente la funzione ***inode_reference()***. Gli elementi della tabella degli inode sono di tipo ‘**`inode_t`**’ (definito nel file ‘`kernel/fs.h`’); una voce della tabella rappresenta un inode utilizzato se il campo dei riferimenti (***references***) ha un valore maggiore di zero.

Figura u148.6. Struttura del tipo '**`inode_t`**', corrispondente agli elementi dell'array ***inode_table[]***.



Listato u148.7. Struttura del tipo '**`inode_t`**', corrispondente agli elementi dell'array ***inode_table[]***.

```
<verbatimpre width="60">
<![CDATA[
typedef struct inode     inode_t;

struct inode {
    mode_t         mode;
    uid_t          uid;
    ssize_t        size;
    time_t         time;
    uint8_t        gid;
    uint8_t        links;
    zno_t          direct[7];
    zno_t          indirect1;
    zno_t          indirect2;
    //------------------------
    sb_t          *sb;
    ino_t          ino;
    sb_t          *sb_attached;
    blkcnt_t       blkcnt;
    unsigned char  references;
    char           changed;
};
```

Figura u148.8. Collegamento tra la tabella degli inode e quella dei super blocchi.

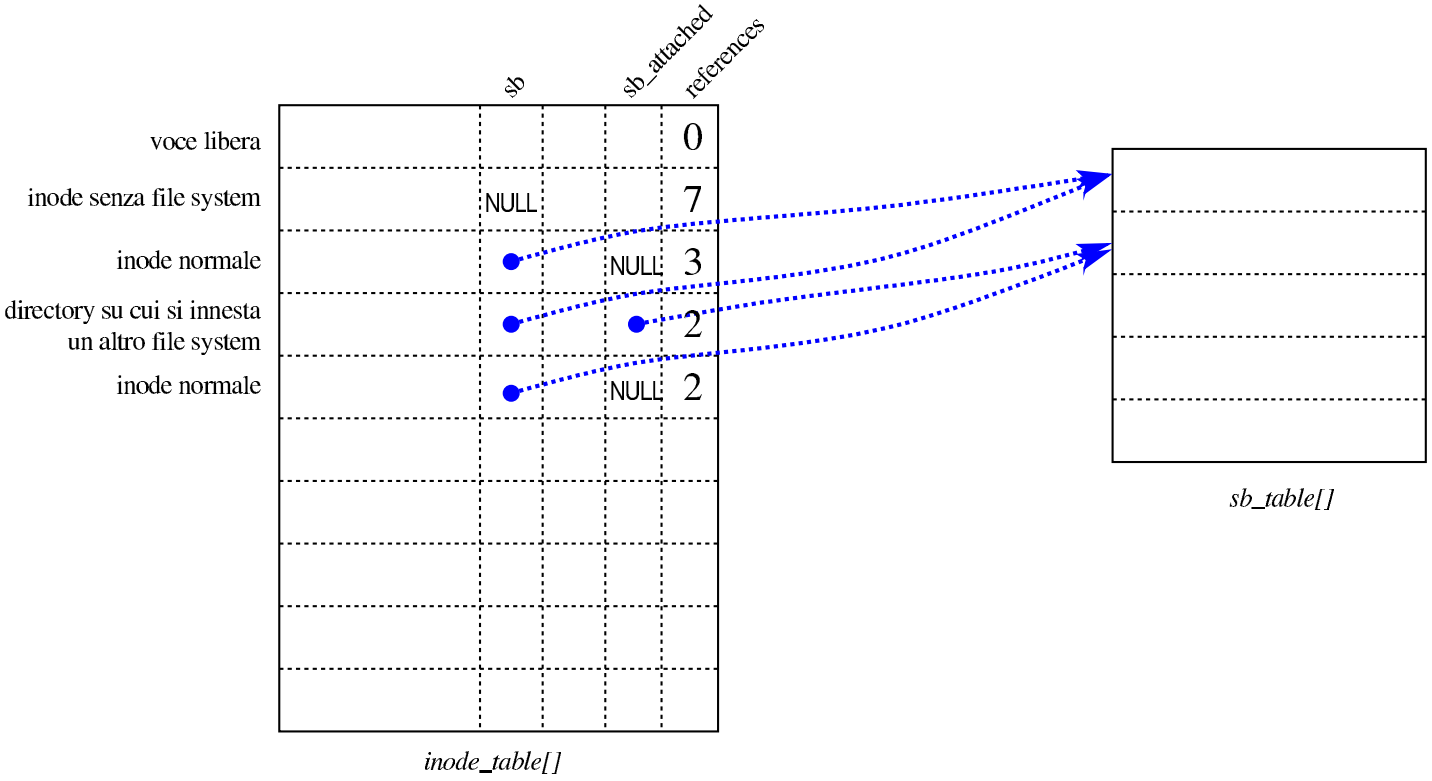


Tabella u148.9. Funzioni per la gestione dei file in forma di inode, definite nei file '`kernel/fs/inode_...`'.

Funzione	Descrizione

`inode_t *`
`inode_reference (dev_t` ***device*****`,`**
`ino_t` ***ino*****`);`
	Restituisce il puntatore a un inode, rappresentato in pratica da un elemento dell'array ***inode_table[]***, corrispondente a quello con il numero di dispositivo e di inode indicati come argomenti. Se entrambi gli argomenti sono a zero, si ottiene il puntatore al primo elemento; se entrambi i valori sono pari a -1, si ottiene il puntatore al primo elemento libero; se viene indicato il dispositivo zero e l'inode numero uno, si ottiene il puntatore all'elemento corrispondente alla directory radice del file system principale.

Funzione	Descrizione
`inode_t *` `inode_alloc (dev_t` ***device*****,** `mode_t` ***mode*****,** `uid_t` ***uid*****);**	La funzione ***inode_alloc()*** cerca un inode libero nel file system del dispositivo indicato, quindi lo alloca (lo segna come utilizzato) e lo modifica aggiornando il tipo e la modalità dei permessi, oltre al proprietario del file. Se la funzione riesce nel suo intento, restituisce il puntatore all'inode in memoria, il quale rimane così aperto e disponibile per ulteriori elaborazioni.

Funzione	Descrizione
`inode_t *` `inode_get (dev_t` ***device*****,** `ino_t` ***ino*****);**	Restituisce il puntatore all'inode rappresentato dal numero di dispositivo e di inode, indicati come argomenti. Se l'inode è già presente nella tabella degli inode, la cosa si risolve nell'incremento di una unità del numero dei riferimenti di tale inode; se invece l'inode non è ancora presente, questo viene caricato dal suo file system nella tabella e gli viene attribuito inizialmente un riferimento attivo.

Funzione	Descrizione
`int inode_put (inode_t *`***inode***`);`	Rilascia un inode che non serve più. Ciò comporta la riduzione del contatore dei riferimenti nella tabella degli inode, tenendo conto che se tale valore raggiunge lo zero, si provvede anche al suo salvataggio nel file system (ammesso che l'inode della tabella risulti modificato, rispetto alla versione presente nel file system). La funzione restituisce zero in caso di successo, oppure -1 in caso contrario.
`int inode_save (inode_t *`***inode***`);`	Salva l'inode nel file system, se questo risulta modificato.

Funzione	Descrizione

`blkcnt_t`
`inode_fzones_read (inode_t *`***`inode`**`,`
`zno_t `***`zone_start`**`,`
`void *`***`buffer`**`,`
`blkcnt_t `***`blkcnt`**`);`
	Legge da un file, identificato attraverso il puntatore all'inode (della tabella di inode), una certa quantità di zone, a partire da una certa zona relativa al file, mettendo il risultato della lettura a partire dalla posizione di memoria rappresentata da un puntatore generico. La funzione restituisce la quantità di zone lette con successo.

`ssize_t`
`inode_file_read (inode_t *`***`inode`**`,`
`off_t `***`offset`**`,`
`void *`***`buffer`**`,`
`size_t `***`count`**`,`
`int *`***`eof`**`);`
	Legge il contenuto di un file, individuato da un inode già caricato nella tabella relativa, aggiornando eventualmente una variabile contenente l'indicatore di fine file. La funzione restituisce la quantità di byte letti con successo, oppure il valore -1 in caso di problemi.

Funzione	Descrizione
`ssize_t inode_file_write (inode_t *`***inode***`, off_t `***offset***`, void *`***buffer***`, size_t `***count***`);`	Scrive una certa quantità di byte nel file individuato da un inode già caricato nella tabella relativa. La funzione restituisce la quantità di byte scritti effettivamente, oppure il valore −1 in caso di problemi.
`zno_t inode_zone (inode_t *`***inode***`, zno_t `***fzone***`, int `***write***`);`	Restituisce il numero di zona effettivo, corrispondente a un numero di zona relativo a un certo file di un certo inode. Se il parametro ***write*** è pari a zero, si intende che la zona deve esistere, quindi se questa non c'è, si ottiene semplicemente un valore pari a zero; se invece l'ultimo parametro è pari a uno, nel caso la zona cercata fosse attualmente mancante, verrebbe creata al volo nel file system.

Funzione	Descrizione
`int` `inode_truncate (inode_t *`***`inode`**`);`	Riduce la dimensione del file a cui si riferisce l'inode a zero. In pratica fa sì che le zone allocate del file siano liberate. La funzione restituisce zero se l'operazione si conclude con successo, oppure −1 in caso di problemi.

`int`
`inode_check (inode_t *`***`inode`**`,`
`mode_t` ***`type`**`,`
`int` ***`perm`**`,`
`uid_t` ***`uid`**`);`
	Verifica che l'inode sia di un certo tipo e abbia i permessi di accesso necessari a un certo utente. Nel parametro ***type*** si possono indicare più tipi validi. La funzione restituisce zero in caso di successo, ovvero di compatibilità, mentre restituisce −1 se il tipo o i permessi non sono adatti.

`int` `inode_dir_empty (inode_t *`***`inode`**`);`	Verifica se la directory a cui si riferisce l'inode è effettivamente una directory ed è vuota, nel qual caso restituisce il valore uno, altrimenti restituisce zero.

Fasi dell'innesto di un file system

«

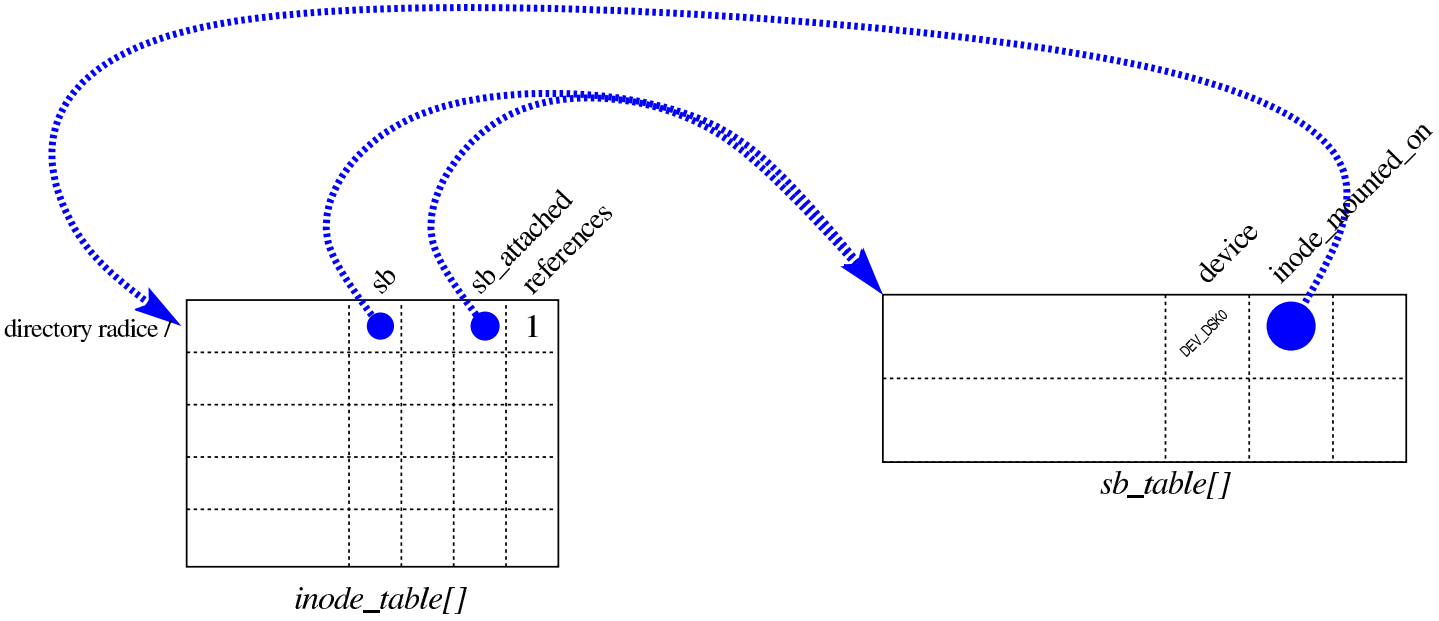
L'innesto e il distacco di un file system, coinvolge simultaneamente la tabella dei super blocchi e quella degli inode. Si distinguono due situazioni fondamentali: l'innesto del file system principale e quello di un file system ulteriore.

