Capitolo 84

Studio per un sistema a 32 bit

«

84.1 Introduzione a os32 . . . 99
84.1.1 Organizzazione . . . 99
84.1.2 Le directory . . . 100
84.1.3 La struttura degli eseguibili . . . 100
84.1.4 Tabelle . . . 102
84.1.5 Guida di stile . . . 102
84.1.6 Tipi derivati speciali . . . 104
84.2 Caricamento ed esecuzione del kernel . . . 105
84.2.1 Multiboot . . . 105
84.2.2 File «kernel.ld», «kernel/main/crt0.s» e «kernel/main/stack.s» . . . 108
84.2.3 File «kernel/main.h» e «kernel/main/*» . . . 109
84.3 Funzioni interne legate all'hardware, nei file «kernel/ibm_i386.h» e «kernel/ibm_i386/...» . . . 112
84.3.1 Funzioni per l'input e l'output con le porte interne 112
84.3.2 Funzioni accessorie alla gestione delle interruzioni hardware . . . 113
84.3.3 Gestione della tabella GDT . . . 113
84.3.4 Gestione della tabella IDT . . . 114
84.3.5 Gestione delle interruzioni . . . 115
84.4 Gestione dei processi . . . 117
84.4.1 File «kernel/ibm_i386/isr.s» . . . 117
84.4.2 La tabella dei processi . . . 121
84.4.3 Chiamate di sistema . . . 124
84.4.4 Funzione «proc_init()» . . . 125
84.4.5 Funzione «sysroutine()» . . . 125
84.4.6 Funzione «proc_scheduler()» . . . 126
84.4.7 Programmazione dei segnali . . . 127
84.4.8 Salvataggio e recupero della pila per i «salti non locali» 129
84.5 Caricamento ed esecuzione delle applicazioni . . . 131
84.5.1 Caricamento in memoria . . . 131
84.5.2 Il codice iniziale dell'applicativo . . . 132
84.6 Gestione della memoria . . . 134
84.6.1 File «kernel/memory.h» e «kernel/memory/...» . . . 135
84.6.2 Scansione della mappa di memoria . . . 136
84.7 Dispositivi . . . 137
84.7.1 File «kernel/dev.h» e «kernel/dev/...» . . . 137
84.7.2 File «kernel/blk.h» e «kernel/blk/...» . . . 138
84.7.3 Numero primario e numero secondario . . . 139
84.7.4 Dispositivi previsti . . . 139
84.7.5 Gestione del terminale . . . 141
84.7.6 Gestione delle unità di memorizzazione in generale 147
84.7.7 Gestione delle unità PATA . . . 147
84.8 Gestione del file system . . . 149
84.8.1 File «kernel/fs/sb_...» . . . 149
84.8.2 File «kernel/fs/zone_...» . . . 151
84.8.3 File «kernel/fs/inode_...» . . . 152
84.8.4 Fasi dell'innesto di un file system . . . 157
84.8.5 Condotti . . . 157
84.8.6 File «kernel/fs/file_...» . . . 159
84.8.7 Descrittori di file . . . 160
84.8.8 File «kernel/fs/path_...» . . . 160

«a2» 2013.11.11 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://informaticalibera.net

84.8.9 File «kernel/fs/fd_...» 163

84.9 Gestione delle interfacce di rete 164

84.9.1 Gestione dei dispositivi NE2K 164

84.9.2 Tabella delle interfacce e funzioni accessorie 165

84.9.3 Tabella ARP 167

84.10 Gestione di IPv4 168

84.10.1 Tabella IPv4 169

84.10.2 Ricezione di un pacchetto IPv4 170

84.10.3 Instradamenti 170

84.11 Gestione del protocollo ICMP 171

84.12 Gestione dei protocolli UDP e TCP 172

84.12.1 UDP 175

84.12.2 TCP 176

addr_t 104 135 arp.h 167 arp_clean() 168 arp_index() 168 arp_init() 168 arp_reference() 168 arp_request() 168 arp_rx() 168 ata_cmd_identify_device() 148 ata_cmd_read_sectors() 148 ata_cmd_write_sectors() 148 ata_device() 148 ata_drq() 148 ata_init() 147 ata_lba28() 148 ata_rdy() 148 ata_read_sector() 149 ata_reset() 147 ata_sector_t 104 147 ata_t 104 147 ata_valid() 147 ata_write_sector() 149 blk.h 138 blk_ata.c 138 blk_cache_init.c 138 blk_cache_read.c 138 blk_cache_save.c 138 blk_cache_t 104 cli() 113 crt0.s 108 dev.h 137 dev/dev_tty.c 137 DEV_CONSOLE 139 DEV_CONSOLE*n* 139 dev_dm.c 137 DEV_DM*mn* 139 dev_io() 137 dev_io.c 137 dev_kmem.c 137 DEV_KMEM_ARP 139 DEV_KMEM_FILE 139 DEV_KMEM_INODE 139 DEV_KMEM_MMP 139 DEV_KMEM_NET 139 DEV_KMEM_PS 139 DEV_KMEM_ROUTE 139 DEV_KMEM_SB 139 DEV_MEM 139 DEV_NULL 139 DEV_PORT 139 DEV_TTY 139 DEV_ZERO 139 directory_t 104 dm_t 147 fd_dup() 163 fd_reference() 163 fd_t 104 file_reference() 159 file_stdio_dev_make() 159 file_t 104 159 fs.h 149 gdt() 114 gdt_load() 114 gdt_print() 114 gdt_segment() 114 gdt_t 104 header_t 104 h_addr_t 104 168 ibm_i386.h 112 icmp_rx() 172 icmp_tx() 172 icmp_tx_echo() 172 icmp_tx_unreachable() 172 idt() 115 idtr_t 104 idt_descriptor() 115 idt_irq_remap() 115 idt_load() 115 idt_print() 115 idt_t 104 inode_alloc() 154 inode_check() 154 inode_dir_empty() 154 inode_file_read() 154 inode_file_write() 154 inode_free() 154 inode_fzones_read() 154 inode_fzones_write() 154 inode_get() 154 inode_pipe_make() 154 inode_pipe_read() 154 inode_pipe_write() 154 inode_print() 154 inode_put() 154 inode_reference() 154 inode_save() 154 inode_stdio_dev_make() 154 inode_t 104 152 inode_truncate() 154 inode_zone() 154 in_16() 112 in_8() 112 ip.h 168 ip_checksum() 168 ip_header() 168 ip_mask() 168 ip_reference() 168 ip_rx() 168 ip_table[] 169 ip_tx() 168 irq_off() 113 irq_on() 113 isr.s 117 isr_exception_name() 116 isr_exception_unrecoverable() 116 isr_irq_clear() 116 isr_irq_clear_pic1() 116 isr_irq_clear_pic2() 116 isr_*n*() 116 kbd_isr() 144 kbd_load() 144 kbd_t 104 kernel.ld 108 kernel/memory.c 135 longjmp() 129 main() 109 mboot_cmdline_opt() 107 mboot_save() 107 mb_alloc() 135 mb_alloc_size() 135 mb_clean() 135 mb_free() 135 mb_print() 135 mb_reduce() 135 mb_reference() 135 mb_size() 135 memory.h 135 MEM_BLOCK_SIZE 135 MEM_MAX_BLOCKS 135 multiboot_t 104 ne2k_check() 164 ne2k_isr() 164 ne2k_isr_expect() 164 ne2k_reset() 164 ne2k_rx() 164 ne2k_rx_reset() 164 ne2k_tx() 164 net.h 164 net_buffer_eth() 166 net_buffer_lo() 166 net_eth_ip_tx() 166 net_eth_tx() 166 net_index() 166 net_index_eth() 166 net_init() 166 net_rx() 166 os32.h 137 out_16() 112 out_8() 112 path_device() 161 path_fix() 161 path_full() 161 path_inode() 161 path_inode_link() 161 proc.h 117 proc_init() 125 proc_scheduler() 126 proc_sig_handler() 127 proc_t 104 121 route_init() 171 route_remote_to_local() 171 route_remote_to_router() 171 route_sort() 171 sb_inode_status() 150 sb_mount() 150 sb_print() 150 sb_reference() 150 sb_save() 150 sb_t 104 149 sb_zone_status() 150 screen_cell() 145 screen_clear() 145 screen_current() 145 screen_init() 145 screen_newline() 145 screen_number() 145 screen_pointer() 145 screen_putc() 145 screen_scroll() 145 screen_select() 145 screen_t 104 screen_update() 145 setjmp() 129 stack.s 108 sti() 113 sysroutine() 124 125 s_chdir() 162 s_chmod() 162 s_chown() 162 163 s_dup() 163 s_dup2() 163 s_fchmod() 163 s_fcntl() 163 s_fstat() 163 s_link() 162 s_longjmp() 129 s_lseek() 163 s_mkdir() 162 s_mknod() 162 s_mount() 162 s_open() 162 s_pipe() 163 s_read() 163 s_setjmp() 129 s_stat() 162 s_umount() 162 s_unlink() 162 s_write() 163 tcp() 174 tcp_close() 174 tcp_connect() 174 tcp_rx_ack() 174 tcp_rx_data() 174 tcp_show() 174 tcp_tx_ack() 174 tcp_tx_raw() 174 tcp_tx_rst() 174 tcp_tx_sock() 174 tty_console() 142 tty_init() 142 tty_read() 142 tty_reference() 142 tty_t 104 tty_write() 142 udp_tx() 174 zno_t 104 zone_alloc() 152 zone_free() 152 zone_print() 152 zone_read() 152 zone_write() 152 _in_16() 112 _in_8() 112 _out_16() 112 _out_8() 112

84.1 Introduzione a os32

os32 è uno studio che applica qualche rudimento relativo ai sistemi operativi, basandosi sull'architettura x86-32 IBM AT, utilizzando come strumenti di sviluppo GNU C, GNU AS e GNU LD, 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.

os32 contiene uno schedulatore banale e molto limitato, un'organizzazione dei processi ad albero e una funzionalità di amministrazione dei segnali, una gestione del file system Minix 1 e delle partizioni in stile Dos (ma solo di quelle primarie), una shell banale e qualche programma di servizio di esempio.

La differenza più importante di os32, rispetto a os16, sta nel fatto che non si utilizzano più le funzioni del BIOS tradizionale, dal momento che si opera in modalità protetta.

84.1.1 Organizzazione

Tutti i file di os32 dovrebbero essere disponibili a partire da *allegati/os32*. In particolare, il file '`disk.hda`' è l'immagine di un vecchio disco PATA, suddiviso in due partizioni: la prima partizione con un file system Dos-FAT, contenente SYSLINUX e il kernel di os32; la seconda contenente un file system Minix 1 con il sistema.

Gli script preparati per os32 prevedono che i file system contenuti nel file-immagine che rappresentano l'unità PATA, in fase di sviluppo si trovino innestati nelle directory '`/mnt/disk.hda.1/`' e '`/mnt/disk.hda.2/`'. Pertanto, se si ricompila os32, tali directory vanno predisposte (oppure vanno modificati gli script con l'organizzazione che si preferisce attuare). Tuttavia, considerato che non è facile lavorare con file-immagine suddivisi in partizioni, altri script aiutano nelle operazioni di manutenzione: '`fdisk`', '`format`', '`mount`', '`umount`'.

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 94.1.2 e 94.1.9), con le opzioni necessarie a operare correttamente.

Per la compilazione del lavoro si usano due script alternativi: '`makeit.mer`' o '`makeit.sep`' (listato 94.1.8). Lo script '`makeit.mer`' conduce una compilazione in cui i file eseguibili degli applicativi sono tali da condividere lo stesso segmento di memoria, sia per il codice, sia per i dati; al contrario, lo script '`makeit.sep`' fa sì che codice e dati siano distinti (il kernel si compila solo in formato ELF). I due script ricreano ogni volta i file-make, basandosi sui file presenti effettivamente nelle varie directory previste; inoltre, alla fine della compilazione, copiano il kernel nella prima partizione del file-immagine del disco PATA (purché risulti innestata come previsto nella directory '`/mnt/disk.hda.1/`') e nella seconda partizione copiano gli applicativi.

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 os32). 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.

84.1.2 Le directory

Gli script descritti nella sezione precedente, si trovano all'inizio della gerarchia prevista per os32. Le directory successive dividono in modo molto semplice le varie componenti per la compilazione:

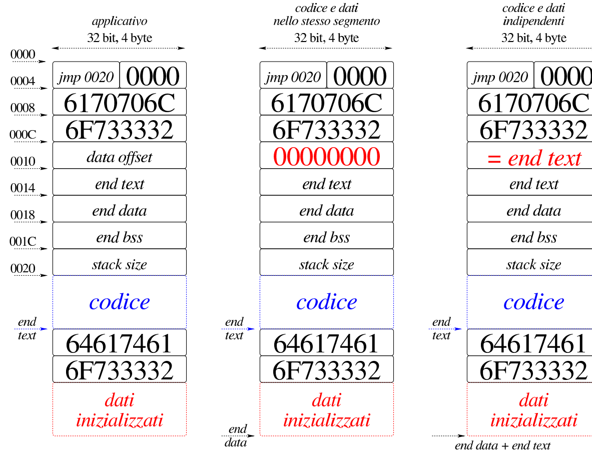
Directory	Contenuto
'`applic/`'	File delle applicazioni da usare con os32.
'`kernel/`'	File per la realizzazione del kernel, inclusi i file di intestazione specifici.
'`lib/`'	File di intestazione generali, file della libreria C per le applicazioni e, per quanto possibile, anche per il kernel.
'`skel/`'	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/_gcc/`' si collocano file contenenti una libreria di funzioni in linguaggio C, necessaria al compilatore GNU C per compiere il proprio lavoro correttamente con valori da 64 bit.

84.1.3 La struttura degli eseguibili

Nell'ottica della massima semplicità, gli eseguibili degli applicativi di os32 hanno una struttura propria, schematizzata dalla figura successiva. Tale struttura viene ottenuta attraverso i file sorgenti '`crt0.mer.s`' o '`crt0.sep.s`', e i file di configurazione di GNU LD '`applic.mer.ld`' o '`applic.sep.ld`'. La sigla '`*.mer.*`' individua la compilazione in un solo segmento, sia per il codice, sia per i dati, mentre la sigla '`*.sep.*`' riguarda la situazione opposta, in cui codice e dati si trovano divisi.

Figura 84.2. Struttura dei file eseguibili degli applicativi di os32.

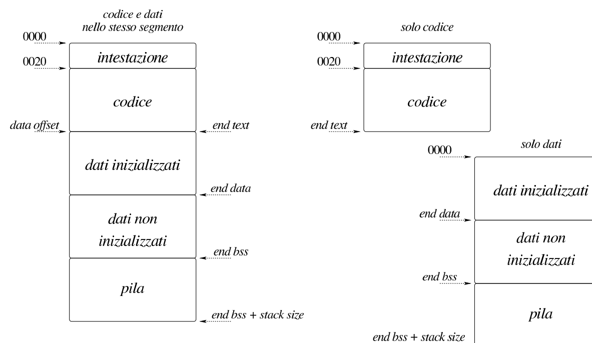


Nella figura si mettono a confronto la struttura dell'eseguibile di un applicativo compilato per avere codice e dati nello stesso segmento, rispetto al caso in cui questi sono separati. Nei primi quattro byte c'è un'istruzione di salto al codice che si trova subito dopo l'intestazione, quindi appare un'impronta di riconoscimento che occupa complessivamente otto byte. Tale impronta è la rappresentazione esadecimale della stringa «os32appl», ma spezzata in due e rovesciata a causa dell'architettura *little endian* (se si legge il file in esadecimale, si vede la sequenza dei caratteri «lppa23so»).