Quando si tratta dell'innesto del file system principale, la tabella dei super blocchi è priva di voci e quella degli inode non contiene riferimenti a file system. La funzione ***sb_mount()*** viene chiamata indicando, come riferimento all'inode di innesto, il puntatore a una variabile puntatore contenente il valore nullo:

```
…
    inode_t *inode;
    sb_t    *sb;
    …
    inode = NULL;
    sb = sb_mount (DEV_DSK0, &inode, MOUNT_DEFAULT);
    …
```

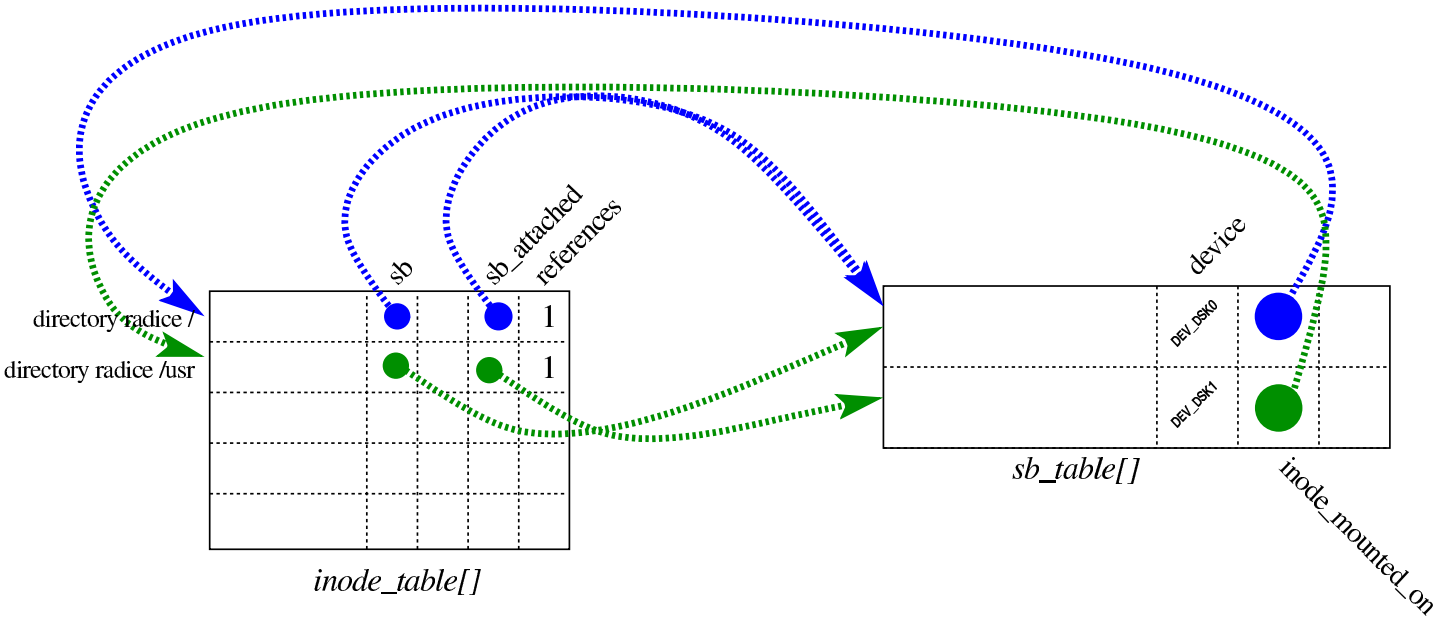
La funzione ***sb_mount()*** carica il super blocco nella tabella relativa, ma trovando il riferimento all'inode di innesto nullo, provvede a caricare l'inode della directory radice dello stesso dispositivo, creando un collegamento incrociato tra le tabelle dei super blocchi e degli inode, come si vede nella figura successiva.

Figura u148.11. Collegamento tra la tabella degli inode e quella dei super blocchi, quando si innesta il file system principale.



Per innestare un altro file system, occorre prima disporre dell'inode di una directory (appropriata) nella tabella degli inode, quindi si può caricare il super blocco del nuovo file system, creando il collegamento tra directory e file system innestato.

Figura u148.12. Innesto di un file system nella directory '`/usr/`'.



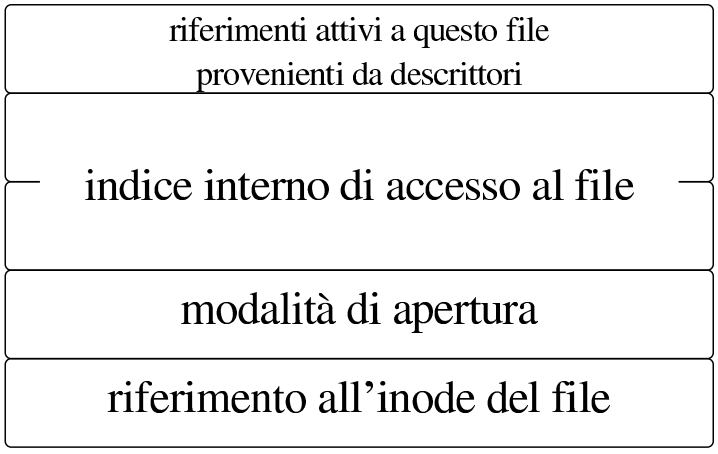
File «kernel/fs/file_...»

«

I file ‘`kernel/fs/file_...`’ descrivono le funzioni per la gestione della tabella dei file, la quale si collega a sua volta a quella degli inode. In realtà, le funzioni di questo gruppo sono in numero molto limitato, perché l’intervento nella tabella dei file avviene prevalentemente per opera di funzioni che gestiscono i descrittori.

La tabella dei file è rappresentata dall’array ***file_table[]*** e per individuare un certo elemento dell’array si usa preferibilmente la funzione ***file_reference()***. Gli elementi della tabella dei file sono di tipo ‘**`file_t`**’ (definito nel file ‘`kernel/fs.h`’); una voce della tabella rappresenta un file aperto se il campo dei riferimenti (***references***) ha un valore maggiore di zero.

Figura u148.13. Struttura del tipo ‘**`file_t`**’, corrispondente agli elementi dell’array ***file_table[]***.




```
typedef struct file file_t;

struct file {
  int       references;
  off_t     offset;
  int       oflags;
  inode_t *inode;
};
```

Nel membro ***oflags*** si annotano esclusivamente opzioni relative alla modalità di apertura del file: lettura, scrittura o entrambe; pertanto si possono usare le macro-variabili ***O_RDONLY***, ***O_WRONLY*** e ***O_RDWR***, come dichiarato nel file di intestazione ‘`lib/fcntl.h`’. Il membro ***offset*** rappresenta l’indice interno di accesso al file, per l’operazione successiva di lettura o scrittura al suo interno. Il

membro ***references*** è un contatore dei riferimenti a questa tabella, da parte di descrittori di file.

La tabella dei file si collega a quella degli inode, attraverso il membro ***inode***. Più voci della tabella dei file possono riferirsi allo stesso inode, perché hanno modalità di accesso differenti, oppure soltanto per poter distinguere l'indice interno di lettura e scrittura. Va osservato che le voci della tabella di inode potrebbero essere usate direttamente e non avere elementi corrispondenti nella tabella dei file.

Figura u148.14. Collegamento tra la tabella dei file e quella degli inode.

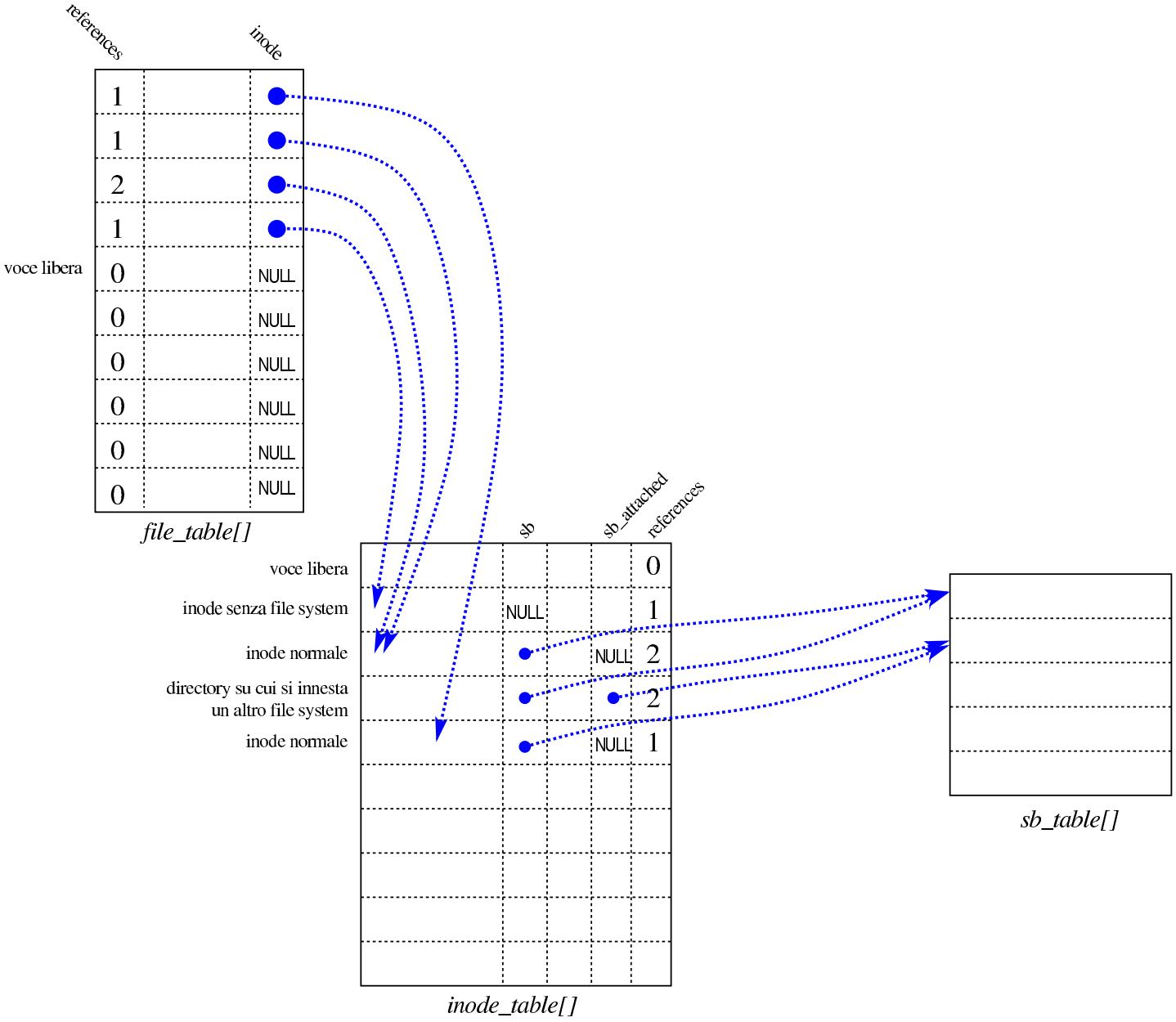


Tabella u148.15. Funzioni fatte esclusivamente per la gestione della tabella dei file ***file_table[]***.