Dopo l'impronta di riconoscimento si trovano, rispettivamente, lo scostamento del segmento dati, espresso in byte, 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. Per distinguere se l'eseguibile è fatto per gestire in un solo segmento codice e dati, oppure se questi devono essere separati, va osservato il valore di ***data_offset***: se questo è zero, significa che il segmento dati parte dall'indirizzo zero, esattamente come il segmento codice, pertanto si trovano nello stesso spazio di indirizzamento. Se invece il valore di ***data_offset*** è diverso da zero, allora deve coincidere con il valore di ***end_text***, in quanto i dati inizializzati si trovano nel file a partire dalla fine del codice, ma devono poi collocarsi in un segmento separato, per cui il valore di ***end data*** è da intendere riferito all'indirizzo iniziale della zona dei dati, ovvero zero.

Nel file eseguibile, la porzione che contiene i dati inizializzati, parte con un'impronta ulteriore, costituita da «os32data», con gli stessi problemi di inversione già descritti per l'intestazione. Lo scopo di questa impronta è semplicemente quello di evitare che ci possano essere dati che iniziano precisamente dall'indirizzo zero, essendo questo riservato per il puntatore nullo.

Figura 84.3. Immagine in memoria dei processi generati dagli eseguibili di os32: a sinistra quelli in cui codice e dati condividono lo stesso segmento di memoria; a destra quelli che usano segmenti distinti.



Il kernel è in formato ELF, ma nella prima parte del codice vie-

ne piazzata un'impronta, secondo le specifiche multiboot (sezione 65.5.1).

Riquadro 84.4. Compilazione degli applicativi con codice e dati separati.

La compilazione degli applicativi che in memoria separano il segmento codice da quello dei dati avviene in modo più complicato rispetto all'altro metodo, perché codice e dati iniziano formalmente dall'indirizzo zero e non è possibile procedere a una compilazione «binaria» normale. Pertanto, in questo caso si crea inizialmente un formato ELF provvisorio; poi, attraverso lo script '`elf-to-os32`' che a sua volta si avvale di '**`objdump`**', vengono individuate le componenti che servono dal file ELF e ricomposte secondo il formato che si aspetta os32.

84.1.4 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/fs/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);
```

84.1.5 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).
`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`**'.
`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.
`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`**'.
`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.

Tipo	Nome	Utilizzo
`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.

84.1.6 Tipi derivati speciali

Nel codice del kernel e nella libreria specifica di os32 si usano dei tipi derivati speciali, riassunti nella tabella successiva.

File di intestazione	Tipo speciale	Descrizione
'`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.
'`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.
'`kernel/ibm_i386.h`'	`gdt_t`	Variabile strutturata, adatta a contenere le informazioni di una voce della tabella GDT.
'`kernel/ibm_i386.h`'	`idt_t`	Variabile strutturata, adatta a contenere le informazioni di una voce della tabella IDT.
'`kernel/ibm_i386.h`'	`idtr_t`	Variabile strutturata, adatta a rappresentare il registro IDTR.
'`kernel/memory.h`'	`addr_t`	Variabile scalare, in grado di rappresentare un indirizzo efficace di memoria (un indirizzo che vada da 00000000_{16} a $FFFFFFFF_{16}$).
'`kernel/multiboot.h`'	`multiboot_t`	Variabile strutturata che riproduce la scomposizione delle informazioni ricevute dal kernel dal sistema di avvio multiboot.
'`kernel/proc.h`'	`proc_t`	Variabile strutturata per rappresentare un elemento della tabella dei processi.
'`kernel/proc.h`'	`header_t`	Variabile strutturata che riproduce la suddivisione delle informazioni contenute nella parte iniziale degli eseguibili di os32.
'`kernel/driver/ata.h`'	`ata_t`	Variabile strutturata per rappresentare un elemento della tabella delle unità ATA.
'`kernel/driver/ata.h`'	`ata_sector_t`	Variabile strutturata per rappresentare un settore di dati relativo alla gestione ATA.
'`kernel/driver/kbd.h`'	`kbd_t`	Variabile strutturata per rappresentare complessivamente lo stato della tastiera, incorporando anche la modalità di interpretazione corretta della mappa attuale.
'`kernel/driver/screen.h`'	`screen_t`	Variabile strutturata per rappresentare il contenuto di uno schermo e la collocazione del cursore; tale variabile strutturata compone un elemento della tabella degli schermi gestiti simultaneamente.
'`kernel/driver/tty.h`'	`tty_t`	Variabile strutturata, adatta a contenere le informazioni e lo stato di un terminale, che costituisce in pratica un elemento della tabella dei terminali.
'`kernel/blk.h`'	`blk_cache_t`	Variabile strutturata, adatta a contenere un blocco di memoria (per i dispositivi a blocchi) e le informazioni che lo riguardano, come elemento della tabella che costituisce la memoria tampone per l'accesso alle unità di memorizzazione.
'`lib/sys/os32.h`'	`h_addr_t`	Variabile scalare a 32 bit, contenente un indirizzo IPv4 in *host byte order*, ovvero rappresentato secondo l'achitettura della CPU, per ciò che riguarda l'ordine dei byte.

84.2 Caricamento ed esecuzione del kernel

Il kernel di os32 è compilato in formato ELF, secondo le specifiche multiboot, in modo da poter essere avviato da un sistema come GRUB 1 o SYSLINUX. Il codice del kernel inizia nel file '`crt0.s`', dove a un certo punto viene eseguita la funzione ***kmain()***, nella quale si sintetizza il funzionamento del kernel stesso. Il sistema di avvio colloca il kernel a partire dall'indirizzo 100000_{16} (1 Mibyte), già in modalità protetta, di conseguenza il codice è organizzato per iniziare da tale posizione.

84.2.1 Multiboot

os32 è conforme alle specifiche multiboot per consentirne l'avvio attraverso GRUB 1 o SYSLINUX, senza doversi prendere carico dei problemi relativi al passaggio alla modalità protetta. Perché il file del kernel sia riconosciuto come aderente a tali specifiche, contiene un'impronta di riconoscimento, definita *multiboot header*, collocata nella parte iniziale, come dichiarato nel file '`kernel/main/crt.s`', entro i primi 8 Kibyte.

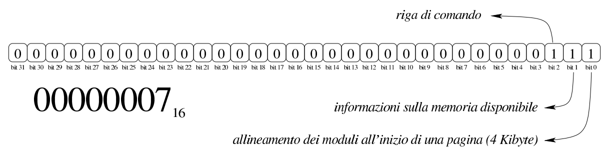
Figura 84.9. La prima parte obbligatoria dell'intestazione multiboot.

Il primo campo da 32 bit, definito ***magic***, contiene l'impronta di riconoscimento vera e propria, costituita precisamente dal numero 1BADB002$_{16}$. Il secondo campo da 32 bit, definito ***flags***, contiene degli indicatori con i quali si richiede un certo comportamento al sistema di avvio. Il terzo campo da 32 bit, definito ***checksum***, contiene un numero calcolato in modo tale che la somma tra i numeri contenuti nei tre campi da 32 bit porti a ottenere zero, senza considerare i riporti.

I nomi indicati sono quelli definiti dallo standard e, come si vede, il campo ***checksum*** si ottiene calcolando −(***magic*** + ***flags***), dove si deve intendere che i calcoli avvengono con valori interi senza segno e si ignorano i riporti.

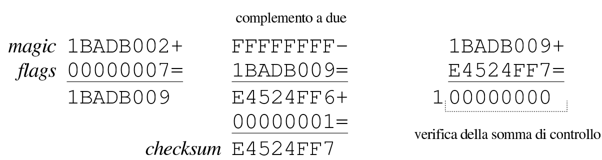
Dal momento che il kernel da avviare è in formato ELF, le informazioni che il sistema di avvio necessita per piazzarlo correttamente in memoria e per passare il controllo allo stesso, sono già disponibili e non c'è la necessità di occuparsi di altri campi facoltativi che possono seguire i tre già descritti. Stante questa semplificazione, per quanto riguarda il campo ***flags***, os32 utilizza precisamente il valore 00000003$_{16}$, con il significato che si vede nella figura successiva.

Figura 84.10. Il campo ***flags*** e il suo utilizzo fondamentale.



Il bit meno significativo del campo ***flags***, se impostato a uno, serve a richiedere il caricamento in memoria dei moduli eventuali (assieme al file-immagine principale) in modo che risultino allineati all'inizio di una «pagina» (ovvero all'inizio di un blocco da 4 Kibyte). os32 non prevede moduli, tuttavia richiede ugualmente questa opzione. Il secondo bit del campo ***flags*** serve a richiedere al sistema di avvio di passare le informazioni disponibili sulla memoria, le quali poi vengono rese disponibili a partire da un'area a cui punta inizialmente il registro ***EBX***.

Figura 84.11. Calcolo del campo ***checksum***.



Quando il sistema di avvio passa il controllo al kernel, dopo averlo caricato in memoria: il microprocessore è in modalità protetta; il registro ***EAX*** contiene il numero 2BADB002$_{16}$; il registro ***EBX*** contiene l'indirizzo fisico, a 32 bit, di una sequenza di campi contenenti informazioni passate dal sistema di avvio (*multiboot information structure*), come si vede nella figura successiva.

Figura 84.12. Inizio della struttura di informazioni offerta da un sistema di avvio aderente alle specifiche *multiboot*.

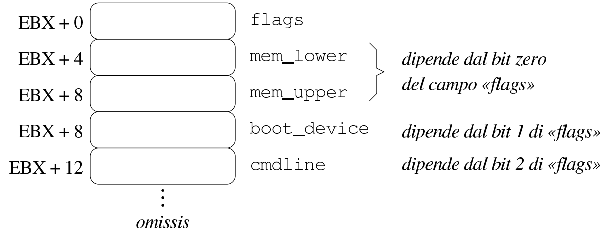


Tabella 84.13. Descrizione dei primi campi della struttura informativa fornita dal sistema di avvio *multiboot*.

Nome mnemonico del campo	bit del campo '`flags`' da cui dipende	Descrizione
`flags`		Il primo campo definisce degli indicatori, con i quali si dichiara se una certa informazione, successiva, viene fornita ed è valida.
`mem_lower` `mem_upper`	0	Se è attivo il bit meno significativo del campo '`flags`', i campi '`mem_lower`' e '`mem_upper`' contengono la dimensione della memoria bassa (da zero a un massimo di 640 Kibyte) e della memoria alta (quella che si trova a partire da un mebibyte). La dimensione è da intendersi in kibibyte (simbolo Kibyte) e, per quanto riguarda la memoria alta, viene indicata solo la dimensione continua fino al primo «buco».
`boot_device`	1	Se è attivo il secondo bit, partendo dal lato meno significativo, il campo '`boot_device`' dà informazioni sull'unità di avvio. L'informazione è divisa in quattro byte, come descritto nelle specifiche *multiboot*.
`cmdline`	2	Se è attivo il terzo bit, partendo dal lato meno significativo, il campo '`cmdline`' contiene l'indirizzo iniziale di una stringa che riproduce la riga di comando passata al kernel.

Come si può intuire leggendo la tabella che descrive i primi cinque campi, il significato dei bit del campo '`flags`' viene attribuito, mano a mano che l'aggiornamento delle specifiche prevede l'espansione della struttura informativa. Per esempio, un campo '`flags`' con il valore 100$_{2}$ sta a significare che esistono i campi fino a '`cmdline`' e il contenuto di quelli precedenti non è valido, ma i campi successivi, non esistono affatto. La comprensione di questo concetto dovrebbe rendere un po' più semplice la lettura delle specifiche.

Tabella 84.14. Funzioni per la gestione delle specifiche multiboot all'interno di os32.

Funzione	Descrizione
`void` `mboot_save (multiboot_t *`***mboot_data***`);`	Salva le informazioni multiboot all'interno della variabile strutturata pubblica ***multiboot***. Listati 94.11, 94.11.2 e 94.11.3.

Funzione	Descrizione
`char ** mboot_cmdline_opt (const char *opt, const char *delim);`	Scandisce la stringa delle opzioni salvata all'interno di ***multiboot.cmdline***, alla ricerca di un'opzione il cui nome corrisponda alla stringa. Dopo il nome dell'opzione deve apparire il segno '**=**' e dopo devono trovarsi i valori associati all'opzione, separati da ***delim***. Questi valori vengono restituiti in forma di array di stringhe, dove l'ultima stringa si riconosce perché vuota. Listati 94.11, 94.11.2 e 94.11.1.

84.2.2 File «kernel.ld», «kernel/main/crt0.s» e «kernel/main/stack.s»

Listati 94.1.7, 94.9.2 e 94.9.6.

Il codice del kernel inizia dal file '`crt.s`'; tuttavia, per la sua corretta interpretazione, va considerato prima il file di configurazione di GNU LD (il collegatore, ovvero il *linker*), costituito dal file '`kernel.ld`', sintetizzabile così:

```
ENTRY (kstartup)
SECTIONS {
    . = 0x00100000;
    …
}
```

Si osserva subito che il punto di inizio del codice, descritto successivamente dal file '`crt.s`', deve corrispondere alla posizione dell'etichetta '**`kstartup`**' e che quel punto deve trovarsi all'indirizzo 100000_{16}, ovvero quello in cui il sistema di avvio lo colloca.

Nel file '`crt.s`', dopo il preambolo in cui si dichiarano i simboli esterni e quelli interni da rendere pubblici, si parte proprio con l'etichetta '**`kstartup`**', e da lì si salta a un'altra posizione ('**`start`**'), per lasciare spazio all'intestazione multiboot.

```
…
.section .text
kstartup:
    jmp start
.align 4
multiboot_header:
    .int 0x1BADB002                          # magic
    .int 0x00000003                          # flags
    .int -(0x1BADB002 + 0x00000003)          # checksum
start:
    …
```

L'immagine del kernel in memoria utilizza un solo segmento per codice e dati, suddividendosi nel modo consueto: codice, dati inizializzati, dati non inizializzati e pila. Per individuare le varie componenti, il file '`kernel.ld`' inserisce dei nomi a cui è possibile fare riferimento nel codice; inoltre, viene utilizzato il file '`stack.s`' per definire lo spazio usato per la pila dei dati.

Figura 84.17. Immagine del kernel in memoria, a partire dall'indirizzo 100000_{16}, evidenziando le etichette dichiarate nei file '`kernel.ld`', '`crt0.s`' e '`stack.s`'.

A partire dall'indirizzo corrispondente all'etichetta '**`start`**', nel file '`crt0.s`' inizia il lavoro preliminare del kernel. Per prima cosa viene attivata la pila dei dati, collocando nel registro ***ESP*** l'indirizzo corrispondente alla fine della stessa, ovvero '**`_k_stack_bottom`**':

```
    movl $_k_stack_bottom, %esp
```

Quindi si azzera il registro EFLAGS, sfruttando per questo la pila appena attivata:

```
    pushl $0
    popf
```

Infine si chiama la funzione ***kmain()*** (del file '`kmain.c`'), fornendo come argomenti la firma di riconoscimento del sistema multiboot, contenuta nel registro EAX, e il puntatore alla struttura contenente le informazioni fornite dal sistema multiboot, contenuto nel registro EBX:

```
    pushl %ebx  # multiboot_t *info;
    pushl %eax  # uint32_t      magic;
    call kmain  # void kmain (uint32_t magic,
                #             multiboot_t *info);
```

Se ci dovesse essere una conclusione della funzione ***kmain()***, si passerebbe al codice successivo, il quale si limita a mettere a riposo la CPU:

```
halt:
    hlt
    jmp halt
```

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

Listato 94.9 e successivi.

Tutto il lavoro del kernel di os32 si sintetizza nella funzione ***kmain()***, contenuta nel file '`kernel/main/kmain.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 si chiama ***kmain()*** (e non ***main()***), perché non è conforme allo schema che dovrebbe avere la prima funzione di un programma per sistemi POSIX. Come già accennato a proposito del file '`crt0.s`', la funzione ***kmain()*** prevede come parametri un codi-

ce di riconoscimento e il puntatore a delle informazioni, forniti dal sistema di avvio.

```
…
void
kmain (uint32_t magic, multiboot_t *mboot_data)
{
  …
  tty_init ();
  if (magic == 0x2BADB002)
    {
      mboot_save (mboot_data);
      k_printf ("os32 build %s ram %i Kibyte\n",
                BUILD_DATE, (int) multiboot.mem_upper);
      mb_size (multiboot.mem_upper * 1024);
      kbd_load ();
      blk_cache_init ();
      fs_init ();
      proc_init ();
    }
  else
    {
      …
      k_exit ();
    }
  menu ();
  …
}
```

Dopo la dichiarazione delle variabili si inizializza la gestione del video della console (funzione ***tty_init()***), si verifica che il codice sia stato avviato da un sistema di avvio multiboot e se ne salvano le informazioni (funzione ***mboot_save()***), quindi si mostra un messaggio iniziale, si imposta la dimensione massima della memoria disponibile in base ai dati ottenuti dal sistema multiboot (funzione ***mb_size()***), si configura la tastiera (funzione ***kbd_load()***), si inizializza la gestione della memoria tampone (funzione ***blk_cache_init()***), del file system (funzione ***fs_init()***) e dei processi elaborativi (***proc_init()***). Fatto tutto questo appare un menù (funzione ***menu()***) e si passa a una fase successiva.

```
…
void
kmain (uint32_t magic, multiboot_t *mboot_data)
{
  …
  menu ();
  for (exit = 0; exit == 0;)
    {
      sys (SYS_0, NULL, 0);
      …
      if …
        …
      else if (strncmp (command, "h", MAX_CANON) == 0)
        {
          menu ();
        }
      else if (strncmp (command, "x", MAX_CANON) == 0)
        …
      else if (strncmp (command, "q", MAX_CANON) == 0)
        {
          k_printf ("System halted!\n");
          return;
        }
    }
}
```

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 comando dalla tastiera, costituito però da un solo carattere: se risulta selezionato un comando previsto, il kernel esegue quanto richiesto e poi 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:

Comando	Risultato
`h`	Mostra il menù di funzioni disponibili.
`t`	Mostra i valori gestiti internamente dell'orologio del kernel.
`f`	Esegue una biforcazione del kernel, nella quale, il processo figlio si limita a mostrare ripetutamente il proprio numero di processo.
`g` `G`	Mostra le prime voci della tabella GDT, in binario.
`i` `I`	Mostra le prime voci della tabella IDT, in binario.
`m`	Mostra la mappa della memoria, elencando le aree continue utilizzate.
`p`	Mostra la situazione dei processi e altre informazioni.
`s`	Mostra delle informazioni sul super blocco.
`n`	Mostra l'elenco degli inode attivi.
`1`	Invia il segnale '**SIGKILL**' al processo numero uno.
`2`|`3`|`4`|`5`| `6`|`7`|`8`|`9`|`A`| `B`|`C`|`D`|`E`|`F`	Invia il segnale '**SIGTERM**' al processo con il numero corrispondente, da 2 a 15.
`a`|`b`|`c`	Avvia il programma '`/bin/aaa`', '`/bin/bbb`' o '`/bin/ccc`'.
`x`	Termina il ciclo e successivamente si passa all'avvio di '`/bin/init`'.
`q`	Ferma il sistema.

Premendo [*x*], 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.

```
      else if (strncmp (command, "x", MAX_CANON) == 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 84.26. Aspetto di os32 in funzione mentre visualizza la tabella dei processi avviati e la mappa della memoria.

```
c
abaabaaba
p
pp  p pg                                 T * 0x1000 D * 0x1000 stack
id id rp  tty  uid euid suid usage s addr  size addr  size pointer  name
 0  0  0 0000    0    0    0 00.03 R 00000 028e 00000 0000 028eb2c os32 kernel
 0  1  0 0000    0    0    0 00.09 r 0051e 000e 0052c 002d 002cf88 /bin/ccc
 1  2  0 0000   10   10   10 00.00 s 002bc 000e 002ca 002d 002cf34 /bin/aaa
 1  3  0 0000   11   11   11 00.00 s 002f7 000e 00305 002d 002cf34 /bin/bbb
ab
m
Hex mem map, blocks of 1000: 0-28f 2bc-332 51e-559
aabaab_
```

Figura 84.27. Aspetto di os32 in funzione con il menù in evidenza, dopo aver dato il comando '**x**' per avviare '**init**'.

```
os32 build 20AAMMGGHHmm ram 130048 Kibyte
[ata_init] ATA drive 0 size 8064 Kib
[ata_drq] ERROR: drive 2 error
[dm_init] ATA drive=0 total sectors=16128
[dm_init] partition type=0c start sector=63 total sectors=2961
[dm_init] partition type=81 start sector=3024 total sectors=13104
.----------------------------------------------------------------.
| h    show this menu                          .--------------.|
| t    show internal timer values              | all commands ||
| f    fork the kernel                         | followed by  ||
| m    memory map (HEX)                        | [Enter]      ||
| g|G  show GDT table first 21+21 items        `--------------'|
| i|I  show IDT table first 21+21 items                        |
| p    process status list                                     |
| s    super block list                                        |
| n    list of active inodes                                   |
| 1..9 kill process  1 to 9                                    |
| A..F kill process 10 to 15                                   |
| a..c run programs '/bin/aaa' to '/bin/ccc' in parallel       |
| x    exit interaction with kernel and start '/bin/init'      |
| q    quit kernel                                             |
`----------------------------------------------------------------'
x
init
os32: 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:
```

84.3 Funzioni interne legate all'hardware, nei file «kernel/ibm_i386.h» e «kernel/ibm_i386/...»

Listato 94.6 e successivi.

Il file '`kernel/ibm_i386.h`' e quelli contenuti nella directory '`kernel/ibm_i386/`', raccolgono il codice del kernel che è legato strettamente all'hardware, escludendo però la gestione dei dispositivi. Tra le altre spiccano particolarmente le funzioni per la gestione dei segmenti di memoria (la tabella GDT), delle interruzioni (la tabella IDT) e l'attivazione delle routine associate alle interruzioni (ISR).

Alcune delle funzioni scritte in linguaggio assemblatore hanno nomi che iniziano con un trattino basso, ma a fianco di queste sono 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.

84.3.1 Funzioni per l'input e l'output con le porte interne

Alcune funzioni e macroistruzioni di questo gruppo sono destinate a facilitare l'input e l'output con le porte interne dell'architettura x86.

Tabella 84.28. Funzioni e macroistruzioni per l'input e l'output con le porte interne x86.

Funzione o macroistruzione	Descrizione
`uint32_t _in_8 (uint32_t port);` `unsigned int in_8 (port);`	Legge un valore a 8 bit da una porta. Listati 94.6 e 94.6.3.
`uint32_t _in_16 (uint32_t port);` `unsigned int in_16 (port);`	Legge un valore a 16 bit da una porta. Listati 94.6 e 94.6.1.
`void _out_8 (uint32_t port, uint32_t value);` `void out_8 (port, value);`	Scrive un valore a 8 bit in una porta. Listati 94.6 e 94.6.6.
`void _out_16 (uint32_t port, uint32_t value);` `void out_16 (port, value);`	Scrive un valore a 16 bit in una porta. Listati 94.6 e 94.6.4.

84.3.2 Funzioni accessorie alla gestione delle interruzioni hardware

Alcune funzioni di questo gruppo sono destinate a facilitare il controllo delle interruzioni hardware che raggiungono la CPU.

Tabella 84.29. Funzioni accessorie per il controllo delle interruzioni hardware.

Funzione o macroistruzione	Descrizione
`void cli (void);`	Disabilita le interruzioni hardware attraverso l'azzeramento dell'indicatore relativo nel registro EFLAGS. Listati 94.6 e 94.6.7.
`void sti (void);`	Abilita le interruzioni hardware attraverso l'attivazione dell'indicatore relativo nel registro EFLAGS. Listati 94.6 e 94.6.27.
`void irq_on (unsigned int irq);`	Abilita selettivamente l'interruzione hardware indicata per numero (da zero a 16). Listati 94.6 e 94.6.20.
`void irq_off (unsigned int irq);`	Disabilita selettivamente l'interruzione hardware indicata per numero (da zero a 16). Listati 94.6 e 94.6.19.

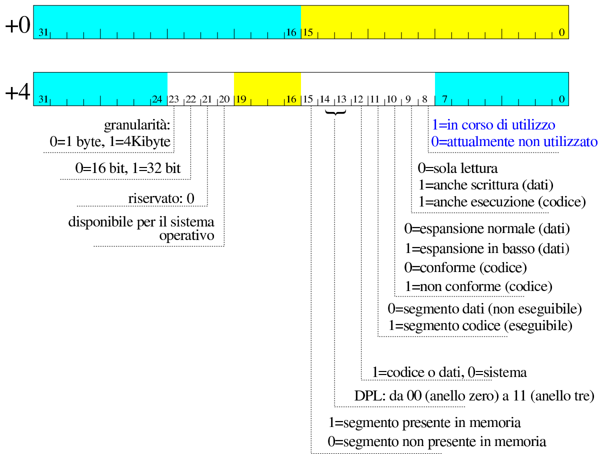
84.3.3 Gestione della tabella GDT

Nel momento in cui il codice del kernel prende il controllo, il microprocessore si trova già a funzionare in modalità protetta, attraverso una tabella GDT già impostata per gestire la memoria in modo lineare, senza particolari accorgimenti. Quando il kernel inizializza la gestione dei processi (funzione ***proc_init()***) costruisce una nuova tabella GDT, nella quale, per ogni processo gestibile, predispone due elementi, per descrivere rispettivamente il segmento codice e il segmento dati di un processo. In pratica, la nuova tabella GDT è composta da una prima voce nulla, obbligatoria, da una coppia di voci che descrivono il segmento codice e dati del kernel, da altre coppie di voci, modificate poi durante il funzionamento, per descrivere i segmenti dei processi.

Tutti i processi vedono la memoria con un indirizzamento che corrisponde a quello reale; tuttavia, disponendo ognuno di una propria coppia di voci nella tabella GDT, è possibile controllarne l'uso in modo da impedire che possano raggiungere aree al di fuori della propria competenza.

La tabella GDT è rappresentata in C dall'array ***gdt_table[]*** dichiarato nel file '`kernel/ibm_i386/gdt_public.c`' (listato 94.6.11), composto da elementi di tipo '**`gdt_t`**' (listato 94.6).

```
typedef struct {
    uint32_t  limit_a              : 16,
              base_a               : 16;
    uint32_t  base_b               : 8,
              accessed             : 1,
              write_execute        : 1,
              expansion_conforming : 1,
              code_or_data         : 1,
              code_data_or_system  : 1,
              dpl                  : 2,
              present              : 1,
              limit_b              : 4,
              available            : 1,
              reserved             : 1,
              big                  : 1,
              granularity          : 1,
              base_c               : 8;
} gdt_t;
```



La tabella viene creata con una quantità di elementi pari al valore della macro-variabile ***GDT_ITEMS***. Sapendo che la prima voce è obbligatoriamente nulla, che se ne usano altre due per il kernel e che ogni processo utilizza due voci della tabella, si possono gestire al massimo (***GDT_ITEMS***−3)/2 processi.

La struttura di ogni elemento della tabella GDT è molto complessa, pertanto, per scriverci un nuovo valore si usa la funzione ***gdt_segment()*** che si occupa di spezzettare e ricollocare i dati come richiesto dal microprocessore (listato 94.6.12)

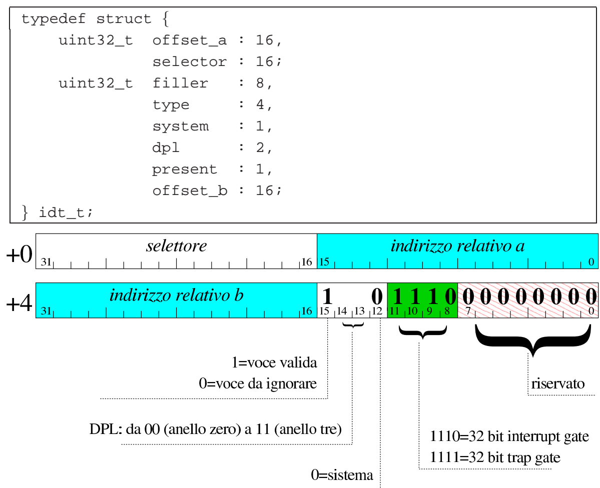
Tabella 84.32. Funzioni per la gestione della tabella GDT.

Funzione	Descrizione
`void gdt_segment (int` ***segment***`, uint32_t` ***base***`, uint32_t` ***limit***`, bool` ***present***`, bool` ***code***`, unsigned char` ***dpl***`);`	Scrive una voce della tabella GDT, sezionando e ricomponendo i dati come richiesto dal microprocessore. Listato 94.6.12.
`void gdt (void);`	Predispone e attiva la tabella GDT; per questo si avvale in modo particolare delle funzioni ***gdt_segment()*** e di ***gdt_load()***. Listato 94.6.8.
`void gdt_load (void *`***gdtr***`);`	Fa sì che il microprocessore carichi la tabella GDT, a partire dal puntatore al registro GDTR; registro che contiene l'informazione della collocazione in memoria della tabella GDT e della sua estensione. Listato 94.6.9.
`void gdt_print (void *`***gdtr***`, unsigned int` ***first***`, unsigned int` ***last***`);`	Funzione diagnostica, usata per visualizzare il contenuto della tabella GDT in binario. Listato 94.6.10.

84.3.4 Gestione della tabella IDT

La tabella IDT serve al microprocessore per conoscere quali procedure avviare al verificarsi delle interruzioni. La funzione ***idt()*** si occupa di predisporre la tabella e di attivarla, ma prima di ciò si prende cura di posizionare le interruzioni hardware a partire dalla voce 32 (la 33-esima). Le procedure a cui fa riferimento la tabella IDT creata con la funzione ***idt()*** sono dichiarate nel file '`kernel/ibm_i386/isr.s`', descritto però nella sezione successiva.

La tabella IDT è rappresentata in C dall'array ***idt_table[]*** dichiarato nel file '`kernel/ibm_i386/idt_public.c`' (listato 94.6.18), composto da elementi di tipo '**`idt_t`**' (listato 94.6).



La tabella viene creata con 129 elementi, anche se più della metà non vengono usati; tuttavia, proprio l'ultimo, corrispondente all'interruzione 128_{10}, ovvero 80_{16}, serve per le chiamate di sistema.

La struttura di ogni elemento della tabella IDT è un po' complicata, pertanto, per scriverci un nuovo valore si usa la funzione ***idt_descriptor()*** che si occupa di spezzettare e ricollocare i dati come richiesto dal microprocessore (listato 94.6.14)

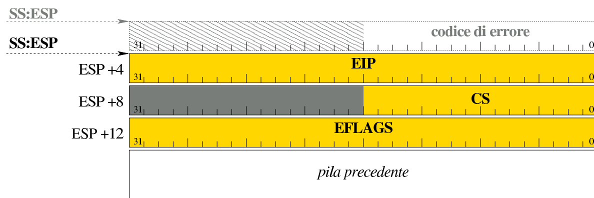
Tabella 84.35. Funzioni per la gestione della tabella IDT.

Funzione	Descrizione
`void idt_descriptor (int` ***desc***`, void *`***isr***`, uint16_t` ***selector***`, bool` ***present***`, char` ***type***`, char` ***dpl***`);`	Scrive una voce della tabella IDT, sezionando e ricomponendo i dati come richiesto dal microprocessore. Listato 94.6.14.
`void idt (void);`	Predispone e attiva la tabella IDT; per questo si avvale in modo particolare delle funzioni ***idt_descriptor()*** e di ***idt_load()***. Listato 94.6.13.
`void idt_load (void *`***idtr***`);`	Fa sì che il microprocessore carichi la tabella IDT, a partire dal puntatore al registro IDTR; registro che contiene l'informazione della collocazione in memoria della tabella IDT e della sua estensione. Listato 94.6.16.
`void idt_irq_remap (unsigned int` ***offset_1***`, unsigned int` ***offset_2***`);`	Modifica la mappatura delle interruzioni hardware (IRQ) spostando il primo gruppo a partire dal valore di ***offset_1*** e il secondo gruppo a partire da ***offset_2***. Listato 94.6.15.
`void idt_print (void *`***idtr***`, unsigned int` ***first***`, unsigned int` ***last***`);`	Funzione diagnostica, usata per visualizzare il contenuto della tabella IDT in binario. Listato 94.6.17.