Funzione	Descrizione
`file_t *` `file_reference (int` ***fno***`);`	Restituisce il puntatore all'elemento ***fno***-esimo della tabella dei file. Se ***fno*** è un valore negativo, viene restituito il puntatore a una voce libera della tabella.

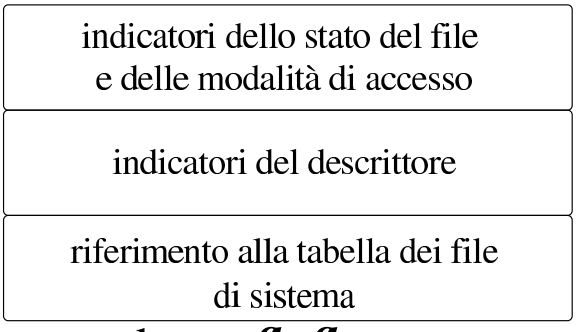
`file_t *`
`file_stdio_dev_make (dev_t` ***device***`,`
`mode_t` ***mode***`,`
`int` ***oflags***`);`
	Crea una voce per l'accesso a un file di dispositivo standard di input-output, restituendo il puntatore alla voce stessa.

Descrittori di file

«

Le tabelle di super blocchi, inode e file, riguardano il sistema nel complesso. Tuttavia, l'accesso normale ai file avviene attraverso il concetto di «descrittore», il quale è un file aperto da un certo processo elaborativo. Nel file '`kernel/fs.h`' si trova la dichiarazione e descrizione del tipo derivato '**`fd_t`**', usato per costruire una tabella di descrittori, ma tale tabella non fa parte della gestione del file system, bensì è incorporata nella tabella dei processi elaborativi. Pertanto, ogni processo ha una propria tabella di descrittori di file.

Figura u148.16. Struttura del tipo '**fd_t**', con cui si costituiscono gli elementi delle tabelle dei descrittori di file, una per ogni processo.

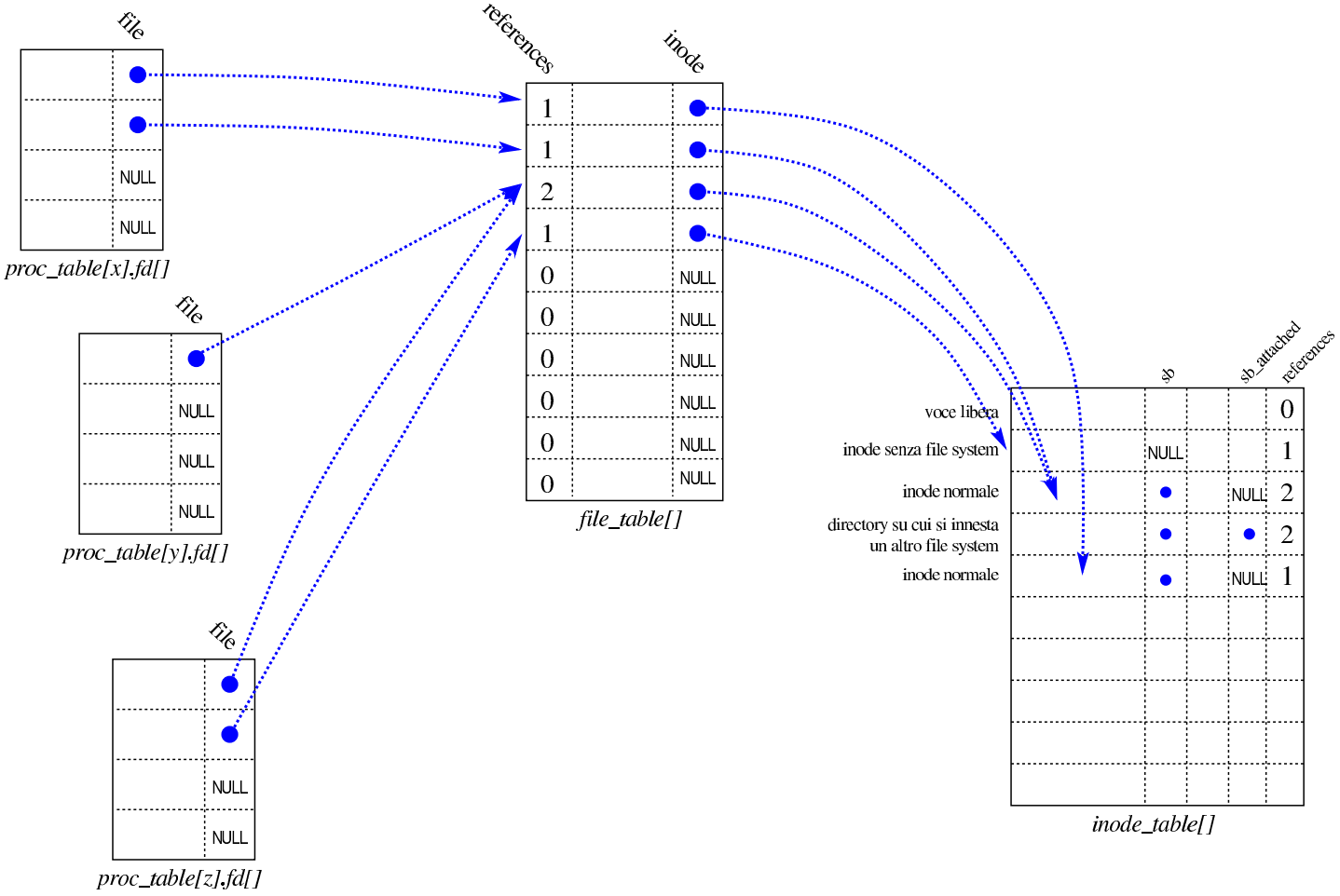



```
typedef struct fd fd_t;
struct fd {
    int     fl_flags;
    int     fd_flags;
    file_t *file;
};
```

Il membro ***fl_flags*** consente di annotare indicatori del tipo '**O_RDONLY**', '**O_WRONLY**', '**O_RDWR**', '**O_CREAT**', '**O_EXCL**', '**O_NOCTTY**', '**O_TRUNC**' e '**O_APPEND**', come dichiarato nella libreria standard, nel file di intestazione '`lib/fcntl.h`'. Tali indicatori si combinano assieme con l'operatore binario OR. Altri tipi di opzione che sarebbero previsti nel file '`lib/fcntl.h`', sono privi di effetto nella gestione del file system di os16.

Il membro ***fd_flags*** serve a contenere, eventualmente, l'opzione '**FD_CLOEXEC**', definita nel file '`lib/fcntl.h`'. Non sono previste altre opzioni di questo tipo.

Figura u148.17. Collegamento tra le tabelle dei descrittori e la tabella complessiva dei file. La tabella ***proc_table[x].fd[]*** rappresenta i descrittori di file del processo elaborativo ***x***.



File «kernel/fs/path_...»

I file '`kernel/fs/path_...`' descrivono le funzioni che fanno riferimento a file o directory attraverso una stringa che ne descrive il percorso.

Tabella u148.18. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, senza indicazioni sul processo elaborativo.

Funzione	Descrizione
`int path_fix (char *path);`	Verifica il percorso indicato semplificandolo, quindi sovrascrive il percorso originario con quello riveduto e corretto. Un percorso assoluto rimane assoluto; un percorso relativo rimane relativo, mancando qualunque indicazione sulla directory corrente.

`int path_full (const char *path,`
`const char *path_cwd,`
`char *full_path);`
	Ricostruisce un percorso assoluto, usando come riferimento la directory corrente indicata in ***path_cwd***, salvandolo in ***path_full***.

Tabella u148.19. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, tenendo conto del processo elaborativo per conto del quale si svolge l'operazione. Del processo elaborativo si considera soprattutto l'identità efficace, per conoscerne i privilegi e determinare se è data effettivamente la facoltà di eseguire l'azione richiesta.

Funzione	Descrizione
`int path_chdir (pid_t pid,` `const char *path);`	Cambia la directory corrente, utilizzando il nuovo percorso indicato. È l'equivalente della funzione standard ***chdir()*** (sezione u0.3).

Funzione	Descrizione
`int path_chmod (pid_t pid, const char *path, mode_t mode);`	Cambia la modalità di accesso al file indicato. È l'equivalente della funzione standard ***chmod()*** (sezione u0.4).
`int path_chown (pid_t pid, const char *path, uid_t uid, gid_t gid);`	Cambia l'utente e il gruppo proprietari del file (va però ricordato che os16 non considera i gruppi, anche se nel file system sono annotati). È l'equivalente della funzione standard ***chown()*** (sezione u0.5).
`dev_t path_device (pid_t pid, const char *path);`	Restituisce il numero del dispositivo di un file di dispositivo; pertanto, il percorso deve fare riferimento a un file di dispositivo, per poter ottenere un risultato valido.
`inode_t * path_inode (pid_t pid, const char *path);`	Apre l'inode del file indicato tramite il percorso, purché il processo ***pid*** abbia i permessi di accesso («x») alle directory che vi conducono. La funzione restituisce il puntatore all'inode aperto, oppure il puntatore nullo se non può eseguire l'operazione.

Funzione	Descrizione
`inode_t *path_inode_link (pid_t` ***pid***`, const char *`***path***`, inode_t *`***inode***`, mode_t` ***mode***`);`	Crea un collegamento fisico con il nome fornito in ***path***, riferito all'inode a cui punta ***inode***, ma se ***inode*** fosse un puntatore nullo, verrebbe semplicemente creato un file vuoto con un nuovo inode. Si richiede inoltre che il processo ***pid*** abbia i permessi di accesso per tutte le directory che portano al file da collegare e che nell'ultima ci sia anche il permesso di scrittura, dovendo intervenire su tale directory in questo modo. Se la funzione riesce nel proprio intento, restituisce il puntatore a ciò che descrive l'inode collegato o creato.
`int path_link (pid_t` ***pid***`, const char *`***path_old***`, const char *`***path_new***`);`	Crea un collegamento fisico. È l'equivalente della funzione standard ***link()*** (sezione u0.23).

Funzione	Descrizione
`int path_mkdir (pid_t` ***pid*****`,`** `const char *`***path*****`,`** `mode_t` ***mode*****`);`	Crea una directory, con la modalità dei permessi indicata. È l'equivalente della funzione standard ***mkdir()*** (sezione u0.25).
`int path_mknod (pid_t` ***pid*****`,`** `const char *`***path*****`,`** `mode_t` ***mode*****`,`** `dev_t` ***device*****`);`	Crea un file vuoto, con il tipo e i permessi specificati da ***mode***; se si tratta di un file di dispositivo, viene preso in considerazione anche il parametro ***device***, per specificare il numero primario e secondario dello stesso. Va osservato che con questa funzione è possibile creare una directory priva delle voci '.' e '..'. È l'equivalente della funzione standard ***mknod()*** (sezione u0.26).
`int path_stat (pid_t` ***pid*****`,`** `const char *`***path*****`,`** `struct stat *`***buffer*****`);`	Aggiorna la variabile strutturata a cui punta ***buffer***, con le informazioni sul file specificato. È l'equivalente della funzione standard ***stat()*** (sezione u0.36).

Funzione	Descrizione
`int path_unlink (pid_t` ***pid*****`,`** `const char *`***path***`);`	Cancella un file o una directory, purché questa sia vuota. È l'equivalente della funzione standard ***unlink()*** (sezione u0.42).