84.3.5 Gestione delle interruzioni

Le interruzioni che individua il microprocessore (eccezioni, interruzioni software e interruzioni hardware) fanno interrompere l'attività normale dello stesso, costringendolo ad accumulare nella pila attuale dei dati lo stato di alcuni registri ed eventualmente di un codice di

errore, saltando poi alla posizione di codice indicata nella voce corrispondente nella tabella IDT. Va osservato che, per semplicità, os32 fa lavorare i propri processi nell'anello zero, come il kernel, per cui i dati accumulati nella pila si limitano a quelli della figura successiva, perché non c'è mai un passaggio da un livello di privilegio a un altro.



Le posizioni del codice a cui il microprocessore deve saltare, secondo le indicazioni della tabella IDT, sono contenute tutte nel file '`kernel/ibm_i386/isr.s`', mentre nel file '`kernel/ibm_i386.h`' vi si fa riferimento attraverso dei prototipi di funzione, benché non si tratti propriamente di funzioni.

Tabella 84.37. Funzioni per la gestione delle interruzioni.

Funzione	Descrizione
`void isr_n (void);`	Si tratta di procedure attivate dalle interruzioni, dove per esempio ***isr_33()*** viene eseguita a seguito del verificarsi dell'interruzione numero 33, la quale ha origine da IRQ 1, ovvero dalla tastiera. L'indicazione di quale procedura attivare per ogni interruzione dipende dalla configurazione della tabella IDT. Listato 94.6.21.
`void isr_exception_unrecoverable (uint32_t eax, uint32_t ecx, uint32_t edx, uint32_t ebx, uint32_t ebp, uint32_t esi, uint32_t edi, uint32_t ds, uint32_t es, uint32_t fs, uint32_t gs, uint32_t interrupt, uint32_t error, uint32_t eip, uint32_t cs, uint32_t eflags);`	Questa funzione viene usata all'interno del file '`isr.s`' per segnalare il verificarsi di un'eccezione non risolvibile, come nel caso di una divisione per zero. Pertanto, la funzione ha soprattutto un significato diagnostico. Listato 94.6.23.
`char *isr_exception_name (int exception);`	Restituisce il puntatore alla stringa contenente il nome dell'eccezione corrispondente al numero di interruzione fornito. Viene usata da ***isr_exception_unrecoverable()*** per dare delle indicazioni comprensibili sull'eccezione che si è verificata. Listato 94.6.22.
`void isr_irq_clear (uint32_t idtn);`	Avvisa il PIC (*programmable interrupt controller*) che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Tuttavia, essendoci due PIC, la funzione stabilisce quale dei due è coinvolto direttamente e di conseguenza come procedere. Listato 94.6.24.
`void isr_irq_clear_pic1 (void);`	Avvisa il PIC1 che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Listato 94.6.25.
`void isr_irq_clear_pic2 (void);`	Avvisa il PIC2 che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Listato 94.6.26.

84.4 Gestione dei processi

Listato 94.6.21; listato 94.14 e successivi.

La gestione dei processi è raccolta nei file '`kernel/proc.h`' e '`kernel/proc/...`'; tuttavia, dal file '`kernel/ibm_i386/isr.s`' hanno origine le procedure attivate dalle interruzioni e dalle chiamate di sistema: le chiamate di sistema e le interruzioni provenienti dal temporizzatore interno provocano l'attivazione dello schedulatore.

Con os32, 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 ***kmain()*** è il kernel che cede volontariamente il controllo a un altro processo (ammesso che ci sia) con una chiamata di sistema nulla.

84.4.1 File «kernel/ibm_i386/isr.s»

Il file '`kernel/ibm_i386/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.

```
.section .data
proc_syscallnr:             .int   0x00000000
proc_msg_offset:            .int   0x00000000
proc_msg_size:              .int   0x00000000
proc_instruction_pointer:   .int   0x00000000
proc_back_address:          .int   0x00000000
_ksp:                       .int   0x00000000
syscall_working:            .int   0x00000000
_clock_kernel:
kticks_lo:                  .int   0x00000000
kticks_hi:                  .int   0x00000000
_clock_time:
tticks_lo:                  .int   0x00000000
tticks_hi:                  .int   0x00000000
```

Si tratta di variabili scalari da 32 bit, tenendo conto che: i simboli '**`kticks_lo`**' e '**`kticks_hi`**' compongono assieme la variabile ***_clock_kernel*** a 64 bit per il linguaggio C; i simboli '**`tticks_lo`**' e '**`tticks_hi`**' compongono assieme la variabile ***_clock_time*** a 64 bit per il linguaggio C.

Dopo la dichiarazione delle variabili inizia il codice vero e proprio,

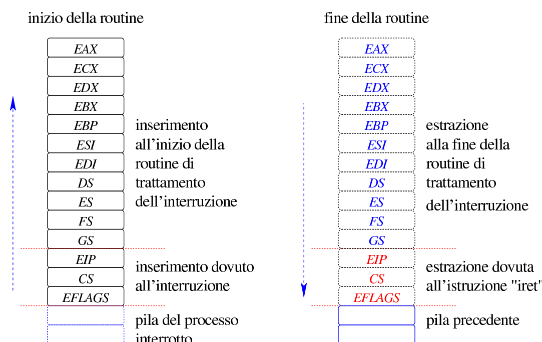
dove i simboli '**`isr_n`**' si riferiscono al codice da usare in presenza dell'interruzione ***n***. Tra tutte, le interruzioni più importanti sono quelle del temporizzatore (***isr_32()***), il quale produce un impulso a circa 100 Hz; quelle della tastiera (***isr_33()***) e delle chiamate di sistema (***isr_128()***).

Il codice per la gestione dei tre tipi di interruzione più importanti ha delle similitudini che conviene analizzare simultaneamente. os32 non cambia mai anello, nel senso che il livello di privilegio dei processi è pari a quello del kernel; pertanto, nel momento in cui si verifica un'interruzione, la pila e il segmento dati in essere sono quelli del processo interrotto. Le procedure che gestiscono le tre interruzioni principali iniziano con il salvataggio dei registri nella pila attuale e il passaggio al segmento dei dati del kernel, lasciando temporaneamente la pila nel segmento dati del processo interrotto; nello stesso modo, terminano con il ripristino del segmento dati originario (al momento dell'interruzione) e il ripristino successivo dei registri, estraendone i valori dalla pila:

```
#
# Save into process stack:
#
pushl %gs
pushl %fs
pushl %es
pushl %ds
pushl %edi
pushl %esi
pushl %ebp
pushl %ebx
pushl %edx
pushl %ecx
pushl %eax
#
# Set the data segments to the kernel data segment,
# so that the following variables can be accessed.
#
mov  $16, %ax # DS, ES, FS and GS.
mov  %ax, %ds
mov  %ax, %es
mov  %ax, %fs
mov  %ax, %gs
…
…
#
# Restore from process stack.
#
popl %eax
popl %ecx
popl %edx
popl %ebx
popl %ebp
popl %esi
popl %edi
popl %ds
popl %es
popl %fs
popl %gs
…
iret
```

Il segmento dati del kernel si trova nella terza voce della tabella GDT (la prima è nulla, la seconda è per il codice del kernel, la terza è per i dati del kernel). Sapendo che ogni voce occupa 8 byte (64 bit), per raggiungere l'inizio della terza voce occorre indicare il valore 16 nel registro di segmento.

Figura 84.40. Inserimento nella pila del processo interrotto.



Durante l'elaborazione di un'interruzione proveniente dal temporizzatore o dalla tastiera, è necessario sapere se è già in corso l'elaborazione di una chiamata di sistema. Se ciò accade, l'impulso del temporizzatore viene recepito, incrementando i contatori, ma non viene fatto altro, mentre l'impulso della tastiera viene semplicemente ignorato.

```
#
# Check if a system call is already working: if so,
# just leave (go to L2).
#
cmpl $1, syscall_working
je L2
```

In pratica, quando si presenta una chiamata di sistema, inizialmente viene assegnato il valore uno alla variabile ***syscall_working***, mentre alla fine del suo compito questa variabile viene azzerata:

```
#
# Tell that it is a system call.
#
movl $1, syscall_working
…
#
# End of system call.
#
movl $0, syscall_working
```

Quando l'interruzione proviene dal temporizzatore e non è in corso l'esecuzione di una chiamata di sistema, oppure quando l'interruzione deriva proprio da una chiamata di sistema, viene attivato lo schedulatore (direttamente o indirettamente, attraverso la funzione che svolge il lavoro richiesto dalla chiamata di sistema), ma per fare questo, è necessario passare alla pila dei dati del kernel, per poi ripristinarla successivamente:

```
#
# Save process stack registers into kernel data segment.
#
mov %ss,  proc_stack_segment_selector
mov %esp, proc_stack_pointer
…
#
# Switch to kernel stack.
#
mov $16, %ax
mov %ax, %ss
mov _ksp, %esp
…
…
#
# Restore process stack registers from kernel data
# segment.
#
mov proc_stack_segment_selector, %ss
mov proc_stack_pointer, %esp
```

Figura 84.44. Scambi delle pile: prima fase.

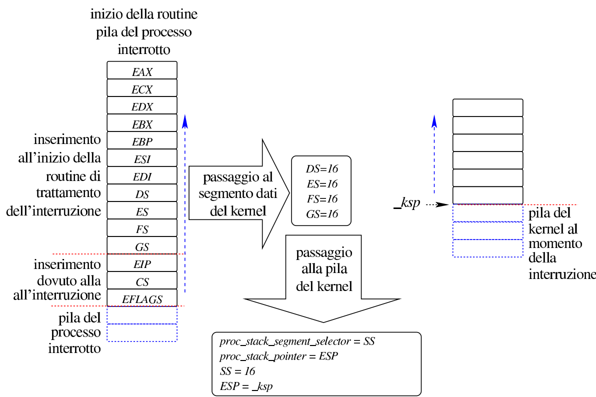
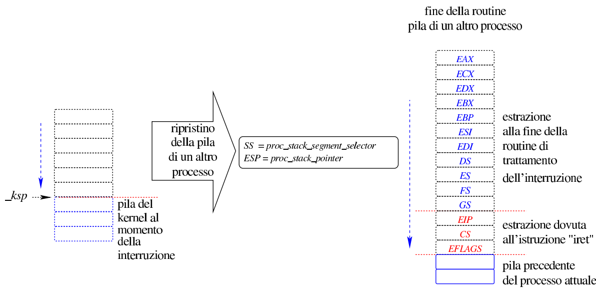


Figura 84.45. Scambi delle pile: seconda fase.



84.4.1.1 Particolarità della routine «isr_32», ovvero «irq_timer»

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, la routine di gestione delle interruzioni del temporizzatore si occupa di incrementare i contatori degli impulsi. Gli impulsi giungono alla frequenza di 100 Hz circa, per cui non c'è la necessità di fare alcun tipo di conversione:

```
...
isr_32:          # IRQ 0: «timer»
    cli
    jmp irq_timer
...
irq_timer:
    ...
    add $1, kticks_lo    # Kernel ticks counter.
    adc $0, kticks_hi    #
    #
    add $1, tticks_lo    # Clock ticks counter.
    adc $0, tticks_hi    #
    ...
```

A questo punto, se l'interruzione è avvenuta mentre era in corso l'elaborazione di una chiamata di sistema, tutto si conclude con il ripristino dei registri e del PIC1, in modo da consentire la ripresa delle interruzioni. Se invece l'interruzione è avvenuta in una situazione differente, si verifica ancora che non sia stato interrotto il funzionamento del kernel stesso, perché se così fosse, anche in questo caso la procedura termina con il solito ripristino dei registri e del PIC1.

```
    #
    # Check if it is already in kernel mode: the kernel has
    # PID 0. If so, just leave (go to L2).
    #
    mov proc_current, %edx       # Interrupted PID.
    mov $0,           %eax       # Kernel PID.
    cmp %eax, %edx
    je L2
```

Se non è stato interrotto il codice del kernel, viene chiamata la funzione ***proc_scheduler()***, la quale può cambiare i valori delle variabili pubbliche ***proc_stack_segment_selector*** e ***proc_stack_pointer***, provocando così la sostituzione del processo interrotto, quando subito dopo si ripristina la pila a cui queste due variabili fanno riferimento.

84.4.1.2 Particolarità della routine «isr_128»

La pila dei dati al momento dell'interruzione dovuta a una chiamata di sistema, contiene anche le informazioni necessarie a conoscere il tipo di funzione richiesta e gli argomenti di questa, in forma di variabile strutturata, di cui viene trasmesso il puntatore.

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, si recuperano dalla pila le informazioni necessarie a ricostruire la funzione richiesta, salvandole in variabili locali:

```
    #
    # Save some more data, from the system call.
    #
    .equ SYSCALL_NUMBER,      60
    .equ MESSAGE_OFFSET,      64
    .equ MESSAGE_SIZE,        68
    #
    mov %esp, %ebp
    mov SYSCALL_NUMBER(%ebp), %eax
    mov %eax, proc_syscallnr
    mov MESSAGE_OFFSET(%ebp), %eax
    mov %eax, proc_msg_offset
    mov MESSAGE_SIZE(%ebp), %eax
    mov %eax, proc_msg_size
```

A questo punto, in modo simile a quanto avviene per le interruzioni del temporizzatore, si verifica se la chiamata di sistema è avvenuta durante il funzionamento del kernel, cosa che os32 consente. Tuttavia, la chiamata di sistema viene eseguita ugualmente, solo che si salva l'indice della pila nella variabile ***_ksp***; pertanto, è proprio attraverso una prima chiamata di sistema nulla che os32 inizializza la gestione delle interruzioni.

```
    #
    # Check if it is already in kernel mode: the kernel has
    # PID 0.
    #
    mov proc_current, %edx       # Interrupted PID.
    mov $0,           %eax       # Kernel PID.
    cmp %eax, %edx
    jne L3
    #
    mov %esp, _ksp
L3:
```

A questo punto viene eseguito il passaggio alla pila del kernel, indipendentemente dal fatto che serva o meno, quindi viene chiamata la funzione ***sysroutine()***, inserendo nella pila attuale i parametri richiesti e salvati precedentemente all'interno di variabili locali:

```
    push proc_msg_size
    push proc_msg_offset
    push proc_syscallnr
    call sysroutine
    add $4, %esp
    add $4, %esp
    add $4, %esp
```

I passi successivi includono il ripristino della pila precedente, secondo quanto annotato nelle variabili globali ***proc_stack_segment_selector*** e ***proc_stack_pointer***, e lo stato dei registri dalla nuova pila.

Va osservato che la funzione ***sysroutine()*** oltre che prendersi carico di eseguire il compito della chiamata di sistema richiesta, provvede poi a sostituire il processo interrotto, avvalendosi a sua volta della funzione ***proc_scheduler()***.

84.4.2 La tabella dei processi

Listato 94.14.

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

Listato 84.51. Struttura del tipo '**proc_t**', corrispondente agli elementi dell'array ***proc_table[]***.

```
typedef struct {
    pid_t               ppid;          // Parent PID.
    pid_t               pgrp;          // Process group ID.
    uid_t               uid;           // Real user ID
    uid_t               euid;          // Effective user ID.
    uid_t               suid;          // Saved user ID.
    gid_t               gid;           // Real group ID
    gid_t               egid;          // Effective group ID.
    gid_t               sgid;          // Saved group ID.
    dev_t               device_tty;    // Controlling terminal.
    char                path_cwd[PATH_MAX];
                                       // Working directory path.
    inode_t            *inode_cwd;     // Working directory inode.
    int                 umask;         // File creation mask.
    unsigned long int sig_status;      // Active signals.
    unsigned long int sig_ignore;      // Signals to be ignored.
    uintptr_t           sig_handler[MAX_SIGNALS];
                                       // Opt. sig. handlers.
    uintptr_t           sig_handler_wrapper;
                                       // Special wrapper.
    clock_t             usage;         // Clock ticks CPU
                                       // time usage.
    unsigned int        status;
    int                 wakeup_events; // Wake up for something.
    int                 wakeup_signal; // Signal waited.
    unsigned int        wakeup_timer;  // Seconds to wait for.
    inode_t            *wakeup_inode;  // Inode waited.
    addr_t              address_text;
    size_t              domain_text;
    addr_t              address_data;
    size_t              domain_data;
    size_t              domain_stack;  // Included inside the
                                       // data.
    size_t              extra_data;    // Extra data for 'brk()'.
    uint32_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 la cosiddetta «u-area» (*user area*) 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 84.52. 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**'.
`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.

Membro	Contenuto
`gid`	Gruppo reale del processo della voce corrispondente: *group id*. Si tratta del numero del gruppo, secondo la classificazione del file '`/etc/group`', per conto del quale il processo è stato avviato. Tuttavia, i privilegi del processo dipendono dall'identità efficace, definita dal membro '**egid**'.
`egid`	Gruppo efficace del processo della voce corrispondente: *effective group id*. Si tratta del numero del gruppo, secondo la classificazione del file '`/etc/group`', per conto del quale il processo è in funzione; pertanto, il processo ha i privilegi di quel gruppo.
`sgid`	Gruppo salvato: *saved group id*. Si tratta del valore che aveva ***egid*** 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.
`sig_handler[]`	Array di funzioni da eseguire al ricevimento del segnale rispettivo.
`sig_handler_wrapper[]`	Array di funzioni da usare per avvolgere quelle da eseguire al ricevimento di un certo segnale. Si tratta in pratica della funzione dichiarata nel file '`lib/signal/_signal_handler_wrapper.s`', ma è riferita al codice dell'applicazione di origine. Queste funzioni hanno il compito di sistemare la pila dopo l'esecuzione della funzione attivata da un segnale.
`usage`	Tempo di utilizzo della CPU, da parte del processo, espresso in impulsi del temporizzatore, il quale li produce alla frequenza di circa 100 Hz.
`status`	Stato del processo, rappresentabile attraverso una macro-variabile simbolica, definita nel file '`proc.h`'. Per os32, 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/os32.h`'.

Membro	Contenuto
`wakeup_signal`	Ammesso che il processo sia in attesa di un segnale, questo membro esprime il numero del segnale atteso.
`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.
`wakeup_inode`	Ammesso che il processo sia in attesa di poter accedere a un inode, questo membro esprime il puntatore a un inode che deve rendersi disponibile.
`address_text` `domain_text`	Il valore di questi membri descrive la memoria utilizzata dal processo per le istruzioni (il segmento codice). La voce ***domain_text*** rappresenta la dimensione occupata a partire da ***address_text***.
`address_data` `domain_data`	Il valore di questi membri descrive la memoria utilizzata dal processo per i dati; tuttavia l'informazione è utile solo se i dati sono distinti dal segmento codice (gli eseguibili di os32 possono essere compilati in modo da condividere codice e dati nello stesso segmento, oppure in modo da tenerli separati).
`domain_stack`	Dimensione della memoria usata per la pila dei dati, la quale, a seconda del tipo di eseguibile, può collocarsi nel segmento dati, oppure nell'unico segmento che include codice e dati; in ogni caso, si tratta della porzione finale della memoria in questione.
`extra_data`	Dimensione della memoria usata per l'allocazione dinamica della memoria. Questo spazio, ammesso che sia utilizzato, si colloca dopo la pila e può essere modificato con la funzione ***brk()***.
`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.
`fd[]`	Tabella dei descrittori dei file relativi al processo.

L'indice della tabella dei processi corrisponde al numero del processo, ovvero il PID, che infatti non è rappresentato al suo interno. Tuttavia, per accedervi più agevolmente, viene usata la funzione ***proc_reference()***, la quale, fornendo il numero PID desiderato, fornisce il puntatore alla voce della tabella che lo descrive(listato 94.14.6).

84.4.3 Chiamate di sistema

I processi eseguono una chiamata di sistema attraverso la funzione ***sys()***, dichiarata nel file '`lib/sys/os32/sys.s`'(listato 95.21.7). 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 (int 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/os32.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/os32.h`' sono definiti dei tipi derivati, riferiti a variabili strutturate, per ogni tipo di chiamata(listato 95.21). 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_128`**' nel file '`kernel/ibm_i386/isr.s`', la quale estrapola le informazioni salienti dalla pila dei dati e poi le fornisce alla funzione ***sysroutine()***:

```
void sysroutine (uint32_t syscallnr,
                 uint32_t msg_off, uint32_t msg_size);
```

I parametri della funzione ***sysroutine()*** 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. A questo proposito, il secondo parametro cambia nome, in quanto ciò che prima era il puntatore a un'area di memoria, qui va interpretato come lo scostamento rispetto al segmento dati del processo (*segment offset*).

84.4.4 Funzione «proc_init()»

Listato 94.14.3.

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

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

- predisporre la tabella GDT, attraverso la chiamata della funzione ***gdt()***;
- impostare il temporizzatore in modo da fornire impulsi alla frequenza dichiarata nella macro-variabile ***CLOCKS_PER_SEC***, pari a 100 Hz;
- predisporre la tabella IDT, attraverso la chiamata della funzione ***idt()***;
- azzerare la tabella dei processi, inserendovi però i dati relativi al kernel (il processo zero);
- allocare la memoria già utilizzata dal kernel;
- attivare selettivamente le interruzioni hardware desiderate;
- attivare la gestione delle unità PATA;
- innestare il file system principale.

84.4.5 Funzione «sysroutine()»

Listato 94.14.28.

La funzione ***sysroutine()*** viene chiamata esclusivamente dalla routine attivata dalle chiamate di sistema (tale routine è introdotta dal simbolo '**`isr_128`**' nel file '`kernel/ibm_i386/isr.s`') e ha i parametri che si possono vedere dal prototipo:

```
void sysroutine (uint32_t syscallnr,
                 uint32_t msg_off, uint32_t msg_size);
```

Il primo parametro è il numero della chiamata di sistema che ha provocato l'interruzione; gli altri due danno la posizione e la dimensione del messaggio inviato attraverso la chiamata di sistema.

All’inizio della funzione viene dichiarato un puntatore a un’unione di tutti i tipi di messaggio gestibili:

```
    union {
        sysmsg_brk_t        brk;
        sysmsg_chdir_t      chdir;
        sysmsg_chmod_t      chmod;
        …
    } *msg;
```

Viene quindi calcolata la collocazione del messaggio originale, per poi poter assegnare a ***msg*** il puntatore a tale messaggio.

Le chiamate di sistema sono fatte per le applicazioni, ma al kernel è consentito di eseguirne alcune, per motivi particolari. Se però il kernel tenta di eseguire una chiamata differente, si ottiene un messaggio di avvertimento, ma si tenta ugualmente l’esecuzione della richiesta.

Disponendo del puntatore ***msg***, 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***.

Per distinguere il tipo di chiamata si usa una struttura di selezione:

```
    switch (syscallnr)
      {
        case SYS_0:
            break;
        case SYS_BRK:
            msg->brk.ret = s_brk (pid, msg->brk.address);
            sysroutine_error_back (&msg->brk.errno,
                                   &msg->brk.errln,
                                   msg->brk.errfn);
            break;
        case SYS_CHDIR:
            msg->chdir.ret = s_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 vengono modificati. Ciò succede particolarmente con il membro ***ret*** che generalmente rappresenta il valore restituito dalla chiamata di sistema.

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

84.4.6 Funzione «proc_scheduler()»

Listato 94.14.11.

La funzione ***proc_scheduler()*** non prevede parametri e riceve le informazioni che le possono servire attraverso variabili pubbliche: ***_ksp***, ***proc_stack_pointer***, ***proc_stack_segment_selector*** e ***proc_current***. A sua volta, la funzione aggiorna i valori di queste variabili, per mettere in pratica uno scambio di processi.

```
void proc_scheduler (void);
```

La prima cosa che fa la funzione consiste nel verificare che il valore dell’indice della pila del processo interrotto non superi lo spazio disponibile per la pila stessa. Diversamente il processo viene eliminato forzatamente, con una segnalazione adeguata sul terminale attivo. Si ottiene comunque una segnalazione se l’indice si avvicina pericolosamente al limite.

Successivamente la funzione 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   = s_clock ((pid_t) 0);
    proc_table[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 ***proc_current***, ***proc_stack_segment_selector*** e ***proc_stack_pointer***, in modo da ottenere effettivamente lo scambio all’uscita dalla funzione:

```
        else if (proc_table[next].status == PROC_READY)
          {
            if (proc_table[prev].status == PROC_RUNNING)
              {
                proc_table[prev].status = PROC_READY;
              }
            proc_table[prev].sp     = proc_stack_pointer;
            proc_table[next].status = PROC_RUNNING;
            proc_table[next].ret    = 0;
            proc_current            = next;
            proc_stack_segment_selector
                = gdt_pid_to_segment_data (next) * 8;
            proc_stack_pointer      = proc_table[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***.

84.4.7 Programmazione dei segnali

Un processo può ricevere un segnale, a seguito del quale può essere interrotto per compiere una certa azione. La maggior parte dei segnali può essere inibita, in modo tale che ricevendoli il processo non venga a essere disturbato, oppure si può associare loro una funzione, da eseguire al momento del ricevimento del tale segnale. Diversamente, in mancanza di tale associazione, il ricevimento di un segnale comporta un’azione predefinita.

L’associazione di una funzione allo scattare di un segnale si ottiene, nel codice dell’applicazione, con la funzione ***signal()*** (listato 95.17.3), la quale attraverso una chiamata di sistema fornisce al kernel tutti i dati necessari per la programmazione del segnale.

La vera difficoltà sta nell’esecuzione effettiva della funzione, nel momento in cui scatta il segnale previsto per il processo.

```
sighandler_t signal (int sig, sighandler_t handler);
```

La funzione ***signal()*** richiede l'indicazione del numero del segnale da programmare e di un puntatore rappresentato da una funzione che si vuole azionare nel momento in cui scatta il segnale in questione. Il tipo '**`sighandler_t`**' rappresenta il puntatore a una funzione che richiede un parametro di tipo intero, costituito dal numero del segnale ricevuto, e non restituisce alcunché; pertanto, la funzione che si passa come secondo parametro della funzione ***signal()*** deve avere la forma seguente:

```
void handler (int sig);
```

La funzione ***signal()***, a sua volta, esegue finalmente la chiamata di sistema, ma oltre al numero del segnale e al puntatore della funzione da azionare, invia il puntatore di un'altra funzione, denominata ***_sighandler_wrapper()***, il cui scopo è quello di avvolgere la chiamata della funzione da azionare, per sistemare in modo appropriato la pila dei dati (listato 95.17.1). In questa fase della descrizione del problema, va osservato che la funzione ***_sighandler_wrapper()*** si trova nel codice del processo che riceve il segnale.

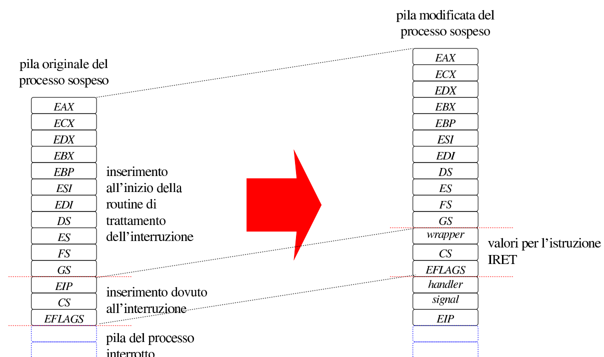
La chiamata di sistema, quando raggiunge il kernel, comporta l'aggiornamento dei dati del processo, annotando sia la funzione da azionare, sia la funzione che deve avvolgerla.

Quando arriva un segnale a un processo che prevede l'azionamento di una funzione, attraverso la funzione ***proc_sch_signals()***, chiamata a sua volta dalla funzione ***proc_scheduler()***, attraverso altri passaggi si arriva alla funzione ***proc_sig_handler()*** (listato 94.14.15).

```
void proc_sig_handler (pid_t pid, int sig);
```

La funzione ***proc_sig_handler()*** ha lo scopo di modificare la pila dei dati del processo ***pid***, in modo da far sì che, nel momento in cui fosse selezionato, prima di riprendere con l'attività sospesa originariamente, esegua la funzione attivata dal segnale ***sig***.

Figura 84.60. Modifica della pila attraverso la funzione ***proc_sig_handler()***.



Come si può vedere nella figura, i valori che servono all'istruzione IRET per concludere l'interruzione, vengono modificati in modo da ripartire iniziando con la funzione che avvolge quella da azionare, ***wrapper***, ovvero quella che dal lato dell'applicazione è chiamata ***_sighandler_wrapper()***. D'altro canto, quando quella funzione viene messa in azione, si trova nella pila dei valori che le servono per poter chiamare a sua volta la funzione da azionare effettivamente.

Il codice di ***_sighandler_wrapper()*** non corrisponde propriamente a una funzione, in quanto ciò che si trova nella pila non è quello che si prevede di solito. Le figure successive mostrano i cambiamenti della pila del processo, prima e dopo l'esecuzione della funzione incaricata di gestire il segnale ricevuto.

Figura 84.61. Dall'avvio di ***_sighandler_wrapper()*** fino alla chiamata della funzione di gestione del segnale ricevuto.

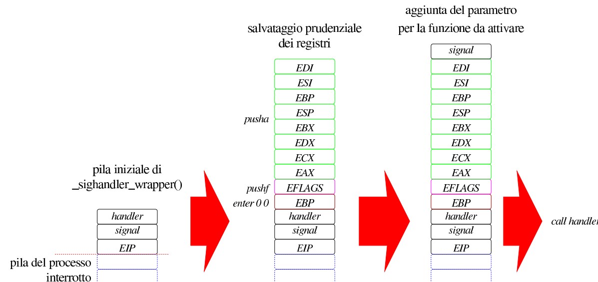
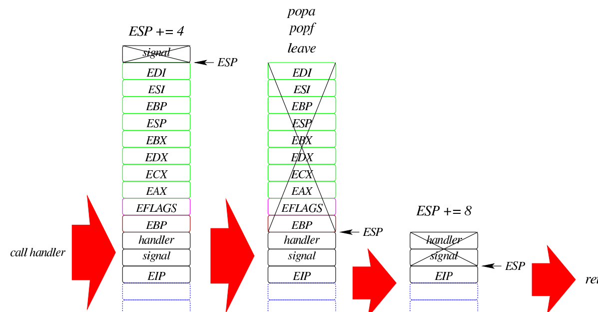


Figura 84.62. Dopo la conclusione della funzione di gestione del segnale ricevuto, fino alla restituzione del controllo al termine di ***_sighandler_wrapper()***.



Lo scopo della funzione ***_sighandler_wrapper()*** è quello di garantire che sia preservato completamente l'ambiente di lavoro del processo nel momento dell'interruzione, perché non è possibile fare affidamento sul rispetto delle convenzioni di chiamata, dato che la funzione da azionare in corrispondenza dell'interruzione, viene iniettata in una posizione arbitraria del codice.

Riquadro 84.63. Programmazione ripetuta dei segnali.

Il sistema tradizionale con cui si programma una funzione in corrispondenza di un segnale, richiede che lo scattare del segnale riporti la gestione di questo allo stato predefinito, perché altrimenti la stessa funzione azionata potrebbe essere interrotta da un segnale che ne aziona un'altra. Se così fosse, la pila dei dati potrebbe riempirsi velocemente, portando il processo al collasso. Di conseguenza, se il programmatore desidera ripristinare una funzione associata a un segnale, lo deve richiedere alla fine della funzione stessa (chiamando ***signal()*** di nuovo), ma in tal caso c'è la possibilità che quel segnale raggiunga il processo nell'intervallo di tempo tra l'azionamento della funzione e il ripristino della programmazione della stessa. Per esempio, ciò significa che se si vuole controllare il segnale ***SIG_TERM*** per impedire che questo porti alla conclusione il processo, anche se la funzione azionata dal segale richiama ***signal()*** per programmarsi nuovamente dopo la chiamata, in modo da non perdere il controllo del segnale, se il processo viene interessato da una raffica si segnali ***SIG_TERM***, prima o poi il processo viene concluso, perché un segnale lo raggiunge quando quella funzione non ha ancora fatto in tempo a chiamare ***signal()***.

84.4.8 Salvataggio e recupero della pila per i «salti non locali»

Il linguaggi C prevede la disponibilità di due funzioni, attraverso le quali è possibile salvare il contesto della pila dei dati per poterne recuperare lo stato in un momento successivo. Tuttavia, tale recupero può avvenire solo se dopo il salvataggio il contenuto della pila precedente rimane valido, in quanto il recupero avviene solo in una situazione in cui la pila sia stata incrementata ulteriormente.

Il salvataggio si ottiene con la funzione ***setjmp()*** e il recupero con ***longjmp()***. L'effetto della chiamata della funzione ***longjmp()*** comporta il riportare il processo alla situazione in cui si trovava dopo l'esecuzione della funzione ***setjmp()***, con la differenza che nel secondo caso, la funzione ***setjmp()*** restituisce un valore differente.

Si tratta di un modo pessimo di programmare, tuttavia fa parte dello standard del linguaggio C.

Generalmente, la realizzazione delle funzioni ***setjmp()*** e ***longjmp()*** avviene nella libreria, senza coinvolgere il kernel in alcun modo. Ma os32 procede diversamente e si avvale invece di chiamate di sistema. Si tratta comunque di una scelta motivata esclusivamente da una più semplice comprensione del codice, facendo rientrare il meccanismo in quello più generale della gestione dei processi.

La funzione ***setjmp()*** è realizzata dal file '`lib/setjmp/setjmp.s`' (listato 95.16.2). La funzione svolge sostanzialmente il compito che si può vedere tradotto in linguaggio C nel codice seguente, se il compilatore gestisse la pila dei dati nella forma più compatta e prevedibile:

```
#include <sys/os32.h>
#include <setjmp.h>
int
setjmp (jmp_buf env)
{
  sysmsg_jmp_t msg;
  msg.env = env;
  msg.ret = 0;
  sys (SYS_SETJMP, &msg, sizeof msg);
  return (msg.ret);
}
```

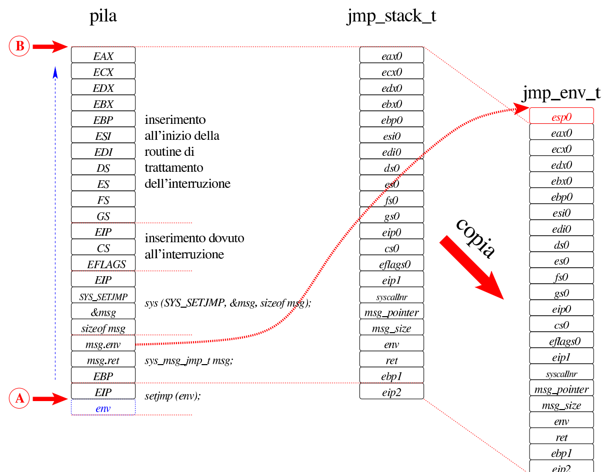
La funzione ***longjmp()*** è realizzata invece in C, nel file '`lib/setjmp/longjmp.c`' (listato 95.16.1), perché non c'è la necessità di conoscere esattamente la struttura della sua pila.

La struttura corrispondente al tipo '**`sysmsg_jmp_t`**' si limita a due campi: un puntatore che deve fare riferimento alla memoria in cui viene salvato il contenuto della pila e il valore che deve restituire ***setjmp()*** quando rivive attraverso la chiamata di ***longjmp()***.

```
typedef struct {
    void  *env;
    int    ret;
} sysmsg_jmp_t;
```

Le due chiamate di sistema raggiungono, rispettivamente, le funzioni ***s_setjmp()*** e ***s_longjmp()*** del kernel (listati 94.8.38 e 94.8.22). La funzione ***s_setjmp()*** salva lo stato della pila, a partire dalla chiamata della funzione ***setjmp()***, mentre ***s_longjmp()*** lo ripristina, rimettendo anche l'indice della pila allo stato che aveva al momento della chiamata di ***setjmp()***.

Figura 84.66. Lo stato della pila durante le varie fasi che riguardano la chiamata di ***setjmp()***, a confronto con i tipi '**`jmp_stack_t`**' e '**`jmp_env_t`**'.



La funzione ***setjmp()*** prevede un argomento di tipo '**`jmp_buf`**' che lo standard prescrive sia come un array:

```
int setjmp (jmp_buf env);
```

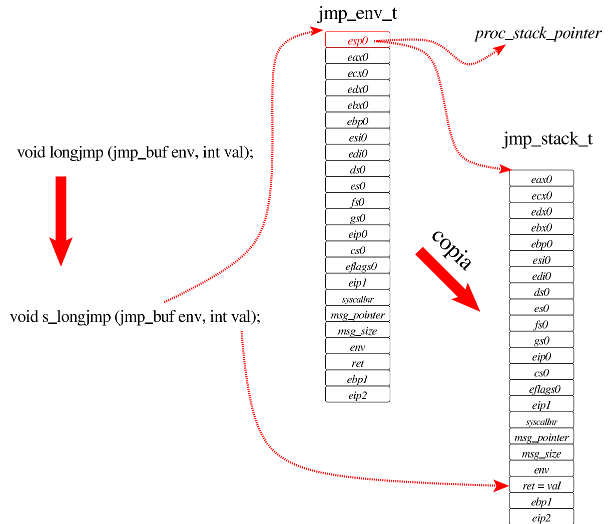
In pratica, l'array serve solo a occupare lo spazio necessario a rap-

presentare il tipo '**`jmp_env_t`**', i cui membri si vedono rappresentati nella figura già apparsa. La funzione ***s_setjmp()*** si occupa si salvare lo stato della pila, dal punto «A» al punto «B» della figura, all'interno di ***env***, secondo la struttura di '**`jmp_env_t`**', mettendo, oltre al contenuto della pila, il valore del suo indice attuale.

La funzione ***longjmp()*** deve portare al ripristino della pila, in una posizione antecedente rispetto a quella attuale.

```
void longjmp (jmp_buf env, int val);
```

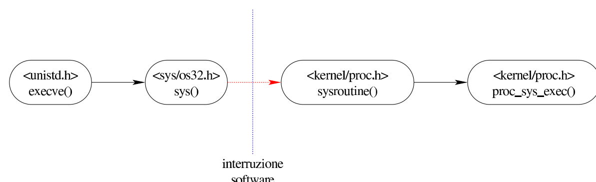
Figura 84.67. La chiamata di ***longjmp()*** ricostruisce la vecchia pila di ***setjmp()***, nella posizione in cui si trovava, ricollocando l'indice della pila e modificando il valore che poi ***setjmp()*** rediviva va a restituire.



84.5 Caricamento ed esecuzione delle applicazioni

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 84.68. Da ***execve()*** a ***proc_sys_exec()***.



84.5.1 Caricamento in memoria

La funzione ***proc_sys_exec()*** (listato 94.14.22) del kernel è quella che svolge il compito di caricare un processo in memoria e di annotarlo nella tabella dei processi.

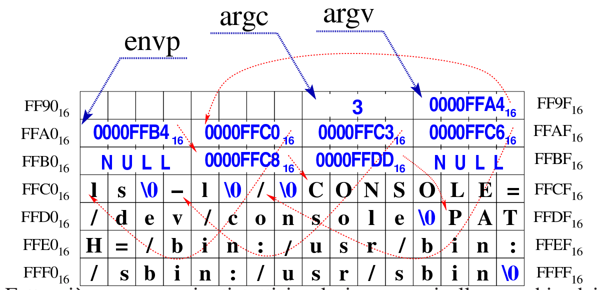
La funzione, dopo aver verificato che si tratti di un file eseguibile valido e che ci siano i permessi per metterlo in funzione, procede all'allocazione della memoria, dividendo se necessario l'area codice da quella dei dati, quindi legge il file e copia opportunamente le componenti di questo nelle aree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

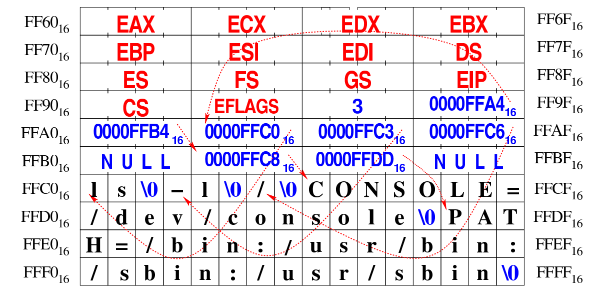
Prima si mettono sul fondo le stringhe delle variabili di ambiente e quelle degli argomenti della chiamata, quindi si aggiungono i puntatori alle stringhe delle variabili di ambiente, ricostruendo così l'array noto convenzionalmente come '**`envp[]`**', continuando con l'aggiunta dei puntatori alle stringhe 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 84.69. 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 84.70. 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.

84.5.2 Il codice iniziale dell'applicativo

«

I programmi iniziano con il codice che si trova nel file '`applic/crt0.mer.s`', oppure '`applic/crt0.sep.s`', a seconda che si compilino in modo da avere codice e dati nello stesso segmento, oppure in segmenti di memoria differenti. Questo file è abbastanza diverso da '`kernel/main/crt0.s`' del kernel; in particolare va osservato che, a differenza del kernel, il codice delle applicazioni viene eseguito in un momento in cui l'indice della pila è già collocato correttamente; inoltre, se la funzione ***main()*** delle applicazioni 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 84.71. Codice iniziale degli applicativi e variabile strutturata di tipo '**`header_t`**'.

```
.section .text
startup:
    jmp startup_code
filler:
    .space (0x0004 - (filler - startup))
magic:
    .quad 0x6F7333326170706C   # os32appl
doffset:
    .int _text_start | _text_end
etext:
    .int _text_end
edata:
    .int _data_end
ebss:
    .int _bss_end
stack_size:
    .int 0x8000
.align 4
startup_code:
...
```

```
typedef struct {
  uint32_t filler0;
  uint64_t magic;
  uint32_t data_offset;
  uint32_t etext;
  uint32_t edata;
  uint32_t ebss;
  uint32_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`' (nel codice si può notare la differenza tra '`crt0.mer.s`' e '`crt0.sep.s`', relativa al valore assegnato alla variabile ***doffset***). 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.

```
    pop %eax              # argc
    pop %ebx              # argv
    pop %ecx              # envp
    mov %ecx, environ     # Variable 'environ' comes
                          # from <unistd.h>.
    push %ecx
    push %ebx
    push %eax
```

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 %ecx
    call _environment_setup
    add  $4, %esp

    mov  $_environment, %eax
    mov  %eax, environ

    pop %eax              # argc
    pop %ebx              # argv[][]
    pop %ecx              # envp[][]
    mov $_environment, %ecx
    push %ecx
    push %ebx
    push %eax
```

Come si vede dall'estratto del file '`applic/crt0.*.s`', si vede l'uso della funzione ***_environment_setup()*** (il registro ECX 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 delle funzioni, per inizializzare la gestione dei flussi di file e delle directory, sempre in forma di flussi, e per predisporre la tabella delle funzioni da eseguire alla conclusione del processo.

```
    call _stdio_stream_setup
    call _dirent_directory_stream_setup
    call _atexit_setup

    call main

    mov  %eax, exit_value
    …
.align 4
.section .data
exit_value:
    .int 0x00000000
.align 4
.section .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()***, contenuta nel file '`lib/dirent/DIR.c`', compie un lavoro analogo, limitandosi però a inizializzare un array di flussi di directory; la funzione ***_atexit_setup()***, contenuta nel file '`lib/stdlib/atexit.c`' azzera l'array ***_atexit_table[]***, destinato a contenere l'elenco di funzioni da eseguire alla conclusione del processo.

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:
    pushl $2            # Size of message.
    pushl $exit_value   # Pointer to the message.
    pushl $6            # SYS_EXIT
    call sys
    add  $4, %esp
    add  $4, %esp
    add  $4, %esp
    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_value*`);`**'.

84.6 Gestione della memoria

Dal punto di vista del kernel di os32, l'allocazione della memoria riguarda la collocazione dei processi elaborativi nella stessa. Per semplicità 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 4096 byte, pertanto, sapendo che la memoria può arrivare solo fino a 4 Gibyte, si gestiscono al massimo 1048576 blocchi.

Per la scansione della mappa si utilizzano interi da 32 bit, pertanto tutta la mappa si riduce a 32768 di questi interi, ovvero 128 Kibyte. Nell'ambito di ogni intero da 32 bit, il bit più significativo rappresenta il primo blocco di memoria di sua competenza. Per esempio, per indicare che si stanno utilizzando i primi 28672 byte, pari ai primi 7 blocchi di memoria, si rappresenta la mappa della memoria come «FE0000000...».

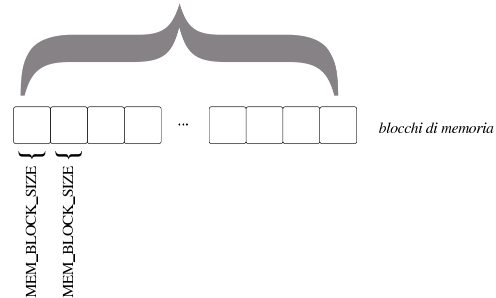
Il fatto che la mappa della memoria vada scandito a ranghi di 32 bit va tenuto in considerazione, perché se invece si andasse con ranghi differenti, si incapperebbe nel problema dell'inversione dei byte.

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

Listato 94.10 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 il tipo derivato '**`addr_t`**', corrispondente a un indirizzo di memoria reale.

Figura 84.76. 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 32 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.

Tabella 84.77. Funzioni per la gestione della mappa della memoria, dichiarate nel file di intestazione '`kernel/memory.h`' e realizzate nella directory '`kernel/memory/`'.

Funzione	Descrizione
`uint32_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 indicato, per la quantità di byte richiesta. L'allocazione ha termine anticipatamente se si incontra un blocco già utilizzato. La funzione restituisce la dimensione allocata effettivamente.
`void mb_free (addr_t` ***address***`, size_t` ***size***`);`	Libera la memoria a partire dall'indirizzo indicato, per la quantità di byte richiesta. 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_reduce (addr_t` ***address***`, size_t` ***new***`, size_t` ***previous***`);`	Riduce un'area di memoria già utilizzata. Restituisce zero se l'operazione si conclude con successo, oppure -1 in caso contrario, aggiornando la variabile ***errno*** di conseguenza.
`void mb_clean (addr_t` ***address***`, size_t` ***size***`);`	Azzera l'area di memoria specificata.

Funzione	Descrizione
`addr_t mb_alloc_size (size_t size);`	Alloca un'area di memoria della dimensione richiesta, restituendone l'indirizzo. La funzione conclude con successo il proprio lavoro se il valore restituito è diverso da zero; se invece l'indirizzo ottenuto è pari a zero si è verificato un errore che può essere verificato analizzando il contenuto della variabile ***errno***.
`void mb_size (size_t size);`	Questa funzione, usata una sola volta all'interno di ***kmain()***, serve a definire la dimensione massima della memoria disponibile in blocchi. In pratica, le si fornisce la dimensione effettiva della memoria che viene così divisa per la dimensione del blocco, ignorando il resto. Questa informazione viene conservata nella variabile ***mb_max***.
`void mb_print (void);`	Funzione diagnostica che visualizza gli intervalli di memoria utilizzati, esprimendoli però in blocchi.