`int`
`path_mount (pid_t` ***pid*****`,`**
`const char *`***path_dev*****`,`**
`const char *`***path_mnt*****`,`**
`int` ***options***`);`
	Innesta il dispositivo corrispondente a ***path_dev***, nella directory ***path_mnt*** (tenendo conto della directory corrente del processo ***pid***), con le opzioni specificate, per conto dell'utente ***uid***. Le opzioni disponibili sono solo '**`MOUNT_DEFAULT`**' e '**`MOUNT_RO`**', come dichiarato nel file di intestazione '`lib/sys/os16.h`'.

`int`
`path_umount (pid_t` ***pid*****`,`**
`const char *`***path_mnt***`);`
	Stacca l'unità innestata nella directory indicata, purché nulla al suo interno sia attualmente in uso.

File «kernel/fs/fd_...»

«

I file '`kernel/fs/fd_...`' descrivono le funzioni che fanno riferimento a file o directory attraverso il numero di descrittore, riferito a sua volta a un certo processo elaborativo. Pertanto, il numero del processo e il numero del descrittore sono i primi due parametri obbligatori di tutte queste funzioni.

Tabella u148.20. Funzioni per la gestione dei file, a cui si fa riferimento attraverso il descrittore, relativamente a un certo processo elaborativo. La funzione ***fd_open()*** fa eccezione, in quanto apre un descrittore, ma per identificare il file non ancora aperto, ne richiede il percorso.

Funzione	Descrizione

`int fd_chmod (pid_t` ***pid***`,`
`int` ***fdn***`,`
`mode_t` ***mode***`);`
	Cambia la modalità dei permessi (solo gli ultimi 12 bit del parametro ***mode*** vengono considerati). È l'equivalente della funzione standard ***fchmod()*** (sezione u0.4).

`int fd_chown (pid_t` ***pid***`,`
`int` ***fdn***`,`
`uid_t` ***uid***`,`
`gid_t` ***gid***`);`
	Cambia la proprietà (utente e gruppo). È l'equivalente della funzione standard ***fchown()*** (sezione u0.5).

`int fd_close (pid_t` ***pid***`,` `int` ***fdn***`);`	Chiude il descrittore di file. È l'equivalente della funzione standard ***close()*** (sezione u0.7).

`int fd_dup (pid_t` ***pid***`,`
`int` ***fdn_old***`,`
`int` ***fdn_min***`);`
	Duplica il descrittore ***fdn_old***, creandone un altro con numero maggiore o uguale a ***fdn_min*** (viene scelto il primo libero a partire da ***fdn_num***). È l'equivalente della funzione standard ***dup()*** (sezione u0.8).

Funzione	Descrizione
`int fd_dup2 (pid_t pid, int fdn_old, int fdn_new);`	Duplica il descrittore ***fdn_old***, creandone un altro con numero ***fdn_new***. Se però ***fdn_new*** è già aperto, prima della duplicazione questo viene chiuso. È l'equivalente della funzione standard ***dup2()*** (sezione u0.8).
`int fd_fcntl (pid_t pid, int fdn, int cmd, int arg);`	Svolge il compito della funzione standard ***fcntl()*** (sezione u0.13).
`off_t fd_lseek (pid_t pid, int fdn, off_t offset, int whence);`	Riposiziona l'indice interno di accesso del descrittore di file. È l'equivalente della funzione standard ***lseek()*** (sezione u0.24).
`int fd_open (pid_t pid, const char *path, int oflags, mode_t mode);`	Apre un descrittore, fornendo però il percorso del file. È l'equivalente della funzione standard ***open()*** (sezione u0.28).
`ssize_t fd_read (pid_t pid, int fdn, void *buffer, size_t count, int *eof);`	Legge da un descrittore, aggiornando eventualmente la variabile ****eof*** in caso di fine del file. È l'equivalente della funzione standard ***read()*** (sezione u0.29).

Funzione	Descrizione
`fd_t *fd_reference (pid_t` ***pid*****,** `int *`***fdn***`);`	Produce il puntatore ai dati del descrittore ****fdn***. Se ****fdn*** è minore di zero, si ottiene il riferimento al primo descrittore libero, aggiornando anche ****fdn*** stesso.
`int fd_stat (pid_t` ***pid*****,** `int` ***fdn*****,** `struct stat *`***buffer***`);`	Svolge il compito della funzione standard ***fstat()*** (sezione u0.36).

`ssize_t`
`fd_write (pid_t` ***pid*****,**
`int` ***fdn*****,**
`const void *`***buffer*****,**
`size_t` ***count***`);`
	Scrive nel descrittore. È l'equivalente della funzione standard ***write()*** (sezione u0.44).

File «kernel/proc/_isr.s» e «kernel/proc/_ivt_load.s» 3142
Routine «isr_1C» 3148
Routine «isr_80» 3151
La tabella dei processi 3154
Chiamate di sistema 3161
File «kernel/proc/...» 3162
Funzione «proc_init()» 3163
Funzione «sysroutine()» 3164
Funzione «proc_scheduler()» 3167

`proc.h` 3141 `proc_init()` 3163 `proc_reference()` 3162 `proc_scheduler()` 3167 `proc_t` 3154 `sysroutine()` 3161 3164 `_isr.s` 3142 `_ivt_load.s` 3142

La gestione dei processi è raccolta nei file ‘`kernel/proc.h`’ e ‘`kernel/proc/...`’, dove il file ‘`kernel/proc/_isr.s`’, in particolare, contiene il codice attivato dalle interruzioni. Nella semplicità di os16, ci sono solo due interruzioni che vengono gestite: quella del temporizzatore il quale produce un impulso 18,2 volte al secondo, e quella causata dalle chiamate di sistema.

Con os16, quando un processo viene interrotto, per lo svolgimento del compito dell'interruzione, si passa sempre a utilizzare la pila dei dati del kernel. Per annotare la posizione in cui si trova l'indice della pila del kernel si usa la variabile ***_ksp***, accessibile anche dal codice in linguaggio C.

Il codice del kernel può essere interrotto dagli impulsi del temporizzatore, ma in tal caso non viene coinvolto lo schedulatore per lo scambio con un altro processo, così che dopo l'interruzione è sempre il kernel che continua a funzionare; pertanto, nella funzione ***main()*** è il kernel che cede volontariamente il controllo a un altro processo (ammesso che ci sia) con una chiamata di sistema nulla.

File «kernel/proc/_isr.s» e «kernel/proc/_ivt_load.s»

«

Listati i160.9.1 e i160.9.2.

Il file '`kernel/proc/_isr.s`' contiene il codice per la gestione delle interruzioni dei processi. Nella parte iniziale del file, vengono dichiarate delle variabili, alcune delle quali sono pubbliche e accessibili anche dal codice in C.

```
…
proc_ss_0:        .word 0x0000
proc_sp_0:        .word 0x0000
proc_ss_1:        .word 0x0000
proc_sp_1:        .word 0x0000
proc_syscallnr:   .word 0x0000
proc_msg_offset:  .word 0x0000
proc_msg_size:    .word 0x0000
__ksp:            .word 0x0000
__clock_ticks:
ticks_lo:         .word 0x0000
ticks_hi:         .word 0x0000
__clock_seconds:
seconds_lo:       .word 0x0000
seconds_hi:       .word 0x0000
…
```

Si tratta di variabili scalari da 16 bit, tenendo conto che: i simboli '**ticks_lo**' e '**ticks_hi**' compongono assieme la variabile ***_clock_ticks*** a 32 bit per il linguaggio C; i simboli '**seconds_lo**' e '**seconds_hi**' compongono assieme la variabile ***_clock_seconds*** a 32 bit per il linguaggio C.

Dopo la dichiarazione delle variabili inizia il codice vero e proprio. Il simbolo '**isr_1C**' si riferisce al codice da usare in presenza dell'interruzione $1C_{16}$, mentre il simbolo '**isr_80**' riguarda l'interruzione 80_{16}.

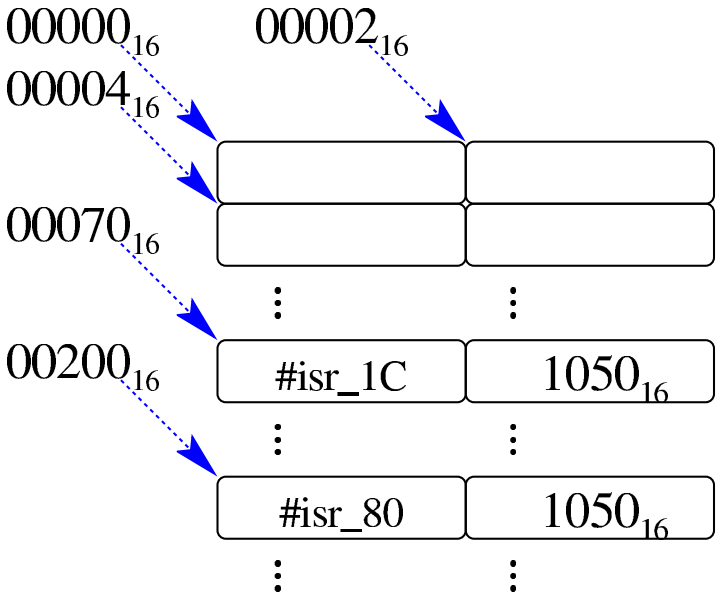
Nel file '`kernel/proc/_ivt_load.s`', la funzione ***_ivt_load()*** che inizia con il simbolo '**__ivt_load**', modifica la tabella IVT (*Interrupt vector table*) in modo che le interruzioni $1C_{16}$ e 80_{16} portino all'esecuzione del codice che inizia rispettivamente in corrispondenza dei simboli '**isr_1C**' e '**isr_80**' (del file '`kernel/proc/_isr.s`').

```
…
__ivt_load:
    enter #0, #0          ; No local variables.
    pushf
    cli
    pusha
    ;
    mov   ax,   #0        ; Change the DS segment to 0.
    mov   ds,   ax        ;
    ;
    mov   bx,   #112      ; Timer      INT 0x08 (8) --> 0x1C
    mov   [bx], #isr_1C   ; offset
    mov   bx,   #114      ;
    mov   [bx], cs        ; segment
    ;
    mov   bx,   #512      ; Syscall    INT 0x80 (128)
```

```
    mov   [bx], #isr_80   ; offset
    mov   bx,   #514      ;
    mov   [bx], cs        ; segment
    ;
    mov   ax,   #0x0050   ; Put the DS segment back to the
    mov   ds,   ax        ; right value.
    ;
    popa
    popf
    leave
    ret
```

Per compiere il suo lavoro, la funzione ***_ivt_load()*** salva inizialmente lo stato degli indicatori contenuti nel registro ***FLAGS*** e gli altri registri principali, quindi modifica il registro ***DS*** in modo che il segmento dati corrisponda allo zero, per poter accedere al contenuto della tabella IVT (che inizia proprio dall'indirizzo 00000_{16}). A quel punto, all'indirizzo efficace 00070_{16} (112_{10}) scrive l'indirizzo relativo del simbolo '**`isr_1C`**' (l'indirizzo relativo al segmento codice attuale) e il valore del segmento codice all'indirizzo efficace 00072_{16} (114_{10}). Nello stesso modo agisce per il simbolo '**`isr_80`**', scrivendo il suo indirizzo relativo all'indirizzo efficace 00200_{16} (512_{10}), assieme al valore del segmento codice che va invece in 00202_{16} (514_{10}). In tal modo, quando scatta l'interruzione $1C_{16}$ che deriva dalla scansione del temporizzatore interno, viene eseguito il codice che si trova nella voce corrispondente della tabella IVT, ovvero, proprio ciò che comincia con il simbolo '**`isr_1C`**', mentre quando scatta l'interruzione 80_{16} si ottiene l'esecuzione del codice che si trova a partire dal simbolo '**`isr_80`**'.

Figura u149.3. Modifica della tabella IVT attraverso la funzione ***_ivt_load()***. Il valore del segmento codice è sicuramente 1050_{16}, in quanto si tratta di quello del kernel, il quale va a collocarsi in quella posizione.



Le interruzioni previste con os16 sono solo due: quella del temporizzatore (*timer*) che invia un impulso a 18,2 Hz circa e quella che serve per le chiamate di sistema. Per la precisione, il temporizzatore fa scattare l'interruzione 08_{16}, ma se si utilizza il codice del BIOS, non può essere ridiretta; pertanto, il codice predefinito per tale interruzione, al termine del suo compito, fa scattare l'interruzione $1C_{16}$, la quale può essere ridiretta come appena mostrato.

Il codice per le due interruzioni gestite è simile, con la differenza fondamentale che per l'interruzione proveniente dal temporizzatore si incrementano i contatori rappresentati dalle variabili ***_clock_ticks*** e ***_clock_seconds***. Il codice equivalente della gestione delle due interruzioni è il seguente:

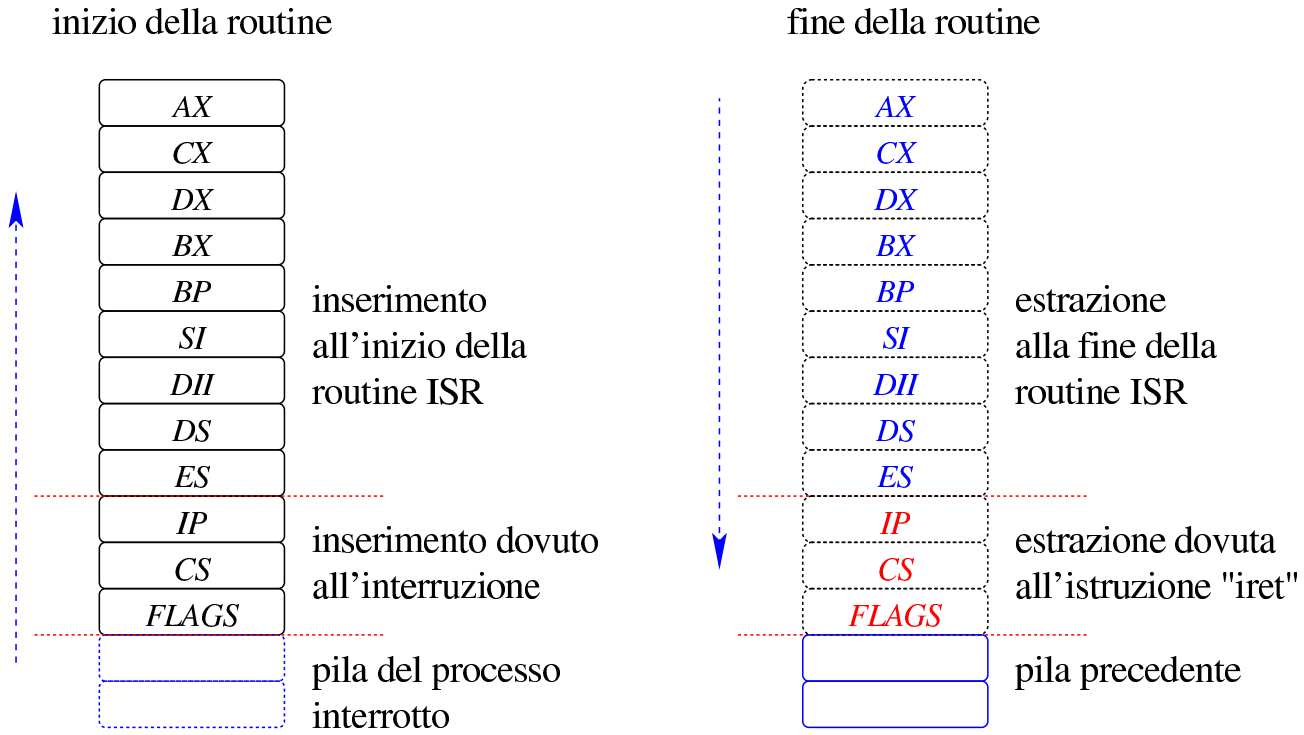
```
...
isr_1C:|isr_80:
```

```
    push   es   ; extra segment
    push   ds   ; data segment
    push   di   ; destination index
    push   si   ; source index
    push   bp   ; base pointer
    push   bx   ; BX
    push   dx   ; DX
    push   cx   ; CX
    push   ax   ; AX
    ;
    mov ax, #0x0050  ; DS and ES.
    mov ds, ax       ;
    mov es, ax       ;
    …
    …
    pop    ax
    pop    cx
    pop    dx
    pop    bx
    pop    bp
    pop    si
    pop    di
    pop    ds
    pop    es
    ;
    iret
…
```

Mentre viene eseguito il codice che si trova a partire da '**isr_1C**' o da '**isr_80**', il segmento codice è quello del kernel, ma quello dei dati è quello del processo che è stato interrotto poco prima. Nella pila dei dati di quel processo, nel momento in cui viene raggiunto questo codice ci sono già i valori di alcuni registri, nello stato in

cui erano al verificarsi dell'interruzione: ***FLAGS***, ***CS***, ***IP***. Come si vede dal codice appena mostrato, si aggiungono nella pila altri registri.

Figura u149.5. Inserimento nella pila del processo interrotto.

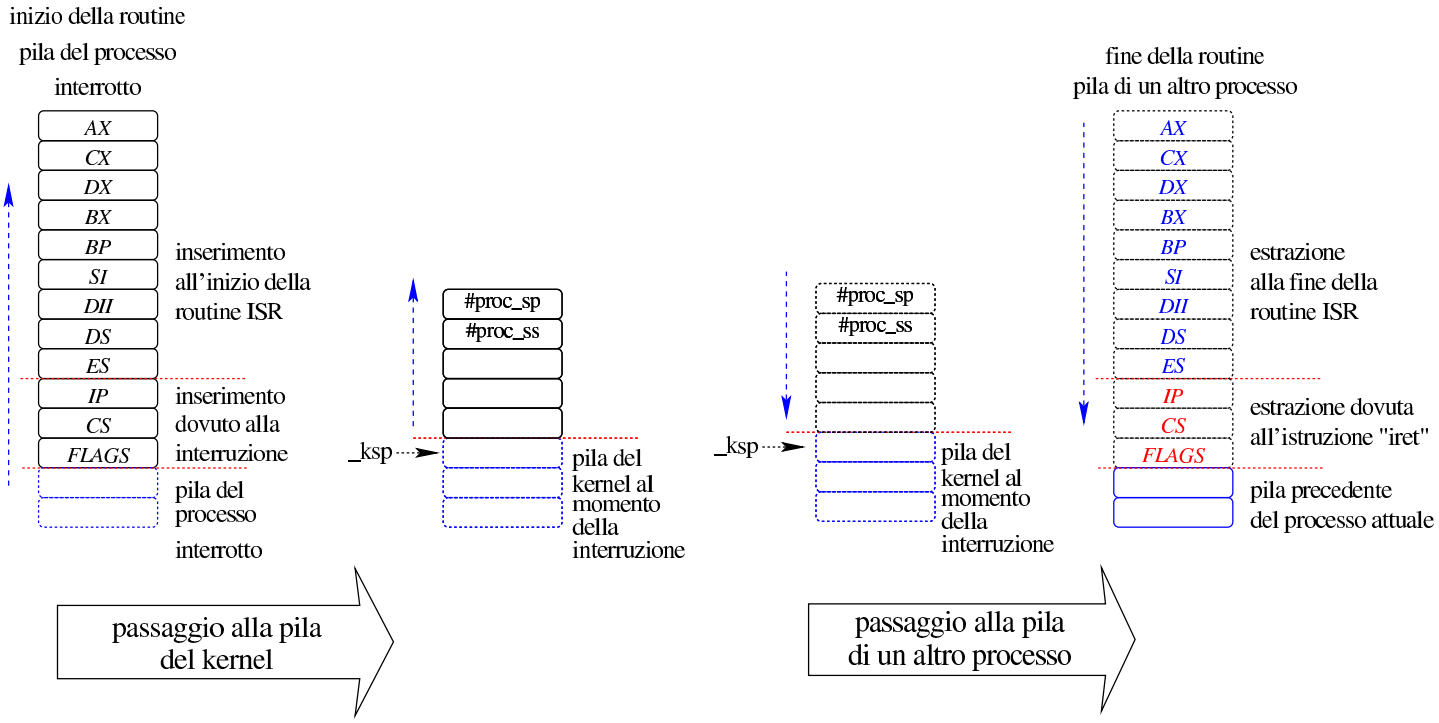


Dopo il salvataggio nella pila dei registri principali, viene modificato il valore dei registri ***DS*** e ***ES***, per consentire l'accesso alle variabili dichiarate all'inizio del file '`kernel/_isr.s`'. Il valore che si attribuisce a tali registri è 0050_{16}, perché il segmento dati del kernel inizia all'indirizzo efficace 00500_{16}. Va osservato che il segmento usato per la pila dei dati non viene ancora modificato e rimane nel segmento dati del processo interrotto.

A questo punto iniziano le differenze tra le due routine di gestione delle interruzioni. In ogni caso rimane il principio di massima, descritto intuitivamente dalla figura successiva, per cui si scambia la pila del processo interrotto con quella del kernel, poi si esegue la chiamata di sistema o si attiva lo schedulatore, quindi si passa nuova-

mente alla pila di un processo, il quale può essere diverso da quello interrotto.

Figura u149.6. Scambi delle pile.



Routine «isr_1C»

«

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, la routine '**`isr_1C`**' si occupa di incrementare i contatori degli impulsi e dei secondi:

```
...
isr_1C:
    ...
    add ticks_lo, #1    ; Clock ticks counter.
    adc ticks_hi, #0    ;
    ;
    mov dx, ticks_hi    ;
    mov ax, ticks_lo    ; DX := ticks % 18
    mov cx, #18         ;
```

```
    div cx                ;
    mov ax, #0            ; If the ticks value can be divided
    cmp ax, dx            ; by 18, the seconds is incremented
    jnz L1                ; by 1.
    add seconds_lo, #1    ;
    adc seconds_hi, #0    ;
    ;
L1:
    …
```

Per semplificare i calcoli, si considera che ogni 18 impulsi sia trascorso un secondo e di conseguenza va interpretata la divisione che viene eseguita. In ogni caso, quando si arriva al simbolo '**L1**' le variabili sono state aggiornate correttamente.

A questo punto viene salvato il valore del segmento in cui si trova la pila dei dati e l'indice all'interno della stessa, usando delle variabili locali, le quali non sono però accessibili dal codice in linguaggio C:

```
…
L1:
    mov proc_ss_0, ss    ; Save process stack segment.
    mov proc_sp_0, sp    ; Save process stack pointer.
    …
```

Poi si verifica se la pila dei dati del processo interrotto si trova nel kernel. In tal caso, il suo segmento avrebbe il valore 0050_{16}. Se il segmento dati è proprio quello del kernel, si saltano le istruzioni successive, riprendendo dal ripristino dei registri dalla pila dei dati (dal simbolo '**L2**').

```
    …
    mov dx, proc_ss_0
```

```
    mov ax, #0x0050     ; Kernel data area.
    cmp dx, ax
    je L2
    …
```

Se non è il kernel che è stato interrotto, si fa in modo di saltare all'utilizzo della pila dei dati del kernel. Per fare questo viene sostituito il valore del registro '**SS**', facendo in modo che corrisponda al segmento dati del kernel stesso, quindi si modifica il valore del registro '**SP**', mettendovi il valore salvato precedentemente nella variabile ***_ksp*** (ovvero il simbolo '**__ksp**').

```
    …
    mov ax, #0x0050     ; Kernel data area.
    mov ss, ax
    mov sp, __ksp
    …
```

Nella variabile ***_ksp*** c'è sicuramente l'indice della pila del kernel, aggiornata dalla funzione ***proc_scheduler()***. Tale aggiornamento della variabile ***_ksp*** avviene quando il gestore dei processi elaborativi sospende il codice del kernel per mettere in funzione un altro processo.