84.6.2 Scansione della mappa di memoria

Listato 94.10 e successivi.

La mappa della memoria si rappresenta (a sua volta in memoria), con un array di interi a 32 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 32.

Il primo elemento di questo array, ovvero ***mb_table[0]***, individua i primi 32 blocchi di memoria, dove il bit più significativo si riferisce precisamente al primo blocco. Per esempio, se ***mb_table[0]*** contiene il valore $F8000000_{16}$, ovvero 1111100000000000_2, significa che i primi cinque blocchi di memoria sono occupati, mentre i blocchi dal sesto al trentaduesimo 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 32, 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 considerare, 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 2/32 dà zero, come risultato intero), mentre la maschera necessaria a trovare il bit corrispondente è $00100000000000000000000000000000_2$, 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.

84.7 Dispositivi

La gestione dei dispositivi fisici, da parte di os32, è limitata ed essenziale. Tutte le operazioni di lettura e scrittura di dispositivi, passano attraverso la gestione comune della funzione ***dev_io()***.

Nel file '`lib/sys/os32.h`' (listato 95.21), disponibile sia al kernel, sia alle applicazioni, sono elencate le macro-variabili che descrivono tutti i dispositivi previsti in forma numerica. Queste macro-variabili hanno nomi prefissati dalla sigla ***DEV_***.... Per esempio, ***DEV_DM_MAJOR*** corrisponde al numero primario (*major*) per le unità di memorizzazione di massa, ***DEV_DM00*** corrisponde al numero primario e secondario (*major* e *minor*), in un valore unico, della prima unità di memorizzazione di massa complessiva, mentre ***DEV_DM01*** corrisponde alla prima partizione della stessa.

84.7.1 File «kernel/dev.h» e «kernel/dev/...»

Listati 94.3 e successivi.

Il file '`kernel/dev.h`' incorpora il file '`lib/sys/os32/os32.h`', per acquisire le macro-variabili della 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.

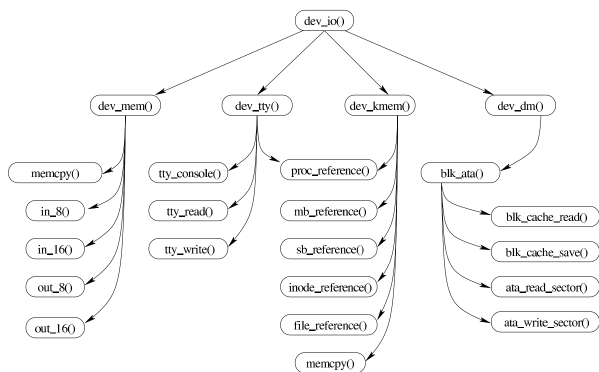
Va osservato il caso particolare dei dispositivi ***DEV_KMEM_***.... In un sistema operativo Unix comune, attraverso ciò che fa capo al file di dispositivo '`/dev/kmem`', si ha la possibilità di accedere all'immagine in memoria del kernel, lasciando a un programma con privilegi adeguati la facoltà di interpretare i simboli che consentono di

individuare i dati esistenti. Nel caso di os32, 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.

os32 gestisce virtualmente tutti i dispositivi come se fossero a caratteri. Tuttavia, nel caso delle unità di memorizzazione di massa il flusso di caratteri, in lettura o in scrittura, viene scomposto in blocchi, sfruttando anche una memoria (*cache*) per questi. Pertanto, la funzione ***dev_dm()*** si avvale di ***blk_ata()***.

Figura 84.80. Interdipendenza tra la funzione ***dev_io()*** e le altre. I collegamenti con le funzioni ***major()*** e ***minor()*** sono omesse.



84.7.2 File «kernel/blk.h» e «kernel/blk/...»

Listati 94.2 e successivi.

I file contenuti nella directory '`kernel/blk/`' riguardano specificatamente la gestione della memoria *cache* per i blocchi di dati usati più di frequente, relativamente ai dispositivi di memorizzazione. In pratica, tale gestione riguarda esclusivamente le unità PATA.

La tabella ***blk_table()*** è composta da elementi '**`blk_cache_t`**', ognuno dei quali rappresenta un blocco singolo, con l'indicazione del dispositivo (dell'unità intera e non di una singola partizione) e del numero di blocco a cui si riferisce, assieme a un numero che ne rappresenta l'«età».

Inizialmente, la funzione ***blk_cache_init()***, usata una volta sola all'interno di ***kmain()***, si azzerano le informazioni sul numero di dispositivo e sul numero del blocco di ogni elemento della tabella, quindi si assegna l'età attraverso un numero progressivo, da 0 a ***BLK_CACHE_MAX_AGE***. Il numero più basso rappresenta l'ultimo blocco letto o modificato, mentre quello più alto riguarda il blocco che da più tempo non è stato utilizzato.

Quando la funzione ***blk_ata()*** deve leggere un blocco da un'unità PATA, prima, attraverso la funzione ***blk_cache_read()***, controlla all'interno della tabella ***blk_table()*** esiste già una copia del blocco; questo viene trovato, la funzione ***blk_cache_read()*** ne azzera l'età, incrementando conseguentemente l'età dei blocchi che avevano prima un valore inferiore al suo. Se il blocco viene trovato nella tabella, la funzione non interpella l'hardware PATA e conclude il suo lavoro, altrimenti provvede alla lettura necessaria e al suo salvataggio nella tabella dei blocchi, con l'aiuto di ***blk_cache_save()***, la quale aggiorna il blocco se questo era già presente nella tabella, oppure rimpiazza il blocco di età maggiore, aggiornando di conseguenza l'età, come nel caso della lettura.

Quando la funzione ***blk_ata()*** deve scrivere un blocco, la scrittura hardware avviene in ogni caso, seguita dal salvataggio nella tabella dei blocchi.

In pratica, la memoria *cache* viene usata solo per le letture, pertanto tutte le scritture sono sincrone.

84.7.3 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.

84.7.4 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 84.82. Classificazione dei dispositivi di os32.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_MEM`	r/w	diretto	Permette l'accesso alla memoria, in modo indiscriminato; tuttavia, solo al kernel è permessa la scrittura.
`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_DM`***mn***	r/w	diretto	Rappresenta la partizione ***n*** dell'unità di memorizzazione ***m***. La prima unità PATA disponibile ottiene il dispositivo ***DEV_DM00***, la seconda il numero ***DEV_DM10***, ecc.
`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.
`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_KMEM_ARP`	r	diretto	Rappresenta la tabella ARP (per la trasformazione degli indirizzi IPv4 in indirizzi Ethernet). L'indirizzo di accesso serve a individuare la voce; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.
`DEV_KMEM_NET`	r	diretto	Rappresenta la tabella delle interfacce di rete. L'indirizzo di accesso serve a individuare la voce della tabella; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.
`DEV_KMEM_ROUTE`	r	diretto	Rappresenta la tabella degli instradamenti IPv4. L'indirizzo di accesso serve a individuare la voce della tabella; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.
`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.

84.7.5 Gestione del terminale

Listato 94.4.42 e successivi.

Il terminale offre solo la funzionalità elementare della modalità canonica, dove è possibile scrivere o leggere sequenzialmente. Ci sono al massimo quattro terminali virtuali, selezionabili attraverso le combinazioni di tasti [*Ctrl q*], [*Ctrl r*], [*Ctrl s*] e [*Ctrl t*] e non è possibile controllare i colori o la posizione del testo che si va a esporre; in pratica si opera come su una telescrivente. Le funzioni di livello più basso, relative al terminale hanno nomi che iniziano per '**`tty_...()`**'.

Per la gestione dei quattro terminali virtuali, si utilizza una tabella, in cui ogni voce rappresenta lo stato del terminale virtuale che rappresenta. La tabella è costituita dall'array ***tty_table[]*** che contiene ***TTY_TOTALS*** elementi. L'array è dichiarato nel file '`kernel/driver/tty_public.c`', mentre la macro-variabile

TTY_TOTALS appare nel file 'kernel/driver/tty.h'. Gli elementi di ***tty_table[]*** sono di tipo '**tty_t**':

```
typedef struct {
    dev_t          device;
    pid_t          pgrp;            // Process group.
    struct termios attr;            // termios attributes.
    unsigned char  status;          // 0 = edit,
                                    // 1 = end edit.
    char           line[MAX_CANON]; // Canonical input line.
    int            lpr;             // Input line position
                                    // read.
    int            lpw;             // Input line position
                                    // write.
} tty_t;
```

Il membro ***attr*** della voce di un terminale è una variabile strutturata di tipo '**struct termios**', come previsto nel file 'termios.h' della libreria standard.

L'input del terminale, proveniente dalla tastiera, viene depositato dalla funzione ***proc_sch_terminals()*** all'interno del membro ***line[]***, annotando in ***lpw*** l'indice di scrittura. Quando si legge dal terminale, si ottiene un carattere alla volta da ***line[]***, con l'ausilio dell'indice ***lpr***. Quando il terminale virtuale riceve input dalla tastiera, è nello stato definito dalla macro-variabile ***TTY_INPUT_LINE_EDITING***, mentre quando l'inserimento risulta concluso, per esempio perché è stato premuto il tasto [*Invio*], lo stato è quello di ***TTY_INPUT_LINE_CLOSED*** ed è possibile procedere con la lettura del contenuto di ***line[]***: quando la lettura termina perché l'indice ***lpr*** ha raggiunto ***lpw***, gli indici vengono azzerati e lo stato ritorna quello di inserimento. Quando un processo tenta di leggere dal terminale, mentre questo è in fase di inserimento, non ancora concluso, viene sospeso e rimane così fino alla conclusione dell'inserimento stesso.

Figura 84.84. A sinistra le fasi dell'inserimento di una riga da tastiera; a destra le fasi della lettura attraverso la funzione ***tty_read()***.

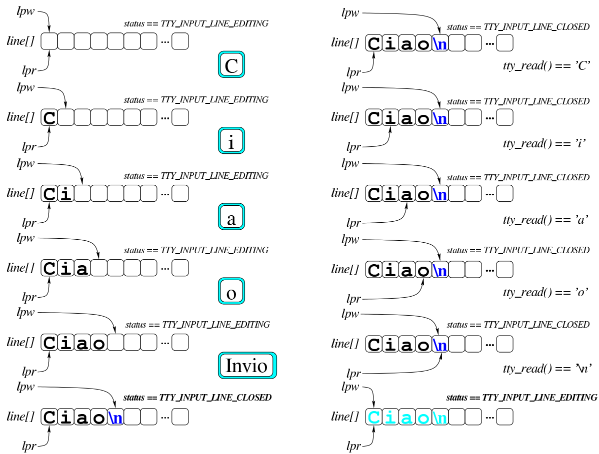


Tabella 84.85. Funzioni per l'accesso al terminale, dichiarate nel file di intestazione 'kernel/driver/tty.h' e descritte nei file contenuti nella directory 'kernel/driver/tty/'.

Funzione	Descrizione
`dev_t tty_console (dev_t device);`	Seleziona un terminale virtuale, rendendolo attivo, specificandone il numero del dispositivo. La funzione restituisce il dispositivo attivo in precedenza e se le viene fornito solo il valore zero, il terminale virtuale non cambia, ma si ottiene comunque di conoscere qual è quello attuale.

Funzione	Descrizione
`void tty_init (void);`	Inizializza la gestione dei terminali virtuali, popolando anche la tabella ***tty_table[]*** con i valori predefiniti. Questa funzione viene usata una volta sola all'interno di ***kmain()***.
`int tty_read (dev_t device);`	Legge un carattere dal terminale virtuale specificato attraverso il numero di dispositivo. La lettura avviene solo se l'input da tastiera risulta concluso, altrimenti la funzione restituisce il valore −1.
`tty_t *tty_reference (dev_t device);`	Restituisce il puntatore alla voce della tabella ***tty_table[]*** contenente le informazioni sul terminale virtuale indicato attraverso il numero di dispositivo. Se il dispositivo indicato non è valido, si ottiene il puntatore nullo; se viene richiesto il dispositivo indefinito, si ottiene il puntatore all'inizio della tabella.
`void tty_write (dev_t device, int c);`	Scrive un carattere sullo schermo del terminale specificato.

84.7.5.1 Gestione della tastiera

«

Listato 94.4.15 e successivi.

Per la gestione della tastiera, nel file 'kernel/driver/kbd_public.c' viene dichiarata una variabile strutturata, di tipo '**kbd_t**', contenente le informazioni sullo stato della stessa e sulla mappa di trasformazione da applicare. A differenza della gestione complessiva dei terminali, in cui ogni terminale virtuale ha un proprio insieme di dati, per la tastiera questo è unico.

Listato 84.86. Definizione del tipo '**kbd_t**', contenuto nel file 'kernel/driver/kbd.h'.

```
typedef struct {
    bool          shift;
    bool          shift_lock;
    bool          ctrl;
    bool          alt;
    bool          echo;
    unsigned char key;
    unsigned char map1[128];
    unsigned char map2[128];
} kbd_t;
```

Nella variabile ***kbd***, come si intuisce dai nomi dei suoi membri, viene annotato lo stato di pressione dei tasti delle maiuscole, dei tasti [*Ctrl*] e [*Alt*], per poter recepire eventuali combinazioni di tasti; inoltre, il membro ***echo***, se attivo, indica la richiesta di vedere sullo schermo ciò che si digita.

Dalla tastiera viene recepito un solo tasto alla volta: se questo si traduce in un carattere, stampabile o meno che sia, questo viene depositato nel membro ***key***, da dove la funzione ***proc_sch_terminals()*** deve provvedere a prelevarlo (per trasferirlo nel membro ***line[]*** della voce che descrive il terminale virtuale attivo), azzerando nuovamente ***key***. Fino a quando il membro ***key*** ha un valore diverso da zero, non è possibile recepire altro dalla tastiera.

Dalla tastiera è possibile ottenere solo i caratteri ASCII; in particolare, quelli non stampabili si ottengono per combinazione con il tasto [*Ctrl*], secondo la convenzione tradizionale. Non sono previste altre funzionalità.

Tabella 84.87. Funzioni per la gestione della tastiera, dichiarate nel file di intestazione 'kernel/driver/kbd.h' e descritte nei file contenuti nella directory 'kernel/driver/kbd/'.

Funzione	Descrizione
`void kbd_isr (void);`	Questa funzione è chiamata dalla routine di gestione delle interruzioni da tastiera, contenuta nel file 'kernel/ibm_i386/isr.s'. La funzione legge un carattere dalla porta di I/O 60_{16}, quindi lo interpreta e aggiorna il contenuto della variabile strutturata ***kbd*** di conseguenza.
`void kbd_load (void);`	Questa funzione è chiamata una sola volta da ***kmain()***, per associare la mappa della tastiera ai codici prodotti dalla stessa. Attualmente questa funzione produce esclusivamente l'associazione necessaria per una tastiera italiana.

La funzione ***proc_scheduler()***, il cui scopo principale è quello di alternare i processi in esecuzione, tra le altre cose, avvia ogni volta la funzione ***proc_scheduler_terminals()***. La funzione ***proc_scheduler_terminals()*** verifica se nella variabile ***kbd.key*** è disponibile un valore diverso da zero e, se c'è, lo acquisisce per conto del terminale attivo. Prima di tutto verifica se si tratta di una combinazione di tasti che richiede lo scambio a un altro terminale virtuale; poi controlla se si tratta di un codice di interruzione (come quello provocato da [*Ctrl c*]) e, se la configurazione del terminale attivo lo permette, conclude il processo più interno appartenente al gruppo che risulta connesso al terminale stesso; alla fine, dopo altre ipotesi particolari, se si tratta di un carattere «normale» e il terminale si trova in fase di inserimento (***TTY_INPUT_LINE_EDITING***), questo viene depositato nell'array ***line[]***, con il conseguente aggiornamento dell'indice di scrittura al suo interno; ricevendo invece un codice che rappresenta la conclusione dell'inserimento, si rimette il terminale nello stato di conclusione dell'inserimento (***TTY_INPUT_LINE_CLOSED***).