A questo punto, il contesto esecutivo è diventato quello del kernel, provenendo però dall'interruzione di un altro processo. Quindi viene chiamata la funzione di attivazione dello schedulatore: ***proc_scheduler()***. Tale funzione richiede dei parametri e gli vengono forniti i puntatori alle variabili contenenti il segmento e l'indice della pila dei dati del processo interrotto.

```
    …
```

```
    push #proc_ss_0     ; &proc_ss_0
    push #proc_sp_0     ; &proc_sp_0
    call _proc_scheduler
    add  sp, #2
    add  sp, #2
    …
```

Al termine del lavoro della funzione ***proc_scheduler()***, i valori contenuti nelle variabili rappresentate dai simboli '**proc_ss_0**' e '**proc_sp_0**' possono essere stati sostituiti con quelli di un altro processo da attivare al posto di quello interrotto precedentemente. Infatti, i registri ***SS*** e ***SP*** vengono sostituiti subito dopo:

```
    …
    mov ss, proc_ss_0   ; Restore process stack segment.
    mov sp, proc_sp_0   ; Restore process stack pointer.
    …
```

Infine, si ripristinano gli altri registri, traendo i dati dalla nuova pila.

Routine «isr_80»

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, la routine '**isr_80**' salva il valore del segmento in cui si trova la pila dei dati e l'indice all'interno della stessa, usando delle variabili locali, le quali non sono però accessibili dal codice in C:

```
    …
    mov proc_ss_1, ss   ; Save process stack segment.
    mov proc_sp_1, sp   ; Save process stack pointer.
    …
```

Vengono quindi salvati dei dati contenuti ancora nella pila attuale, utilizzando delle variabili statiche, che però non sono accessibili dal codice C:

```
    …
    mov bp, sp
    mov ax, +26[bp]
    mov proc_syscallnr,  ax
    mov ax, +28[bp]
    mov proc_msg_offset, ax
    mov ax, +30[bp]
    mov proc_msg_size,   ax
    …
```

Finalmente si passa a verificare se il processo interrotto è il kernel o meno: se si tratta proprio del kernel, il valore del registro ***SP*** viene salvato nella variabile ***_ksp***.

```
    …
    mov dx, ss
    mov ax, #0x0050     ; Kernel data area.
    cmp dx, ax
    jne L3
    mov __ksp, sp
L3:
    …
```

Successivamente si scambia la pila dei dati attuale, passando a quella del kernel, utilizzando la variabile ***_ksp*** per modificare il registro ***SP***. Naturalmente si comprende che se il codice interrotto era già quello del kernel, la sostituzione non cambia in pratica i valori che già avevano i registri ***SS*** e ***SP***:

```
    …
L3:
    mov ax, #0x0050     ; Kernel data area.
    mov ss, ax
    mov sp, __ksp
    …
```

Quando la pila dei dati in funzione è quella del kernel, si passa alla chiamata della funzione ***sysroutine()***, passandole come parametri i dati raccolti precedentemente dalla pila del processo interrotto, fornendo anche i puntatori alle variabili che contengono i dati necessari a raggiungere tale pila.

```
    …
    push proc_msg_size
    push proc_msg_offset
    push proc_syscallnr
    push #proc_ss_1     ; &proc_ss_1
    push #proc_sp_1     ; &proc_sp_1
    call _sysroutine
    add  sp, #2
    add  sp, #2
    add  sp, #2
    add  sp, #2
    add  sp, #2
    …
```

La funzione ***sysroutine()*** chiama a sua volta la funzione ***proc_scheduler()***, la quale può modificare il contenuto delle variabili rappresentate dai simboli '**`proc_ss_1`**' e '**`proc_sp_1`**'; pertanto, quando i valori di tali variabili vengono usati per rimpiazzare il contenuto dei registri ***SS*** e ***SP***, si ottiene lo scambio a un processo

diverso da quello interrotto inizialmente.

```
    …
    mov ss, proc_ss_1   ; Restore process stack segment.
    mov sp, proc_sp_1   ; Restore process stack pointer.
    …
```

Infine, si ripristinano gli altri registri, traendo i dati dalla nuova pila.

La tabella dei processi

«

Listato u0.9.

Nel file ‘`kernel/proc.h`’ viene definito il tipo ‘**`proc_t`**’, con il quale, nel file ‘`kernel/proc/proc_table.c`’ si definisce la tabella dei processi, rappresentata dall’array ***proc_table[]***.

Figura u149.19. Struttura del tipo '**`proc_t`**', corrispondente agli elementi dell'array ***proc_table[]***.

Listato u149.20. Struttura del tipo '**proc_t**', corrispondente agli elementi dell'array ***proc_table[]***.

```
typedef struct {
  pid_t              ppid;
  pid_t              pgrp;
  uid_t              uid;
  uid_t              euid;
  uid_t              suid;
  dev_t              device_tty;
  char               path_cwd[PATH_MAX];
  inode_t           *inode_cwd;
  int                umask;
  unsigned long int  sig_status;
  unsigned long int  sig_ignore;
  clock_t            usage;
  unsigned int       status;
  int                wakeup_events;
  int                wakeup_signal;
  unsigned int       wakeup_timer;
  addr_t             address_i;
  segment_t          segment_i;
  size_t             size_i;
  addr_t             address_d;
  segment_t          segment_d;
  size_t             size_d;
  uint16_t           sp;
  int                ret;
  char               name[PATH_MAX];
  fd_t               fd[FOPEN_MAX];
} proc_t;
```

La tabella successiva descrive il significato dei vari membri previsti dal tipo '**proc_t**'. Va osservato che os16 non gestisce i gruppi di utenti, anche se questi sono previsti comunque nel file system, per-

tanto la tabella dei processi è più semplice rispetto a quella di un sistema conforme allo standard di Unix. Un'altra considerazione va fatta a proposito della cosiddetta «u-area» (*user area*), la quale non viene gestita come un sistema Unix tradizionale e tutti i dati dei processi sono raccolti nella tabella gestita dal kernel. Di conseguenza, dal momento che i processi non dispongono di una tabella personale con i dati della u-area, devono avvalersi sempre di chiamate di sistema per leggere i dati del proprio processo.

Tabella u149.21. Membri del tipo '**`proc_t`**'.

Membro	Contenuto
`ppid`	Numero del processo genitore: *parent process id*.
`pgrp`	Numero del gruppo di processi a cui appartiene quello della voce corrispondente: *process group*. Si tratta del numero del processo a partire dal quale viene definito il gruppo.
`uid`	Identità reale del processo della voce corrispondente: *user id*. Si tratta del numero dell'utente, secondo la classificazione del file '`/etc/passwd`', per conto del quale il processo è stato avviato. Tuttavia, i privilegi del processo dipendono dall'identità efficace, definita dal membro '**`euid`**'.

Membro	Contenuto
`euid`	Identità efficace del processo della voce corrispondente: *effective user id*. Si tratta del numero dell'utente, secondo la classificazione del file '`/etc/passwd`', per conto del quale il processo è in funzione; pertanto, il processo ha i privilegi di quell'utente.
`suid`	Identità salvata: *saved user id*. Si tratta del valore che aveva ***euid*** prima di cambiare identità.
`device_tty`	Terminale di controllo, espresso attraverso il numero del dispositivo.
`path_cwd` `inode_cwd`	Entrambi i membri rappresentano la directory corrente del processo: nel primo caso in forma di percorso, ovvero di stringa, nel secondo in forma di puntatore a inode rappresentato in memoria.
`umask`	Maschera dei permessi associata al processo: i permessi attivi nella maschera vengono tolti in fase di creazione di un file o di una directory.
`sig_status`	Segnali inviati al processo e non ancora trattati: ogni segnale si associa a un bit differente del valore del membro ***sig_status***; un bit a uno indica che il segnale corrispondente è stato ricevuto e non ancora trattato.
`sig_ignore`	Segnali che il processo ignora: ogni segnale da ignorare si associa a un bit differente del valore del membro ***sig_ignore***; un bit a uno indica che quel segnale va ignorato.

Membro	Contenuto
`usage`	Tempo di utilizzo della CPU, da parte del processo, espresso in impulsi del temporizzatore, il quale li produce alla frequenza di circa 18,2 Hz.
`status`	Stato del processo, rappresentabile attraverso una macro-variabile simbolica, definita nel file '`proc.h`'. Per os16, gli stati possibili sono: «inesistente», quando si tratta di una voce libera della tabella dei processi; «creato», quando un processo è appena stato creato; «pronto», quando un processo è pronto per essere eseguito, «in esecuzione», quando il processo è in funzione; «sleeping», quando un processo è in attesa di qualche evento; «zombie», quando un processo si è concluso, ha liberato la memoria, ma rimangono le sue tracce perché il genitore non ha ancora recepito la sua fine.
`wakeup_events`	Eventi attesi per il risveglio del processo, ammesso che si trovi nello stato si attesa. Ogni tipo di evento che può essere atteso corrisponde a un bit e si rappresenta con una macro-variabile simbolica, dichiarata nel file '`lib/sys/os16.h`'.
`wakeup_signal`	Ammesso che il processo sia in attesa di un segnale, questo membro esprime il numero del segnale atteso.

Membro	Contenuto
`wakeup_timer`	Ammesso che il processo sia in attesa dello scadere di un conto alla rovescia, questo membro esprime il numero di secondi che devono ancora trascorrere.

`address_i`
`segment_i`
`size_i`
	Il valore di questi membri descrive la memoria utilizzata dal processo per le istruzioni (il segmento codice). Le informazioni sono in parte ridondanti, perché conoscendo ***segment_i*** si ottiene facilmente ***address_i*** e viceversa, ma ciò consente di ridurre i calcoli nelle funzioni che ne fanno uso.

`address_d`
`segment_d`
`size_d`
	Il valore di questi membri descrive la memoria utilizzata dal processo per i dati (il segmento usato per le variabili statiche e per la pila). Anche in questo caso, le informazioni sono in parte ridondanti, ma ciò consente di semplificare il codice nelle funzioni che ne fanno uso.

`sp`	Indice della pila dei dati, nell'ambito del segmento dati del processo. Il valore è significativo quando il processo è nello stato di pronto o di attesa di un evento. Quando invece un processo era attivo e viene interrotto, questo valore viene aggiornato.
`ret`	Rappresenta il valore restituito da un processo terminato e passato nello stato di «zombie».
`name`	Il nome del processo, rappresentato dal nome del programma avviato.

Membro	Contenuto
`fd`	Tabella dei descrittori dei file relativi al processo.

Chiamate di sistema

I processi eseguono una chiamata di sistema attraverso la funzione ***sys()***, dichiarata nel file '`lib/sys/os16/sys.s`'. La funzione in sé, per come è dichiarata, potrebbe avere qualunque parametro, ma in pratica ci si attende che il suo prototipo sia il seguente:

```
void sys (syscallnr, void *message, size_t size);
```

Il numero della chiamata di sistema, richiesto come primo parametro, si rappresenta attraverso una macro-variabile simbolica, definita nel file '`lib/sys/os16.h`'.

Per fornire dei dati a quella parte di codice che deve svolgere il compito richiesto, si usa una variabile strutturata, di cui viene trasmesso il puntatore (riferito al segmento dati del processo che esegue la chiamata) e la dimensione complessiva.

Nel file '`lib/sys/os16.h`' sono definiti dei tipi derivati, riferiti a variabili strutturate, per ogni tipo di chiamata. Per esempio, per la chiamata di sistema usata per cambiare la directory corrente del processo, si usa un messaggio di tipo '**`sysmsg_chdir_t`**':

```
typedef struct {
    char          path[PATH_MAX];
    int           ret;
    int           errno;
    int           errln;
    char          errfn[PATH_MAX];
} sysmsg_chdir_t;
```

In realtà, la funzione ***sys()***, si limita a produrre un'interruzione software, da cui viene attivata la routine che inizia al simbolo '**isr_80**' nel file 'kernel/_isr.s', la quale estrapola le informazioni salienti dalla pila dei dati e poi le fornisce alla funzione ***sysroutine()***:

```
void sysroutine (uint16_t *sp, segment_t *segment_d,
                 uint16_t syscallnr, uint16_t msg_off,
                 uint16_t msg_size);
```

Nella funzione ***sysroutine()***, gli ultimi tre parametri corrispondono in pratica agli argomenti della chiamata della funzione ***sys()***, con la differenza che nei vari passaggi hanno perso l'identità originaria e giungono come numeri puri e semplici, secondo la «parola» del tipo di architettura utilizzato.

File «kernel/proc/...»

«

Listati successivi a u0.9.

Nella directory 'kernel/proc/' si trovano i file che realizzano le funzioni dichiarate all'interno di 'kernel/proc.h'.

Nella gestione dei processi entrano in gioco due variabili globali importanti: ***_ksp*** e ***_etext***. La prima è dichiarata nel file 'kernel/

`_isr.s`’ e viene utilizzata per annotare l’indice della pila dei dati del kernel; la seconda è dichiarata implicitamente dal collegatore (*linker*) e contiene la dimensione dell’area occupata in memoria dal codice del kernel stesso.

Nel file ‘`kernel/proc/proc_table.c`’ è dichiarata la tabella dei processi, attraverso un array composto da elementi di tipo ‘**`proc_t`**’. La quantità di elementi di questo array costituisce il limite alla quantità di processi gestibili simultaneamente, incluso il kernel e i processi zombie.

Per accedere uniformemente al contenuto della tabella, si usa la funzione ***proc_reference()***, la quale, con l’indicazione del numero del processo (PID), restituisce il puntatore all’elemento della tabella che contiene i dati dello stesso.

Nelle sezioni successive si descrivono solo le funzioni principali della directory ‘`kernel/proc/`’.

Funzione «proc_init()»

```
void proc_init (void);
```

La funzione ***proc_init()*** viene chiamata dalla funzione ***main()***, una volta sola, per attivare la gestione dei processi elaborativi. Si occupa di compiere le azioni seguenti:

- modificare la tabella delle interruzioni (IVT), attraverso la chiamata della funzione ***_ivt_load()*** (per comodità si usa la macroistruzione ***ivt_load()***), dichiarata nel file ‘`kernel/proc/_ivt_load.s`’;

- impostare la frequenza del temporizzatore, ma tale frequenza deve essere obbligatoriamente di 18,2 Hz;
- azzerare la tabella dei processi;
- innestare il file system principale;
- assegnare i valori appropriati alla voce della tabella dei processi che si riferisce al kernel (PID zero);
- allocare la memoria già utilizzata dal kernel e lo spazio che va da zero fino a 00500_{16} (tabella IVT e BDA);
- attivare selettivamente le interruzioni hardware desiderate.

Funzione «sysroutine()»

«

La funzione ***sysroutine()*** viene chiamata esclusivamente dalla routine attivata dalle chiamate di sistema (tale routine è introdotta dal simbolo '**`isr_80`**' nel file '`kernel/proc/_isr.s`') e ha una serie di parametri, come si può vedere dal prototipo:

```
void sysroutine (uint16_t *sp, segment_t *segment_d,
                 uint16_t syscallnr, uint16_t msg_off,
                 uint16_t msg_size);
```

I primi due parametri della funzione sono puntatori a variabili dichiarate nel file '`kernel/proc/_isr.s`'. La prima delle due variabili è l'indice della pila dei dati del processo che ha eseguito la chiamata di sistema; la seconda contiene l'indirizzo del segmento dati di tale processo. Il valore del segmento dati serve a individuare il processo elaborativo nella tabella dei processi, dal momento che con os16 i dati non sono condivisibili tra processi.

Il terzo parametro è il numero della chiamata di sistema che ha provocato l'interruzione. Gli ultimi due parametri danno la posizione e la dimensione del messaggio inviato attraverso la chiamata di sistema.

All'inizio della funzione viene individuato il processo elaborativo corrispondente a quello che utilizza il segmento dati ***segment_d** e l'indirizzo efficace dell'area di memoria contenente il messaggio della chiamata di sistema:

```
    pid_t  pid      = proc_find (*segment_d);
    addr_t msg_addr = address (*segment_d, msg_off);
```

Quindi viene dichiarata un'unione di variabili strutturate, corrispondente alla sovrapposizione di tutti i tipi di messaggio gestibili:

```
    union {
        sysmsg_chdir_t      chdir;
        sysmsg_chmod_t      chmod;
        …
    } msg;
```

A questo punto si verifica se il processo interrotto dalla sua chiamata di sistema è il kernel, perché al kernel è consentito di eseguire solo alcuni tipi di chiamata e tutto il resto sarebbe un errore.

Proseguendo con il codice si vede l'uso della funzione ***dev_io()***, con la quale si legge il messaggio della chiamata di sistema, dalla sua collocazione originale, in un'area tampone del segmento dati del kernel:

```
    dev_io (pid, DEV_MEM, DEV_READ, msg_addr, &msg,
            msg_size, NULL);
```

A questo punto, sapendo di quale chiamata di sistema si tratta, il messaggio può essere letto come:

```
msg.tipo_chiamata
```

Per esempio, per la chiamata di sistema '**SYS_CHDIR**', si deve fare riferimento al messaggio ***msg.chdir***; pertanto, per raggiungere il membro ***ret*** del messaggio si usa la notazione ***msg.chdir.ret***.

Una volta eseguita una copia del messaggio, con la funzione ***dev_io()***, si passa a una struttura di selezione, con cui si eseguono operazioni differenti in base al tipo di chiamata ricevuta:

```
    switch (syscallnr)
      {
        case SYS_0:
            break;
        case SYS_CHDIR:
            msg.chdir.ret = path_chdir (pid, msg.chdir.path);
            sysroutine_error_back (&msg.chdir.errno,
                                   &msg.chdir.errln,
                                   msg.chdir.errfn);
            break;
        …
      }
```

Il messaggio usato per trasmettere i dati della chiamata, può servire anche per restituire dei dati al mittente, pertanto, spesso alcuni contenuti dello stesso vengono modificati. Ciò succede particolarmente con il membro ***ret*** che generalmente rappresenta il valore restituito dalla chiamata di sistema. Per questa ragione, dopo la struttura di selezione si ricopia nuovamente il messaggio nella posizione di partenza:

```
    dev_io (pid, DEV_MEM, DEV_WRITE, msg_addr, &msg,
            msg_size, NULL);
```

Al termine del lavoro, viene chiamata la funzione ***proc_scheduler()***.

Funzione «proc_scheduler()»

La funzione ***proc_scheduler()*** richiede come parametri due puntatori: il primo parametro deve essere il riferimento a un valore che rappresenta l'indice della pila di quel processo; il secondo parametro si riferisce a una variabile contenente il valore del segmento dati del processo interrotto. La funzione richiede queste informazioni in forma di puntatore, per poter modificare i valori delle variabili relative, in modo da consentire così l'attivazione successiva di un altro processo, al posto di quello da cui si proviene.

```
void proc_scheduler (uint16_t *sp, segment_t *segment_d);
```

Inizialmente, la funzione acquisisce il numero del processo interrotto:

```
    prev = proc_find (*segment_d);
```

Quindi svolge delle operazioni che riguardano tutti i processi: aggiorna i contatori dei processi che attendono lo scadere di un certo tempo; verifica la presenza di segnali e predispone le azioni relative; raccoglie l'input dai terminali.

```
    proc_sch_timers ();
    …
    proc_sch_signals ();
    …
    proc_sch_terminals ();
```

A quel punto aggiorna il tempo di utilizzo della CPU del processo appena interrotto:

```
    current_clock   = k_clock ();
    ps[prev].usage += current_clock - previous_clock;
    previous_clock  = current_clock;
```

Quindi inizia la ricerca di un altro processo, candidato a essere ripreso, al posto di quello interrotto. La ricerca inizia dal processo successivo a quello interrotto, senza considerare alcun criterio di precedenza. Il ciclo termina se la ricerca incontra di nuovo il processo di partenza.

```
    for (next = prev+1; next != prev; next++)
      {
        if (next >= PROCESS_MAX)
          {
            next = -1; // At the next loop, 'next' will be
                       // zero.
            continue;
          }
        …
      }
```

All'interno di questo ciclo di ricerca, se si incontra un processo pronto per essere messo in funzione, lo si scambia con quello interrotto: in pratica si salva il valore attuale dell'indice della pila, si scambiano gli stati e si aggiornano i valori di ***sp** e ***segment_d**, in modo da ottenere effettivamente lo scambio all'uscita dalla funzione:

```
        else if (ps[next].status == PROC_READY)
          {
            if (ps[prev].status == PROC_RUNNING)
              {
                ps[prev].status = PROC_READY;
              }
            ps[prev].sp     = *sp;
            ps[next].status = PROC_RUNNING;
            ps[next].ret    = 0;
            *segment_d      = ps[next].segment_d;
            *sp             = ps[next].sp;
            break;
          }
```

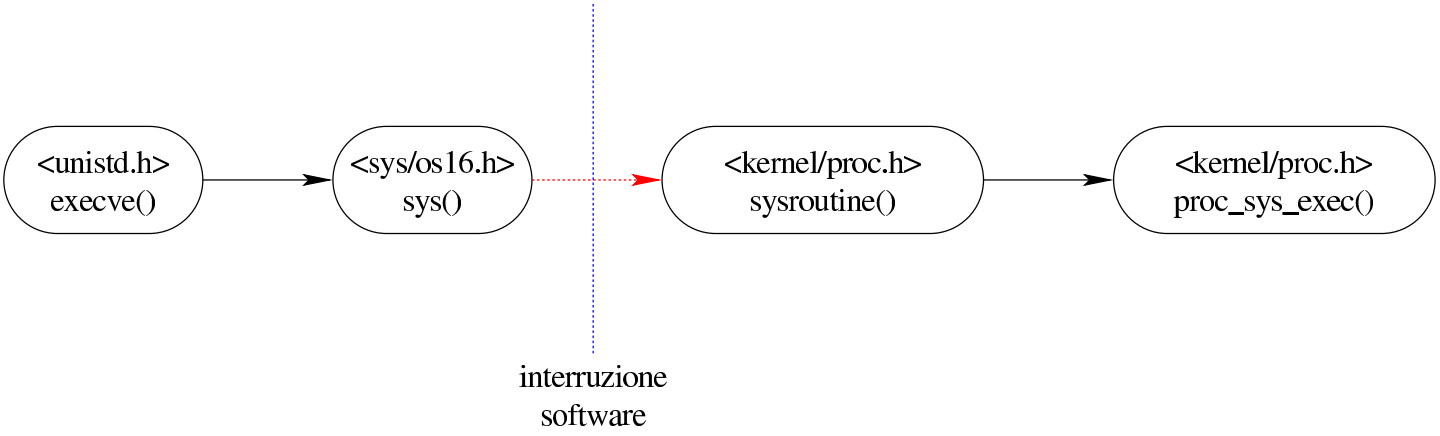
Alla fine del ciclo, occorre verificare se esiste effettivamente un processo successivo attivato, perché in caso contrario, si lascia il controllo direttamente al kernel. In fine, si salva il valore accumulato in precedenza dell'indice della pila del kernel, nella variabile ***_ksp***, quindi si manda il messaggio EOI al circuito del PIC (*programmable interrupt controller*), diversamente non ci sarebbero più, altre interruzioni.

Caricamento ed esecuzione delle applicazioni

Caricamento in memoria 3171
Il codice iniziale dell'applicativo 3174

Caricare un programma e metterlo in esecuzione è un processo delicato che parte dalla funzione ***execve()*** della libreria standard e viene svolto dalla funzione ***proc_sys_exec()*** del kernel.

Figura u150.1. Da ***execve()*** a ***proc_sys_exec()***.



Caricamento in memoria

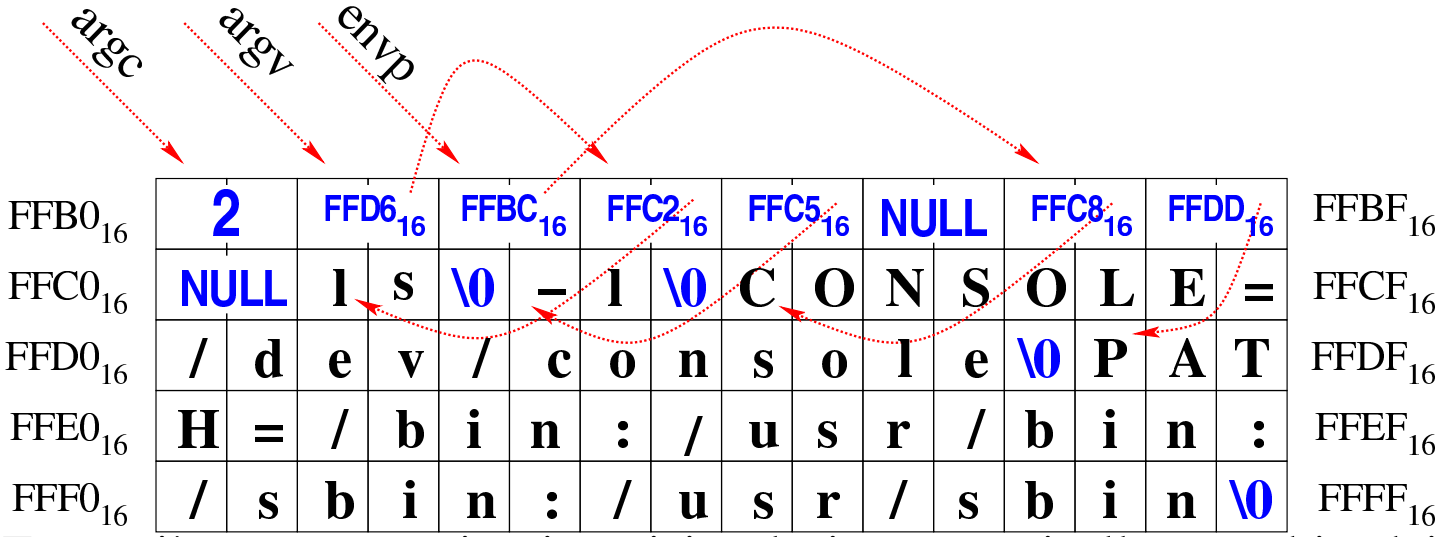
La funzione ***proc_sys_exec()*** (listato i160.9.21) del kernel è quella che svolge il compito di caricare un processo in memoria e di annotarlo nella tabella dei processi.

La funzione, dopo aver verificato che si tratti di un file eseguibile valido e che ci siano i permessi per metterlo in funzione, procede all'allocazione della memoria, dividendo se necessario l'area codice da quella dei dati, quindi legge il file e copia opportunamente le componenti di questo nelle aree di memoria allocate.

La realizzazione attuale della funzione ***proc_sys_exec()*** non è in grado di verificare se un processo uguale sia già in memoria, quindi carica la parte del codice anche se questa potrebbe essere già disponibile.

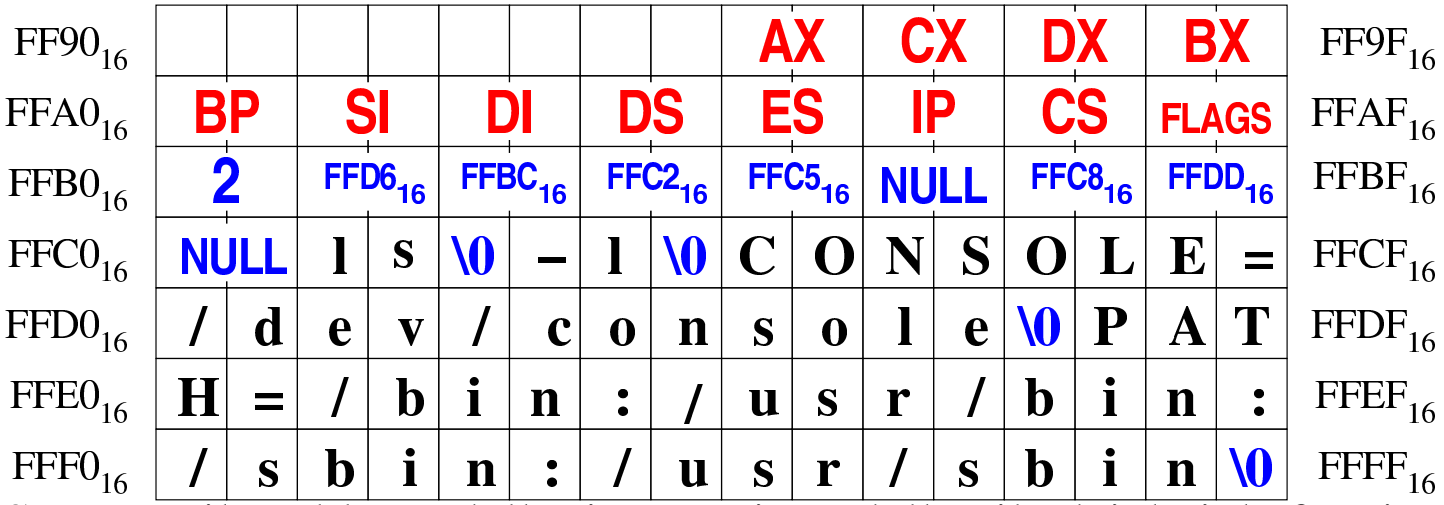
Terminato il caricamento del file, viene ricostruita in memoria la pila dei dati del processo. Prima si mettono sul fondo le stringhe delle variabili di ambiente e quelle degli argomenti della chiamata, quindi si aggiungono i puntatori alle stringhe delle variabili di ambiente, ricostruendo così l'array noto convenzionalmente come '**`envp[]`**', continuando con l'aggiunta dei puntatori alle stringhe degli argomenti della chiamata, per riprodurre l'array '**`argv[]`**'. Per ricostruire gli argomenti della chiamata della funzione ***main()*** dell'applicazione, vanno però aggiunti ancora: il puntatore all'inizio dell'array delle stringhe che descrivono le variabili di ambiente, il puntatore all'array delle stringhe che descrivono gli argomenti della chiamata e il valore che rappresenta la quantità di argomenti della chiamata.

Figura u150.2. Caricamento degli argomenti della chiamata della funzione ***main()***.



Fatto ciò, vanno aggiunti tutti i valori necessari allo scambio dei processi, costituiti dai vari registri da rimpiazzare.

Figura u150.3. Completamento della pila con i valori dei registri.



Superato il problema della ricostruzione della pila dei dati, la funzione ***proc_sys_exec()*** predispone i descrittori di standard input, standard output e standard error, quindi libera la memoria usata dal processo chiamante e ne rimpiazza i dati nella tabella dei processi con quelli del nuovo processo caricato.

Il codice iniziale dell'applicativo

«

I programmi iniziano con il codice che si trova nel file '`applic/crt0.s`'. Questo file ha delle affinità con il file '`kernel/main/crt0.s`' del kernel, dove la prima differenza che si incontra riguarda l'impronta di riconoscimento. A parte questo, va considerato che il codice delle applicazioni viene eseguito in un momento in cui i registri di segmento sono già stati impostati e l'indice della pila è già collocato correttamente; inoltre, se la funzione ***main()*** termina e restituisce il controllo a '`crt0.s`', un ciclo senza fine esegue continuamente una chiamata di sistema per la conclusione del processo elaborativo corrispondente.

Figura u150.4. Codice iniziale degli applicativi e variabile strutturata di tipo '**header_t**'.

```
entry startup
.text
startup:
    jmp startup_code
filler:
    .space (0x0004 - (filler - startup))
magic:
    .data4 0x6F733136
    .data4 0x6170706C
segoff:
    .data2 __segoff
etext:
    .data2 __etext
edata:
    .data2 __edata
ebss:
    .data2 __end
stack_size:
    .data2 0x2000
.align 2
startup_code:
…
```

```
typedef struct {
    uint32_t filler0;
    uint32_t magic0;
    uint32_t magic1;
    uint16_t segoff;
    uint16_t etext;
    uint16_t edata;
    uint16_t ebss;
    uint16_t ssize;
} header_t;
```

La figura mostra il confronto tra il codice iniziale contenuto nel file '`applic/crt0.s`', senza preamboli e senza commenti, con la dichiarazione del tipo derivato '**header_t**', presente nel file '`kernel/proc.h`'. Attraverso questa struttura, la funzione ***proc_sys_exec()*** è in grado di estrapolare dal file le informazioni necessarie a caricarlo correttamente in memoria.

Come già accennato, quando viene eseguito il codice di un programma applicativo, la pila dei dati è già operativa. Pertanto, dopo il simbolo '**startup_code**' si può già lavorare con questa.

```
startup_code:
    pop ax              ; argc
    pop bx              ; argv
    pop cx              ; envp
    mov _environ, cx    ; Variable 'environ' comes from
                        ; 'unistd.h'.
    push cx
    push bx
    push ax
```

Per prima cosa, viene estratto dalla pila il puntatore all'array noto come ***envp[]***, per poter assegnare tale valore alla variabile ***environ***, come richiede lo standard della libreria POSIX. Tuttavia, per poter gestire poi le variabili di ambiente, si rende necessario utilizzare un array più «comodo», quando le stringhe vanno sostituite. A tale proposito, nel file '`lib/stdlib/environment.c`', si dichiarano ***_environment_table[][]*** e ***_environment[]***. Il primo è semplicemente un array di caratteri, dove, utilizzando due indici di accesso, si conviene di allocare delle stringhe, con una dimensione massima prestabilita. Il secondo, invece, è un array di puntatori, per localizzare l'inizio delle stringhe contenute nel primo. In pratica, alla fine ***_environment[]*** e ***environ[]*** devono essere equivalenti. Ma per attuare questo, occorre utilizzare la funzione ***_environment_setup()*** che sistema tutti i puntatori necessari.

```
    push cx
    call __environment_setup
    add  sp, #2
    ;
    mov  ax, #__environment
    mov  _environ, ax
    ;
    pop ax              ; argc
    pop bx              ; argv[][]
    pop cx              ; envp[][]
    mov cx, #__environment
    push cx
    push bx
    push ax
```

Come si vede dall'estratto del file '`applic/crt0.s`', si vede l'uso della funzione ***_environment_setup()*** (il registro CX contiene già il puntatore a ***envp[]***, e viene inserito nella pila proprio come argomento per la funzione). Successivamente viene riassegnata anche la variabile ***environ*** in modo da coincidere con ***_environment***. Alla fine, viene ricostruita la pila per gli argomenti della chiamata della funzione ***main()***, ma prima di procedere con quella chiamata, si utilizzano due funzioni, per inizializzare la gestione dei flussi di file e delle directory, sempre in forma di flussi.

```
    call __stdio_stream_setup
    call __dirent_directory_stream_setup
    ;
    call _main
    ;
    mov  exit_value, ax
    …
.align 2
.data
exit_value:
    .data2 0x0000
.align 2
.bss
```

La funzione ***_stdio_stream_setup()***, contenuta nel file '`lib/stdio/FILE.c`', associa i descrittori standard ai flussi di file standard (standard input, standard output e standard error); la funzione ***_dirent_directory_stream_setup()*** compie un lavoro analogo, limitandosi però a inizializzare un array di flussi di directory.

Dopo queste preparazioni, viene chiamata la funzione ***main()***, la quale riceve regolarmente i propri argomenti previsti. Il valore restituito dalla funzione viene poi salvato in corrispondenza del simbolo '**`exit_value`**'.

```
halt:
    push #2             ; Size of message.
    push #exit_value    ; Pointer to the message.
    push #6             ; SYS_EXIT
    call _sys
    add  sp, #2
    add  sp, #2
    add  sp, #2
    jmp halt
```

All'uscita dalla funzione ***main()***, dopo aver salvato quanto restituito dalla funzione stessa, ci si introduce nel codice successivo al simbolo '**`halt`**', nel quale si chiama la funzione ***sys()*** (chiamata di sistema), per produrre la chiusura formale del processo. Ciò che si vede è comunque l'equivalente di '**`_exit (`*exit_status*`);`**'.[1]

[1] Va tenuto in considerazione che '**`exit_status`**' è un simbolo non raggiungibile dal codice C, perché dovrebbe essere esportato con un nome che inizi con il trattino basso.