84.7.5.2 Gestione dello schermo

«

Listato 94.4.30 e successivi.

Lo schermo di os32 viene gestito secondo quanto prescrive l'hardware VGA (come descritto nella sezione 83.3), per cui ciò che si vuole fare apparire deve essere scritto in memoria a partire dall'indirizzo $B8000_{16}$, usando per ogni carattere 16 bit (8 bit di questo gruppo servono per gli attributi).

Dal momento che si gestiscono dei terminali virtuali, per ognuno di questi occorre tenere una copia dell'immagine dello schermo, così, quando si seleziona un terminale differente, la copia di quel terminale viene usata per sovrascrivere l'area di memoria che rappresenta lo schermo. Per la gestione degli schermi virtuali si usa una tabella, denominata ***screen_table[]***, composta da voci di tipo '**screen_t**'.

Listato 84.88. Definizione del tipo '**screen_t**', contenuto nel file 'kernel/driver/screen.h'.

```
typedef struct {
    uint16_t  cell[SCREEN_CELLS];
    int       position;
} screen_t;
```

All'interno della struttura rappresentata dal tipo '**screen_t**', si vede un array che riproduce la rappresentazione in memoria dello stesso, da copiare a partire dall'indirizzo $B8000_{16}$, quando lo schermo virtuale diventa quello attivo; inoltre si vede il membro ***position***, usato per ricordare la posizione in cui si trova il cursore.

Tabella 84.89. Funzioni per la gestione dello schermo, dichiarate nel file di intestazione 'kernel/driver/screen.h' e descritte nei file contenuti nella directory 'kernel/driver/screen/'.

Funzione	Descrizione
`int screen_clear (screen_t *`***screen***`);`	Ripulisce il contenuto dello schermo selezionato, riposizionando il cursore all'inizio.
`screen_t *screen_current (void);`	Restituisce il puntatore alla voce della tabella ***screen_table[]*** che descrive lo schermo virtuale attivo.
`void screen_init (void);`	Inizializza la gestione degli schermi virtuali, ripulendoli e collocando il cursore all'inizio. Questa funzione viene usata da ***tty_init()***.
`int screen_newline (screen_t *`***screen***`);`	Produce sullo schermo virtuale selezionato un avanzamento alla riga successiva. Ciò può comportare semplicemente il riposizionamento del cursore, oppure lo scorrimento in avanti del contenuto, quando il cursore si trova già sull'ultima riga visualizzabile.
`int screen_number (screen_t *`***screen***`);`	Restituisce il numero dello schermo corrispondente al puntatore fornito, purché questo sia valido. È in pratica l'opposto della funzione ***screen_pointer()***.
`screen_t *screen_pointer (int `***scrn***`);`	Restituisce il puntatore alla voce della tabella ***screen_table[]*** che descrive lo schermo virtuale indicato per numero. È in pratica l'opposto della funzione ***screen_number()***.
`int screen_putc (screen_t *`***screen***`, int `***c***`);`	Colloca sullo schermo virtuale individuato dal puntatore che costituisce il primo parametro, il carattere richiesto come secondo. Se il carattere in questione è *<CR>* o *<LF>*, si produce un avanzamento alla riga successiva, mentre con un carattere *<BS>* si produce un arretramento del cursore.
`uint16_t screen_cell (`***c***`, `***attributo***`);`	Si tratta di una macroistruzione che produce il valore corretto per una cella dello schermo VGA, contenente sia l'informazione sul carattere, sia quella dell'attributo associato.
`int screen_scroll (screen_t *`***screen***`);`	Fa scorrere in avanti lo schermo, di una riga, ricollocando di conseguenza il cursore.
`int screen_select (screen_t *`***screen***`);`	Seleziona lo schermo indicato come schermo attivo, facendone apparire il contenuto sullo schermo VGA reale.

Funzione	Descrizione
`void screen_update (screen_t *screen);`	Aggiorna la memoria VGA sulla base della copia che rappresenta lo schermo virtuale attivo. L'aggiornamento implica anche la collocazione del cursore visibile in corrispondenza della posizione attuale.

84.7.5.3 Configurazione del terminale

Lo standard dei sistemi Unix prescrive che per ogni terminale gestito sia prevista una variabile strutturata, di tipo ***struct termios***, allo scopo di contenere la configurazione dello stesso. os32 gestisce i terminali virtuali soltanto in modalità «canonica», ovvero come se si trattasse di telescriventi, anche se munite di video invece che di carta, pertanto utilizza solo un sottoinsieme delle opzioni previste.

```
typedef uint16_t      tcflag_t;
typedef unsigned char cc_t;
...
struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t     c_cc[NCCS];
};
```

Il membro ***c_cc[]*** è un array di caratteri di controllo, a cui viene attribuita una definizione.

Tabella 84.91. Caratteri di controllo riconosciuti da os32, secondo le definizioni del file '`termios.h`'.

Definizione	Corrispondenza	Descrizione
VEOF	04_{16} *<EOT>*	Carattere di fine file.
VERASE	08_{16} *<BS>*	Carattere di cancellazione.
VINTR	03_{16} *<ETX>*	Carattere di interruzione.
VQUIT	$1C_{16}$ *<FS>*	Carattere di abbandono.

Il membro ***c_iflag*** serve a contenere opzioni sull'inserimento, ovvero sul controllo della digitazione.

Tabella 84.92. Opzioni del membro ***c_iflag*** riconosciute da os32.

Opzione	Descrizione
BRKINT	Se questa opzione è attiva e, nel contempo, non è attiva ***IGNBRK***, si intendono recepire i codici di interruzione ***VINTR***. Se l'opzione ***ISIG*** del membro ***c_lflag*** è attiva, il processo più interno del gruppo a cui appartiene il terminale viene concluso; in ogni caso, viene annullato il contenuto della riga di inserimento in corso.
ICRNL	Se si riceve il carattere *<CR>*, questo viene convertito in *<NL>*.
IGNBRK	Se questa opzione è attiva, fa sì che il carattere definito come ***VINTR*** sia ignorato.
IGNCR	Se si riceve il carattere *<CR>*, questo viene ignorato semplicemente.
INLCR	Se si riceve il carattere *<NL>*, questo viene convertito in *<CR>*.

Il membro ***c_lflag*** serve a contenere delle opzioni definite come «locali», le quali si occupano in pratica di controllare la visualizzazione della digitazione introdotta e di decidere se l'interruzione ricevuta da tastiera debba produrre l'invio di un segnale di interruzione al processo con cui si sta interagendo. Gli altri due membri della struttura non vengono utilizzati da os32.

Tabella 84.93. Opzioni del membro ***c_lflag*** riconosciute da os32.

Opzione	Descrizione
ECHO	Abilita la visualizzazione sullo schermo del testo inserito da tastiera.
ECHOE	Ammesso che sia attiva l'opzione ***ECHO***, questa abilita il recepimento del carattere definito come ***VERASE*** per cancellare l'ultimo carattere inserito, indietreggiando di una posizione.
ECHONL	Indipendentemente dall'opzione ***ECHO***, questa abilita il recepimento del carattere *<NL>* per fare avanzare il cursore alla riga successiva, con l'eventuale scorrimento in avanti se si trova già sull'ultima.
ISIG	Ammesso che sia recepito e accettato un codice di interruzione, definito come ***VINTR***, con questa opzione si ottiene l'invio di un segnale di interruzione al processo più interno del gruppo collegato al terminale (il processo più interno dovrebbe corrispondere a quello in primo piano al momento della digitazione).

84.7.6 Gestione delle unità di memorizzazione in generale

Listato 94.4 e successivi.

Le unità di memorizzazione vengono viste da os32 attraverso un gruppo di dispositivi astratti, definiti come ***DEV_DM****, dove la prima unità PATA disponibile ottiene il numero ***DEV_DM00***, la seconda ***DEV_DM10***,...

Il numero del dispositivo che rappresenta queste unità è composto in modo solito per ciò che riguarda la distinzione tra numero primario e numero secondario, ma il numero secondario si scompone ulteriormente in due parti: l'unità intera e la partizione. Per esempio, il numero 0810_{16} individua la seconda unità di memorizzazione per intero, mentre 0811_{16} rappresenta la prima partizione della seconda unità.

Le unità di memorizzazione riconosciute dal sistema sono raccolte in una tabella, denominata ***dm_table[]***. Ogni elemento di questa tabella contiene l'informazione sul tipo di unità, un puntatore per raggiungere un'altra tabella con le informazioni specifiche sull'unità, in base al tipo di questa (attualmente l'unica tabella in questione può essere quella delle unità PATA), le informazioni sulle partizioni esistenti (ma solo quelle primarie).

Inizialmente, all'interno di ***proc_init()***, viene avviata la funzione ***dm_init()***, la quale a sua volta scandisce le unità PATA attraverso ***ata_init()*** e ne raccoglie le informazioni nella propria tabella ***dm_table()***.

84.7.7 Gestione delle unità PATA

Listato 94.4.3 e successivi.

La gestione delle unità PATA di os32 si limita alla modalità PIO (*programmed input-output*), con accesso LBA28, senza nemmeno considerare le partizioni. La spiegazione sul come avvenga la gestione di un'unità PATA, secondo le stesse modalità usate da os32 è disponibile nella sezione 83.9.

Per la gestione delle unità PATA, os32 utilizza una tabella, denominata ***ata_table[]***, composta da voci di tipo ***ata_t***, ognuna delle quali contiene lo stato di un'unità. L'indice della tabella corrisponde al numero dell'unità, ovvero al parametro ***drive*** di varie funzioni. La tabella è dichiarata formalmente nel file '`kernel/driver/ata/ata_public.c`', mentre il tipo derivato '**`ata_t`**' è descritto nel file '`kernel/driver/ata.h`'. Per comodità, si trova anche il tipo '**`ata_sector_t`**', usato per descrivere lo spazio di memoria usato per collocare la copia di un settore di dati di un'unità PATA.

Tabella 84.94. Funzioni per la gestione delle unità PATA, dichiarate nel file di intestazione '`kernel/driver/ata.h`' e descritte nei file contenuti nella directory '`kernel/driver/ata/`'. Le funzioni sono raggruppate in insiemi logici.

Funzione	Descrizione
`void ata_init (void);`	Inizializza la gestione delle unità PATA, predisponendo i contenuti della tabella ***ata_table[]***, verificando la presenza delle unità. Questa funzione viene usata una volta sola, nella funzione ***proc_init()***.
`void ata_reset (int` ***drive***`);`	Azzera lo stato di funzionamento dell'unità PATA specificata.
`int ata_valid (int` ***drive***`);`	Verifica se l'unità richiesta è presente effettivamente. In caso di successo restituisce il valore zero, altrimenti si ottiene -1.

Funzione	Descrizione
`int ata_cmd_identify_device (int` ***drive***`, void *`***buffer***`);`	Richiede all'unità specificata le informazioni sulla sua identificazione. Se l'unità è presente, in corrispondenza del puntatore fornito si ottengono le informazioni nello spazio di un settore (***ATA_SECTOR_SIZE***); l'analisi successiva di questi dati può dare maggiori informazioni sull'unità.
`int ata_cmd_read_sectors (int` ***drive***`, unsigned int` ***sector***`, unsigned char` ***count***`, void *`***buffer***`);`	Legge dall'unità ***drive***, a partire dal settore ***sector***, una quantità pari a ***count*** settori, mettendo il risultato a partire dall'indirizzo di memoria ***buffer***. Se ***count*** fosse pari a zero, si intenderebbero 256 settori. Se l'operazione fallisce, restituisce un valore negativo.
`int ata_cmd_write_sectors (int` ***drive***`, unsigned int` ***sector***`, unsigned char` ***count***`, void *`***buffer***`);`	Scrive nell'unità ***drive***, a partire dal settore ***sector***, una quantità pari a ***count*** settori, leggendoli a partire dall'indirizzo di memoria ***buffer***. Se ***count*** fosse pari a zero, si intenderebbero 256 settori. Se l'operazione fallisce, restituisce un valore negativo.
`int ata_device (int` ***drive***`, unsigned int` ***sector***`);`	Imposta il registro ***device*** dell'unità PATA specificata, con l'indicazione di un numero di settore.

Funzione	Descrizione
`int ata_rdy (int` ***drive***`, clock_t` ***timeout***`);`	Attende che l'unità ***drive*** sia pronta, purché ciò avvenga entro il tempo ***timeout***. Se l'operazione ha successo, la funzione restituisce zero, altrimenti dà un valore negativo.
`int ata_drq (int` ***drive***`, clock_t` ***timeout***`);`	Attende che l'unità ***drive*** sia pronta a ricevere dati, purché ciò avvenga entro il tempo ***timeout***. Se l'operazione ha successo, la funzione restituisce zero, altrimenti dà un valore negativo.

Funzione	Descrizione
`int ata_lba28 (int` ***drive***`, unsigned int` ***sector***`, unsigned char` ***count***`);`	Invia all'unità ***drive*** la prima parte di un comando, in cui sono contenute le coordinate LBA28.

Funzione	Descrizione
`int ata_read_sector (int` ***drive***`, unsigned int` ***sector***`, void *`***buffer***`);`	È una macroistruzione che legge dall'unità ***drive***, il settore ***sector***, mettendo il risultato a partire dall'indirizzo di memoria ***buffer***. La macroistruzione si avvale praticamente della funzione ***ata_cmd_read_sectors()***, per leggere un solo settore.
`int ata_write_sector (int` ***drive***`, unsigned int` ***sector***`, void *`***buffer***`);`	È una macroistruzione che scrive nell'unità ***drive***, il settore ***sector***, traendo i dati dall'indirizzo di memoria ***buffer***. La macroistruzione si avvale praticamente della funzione ***ata_cmd_write_sectors()***, per scrivere un solo settore.

84.8 Gestione del file system

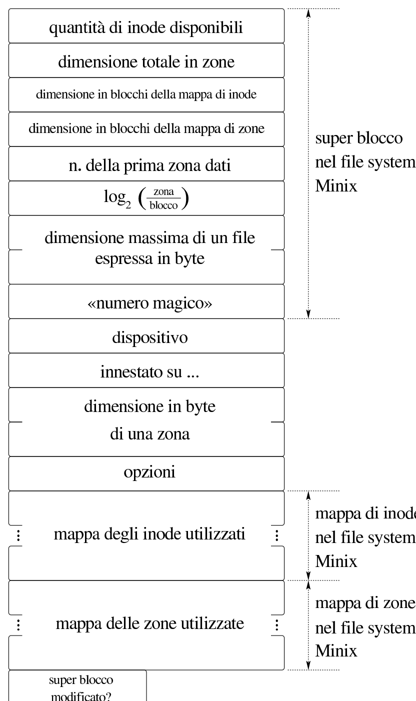
La gestione del file system è suddivisa in diversi file contenuti nella directory '`kernel/fs/`', facenti capo al file di intestazione '`kernel/fs.h`'.

Listato 94.5 e successivi.

84.8.1 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 84.95. Struttura del tipo '**sb_t**', corrispondente agli elementi dell'array ***sb_table[]***.



Listato 84.96. 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;
};
```

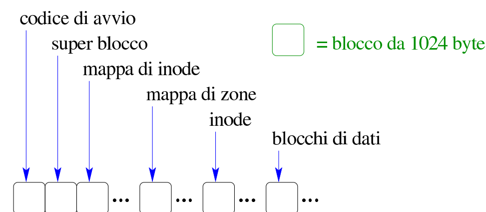
Il super blocco rappresentato dal tipo '**sb_t**' include anche le mappe delle zone e degli inode impegnati. Queste mappe hanno una dimensione fissa in memoria, mentre nel file system reale possono essere di dimensione minore. La tabella di super blocchi, contiene le informazioni dei dispositivi di memorizzazione 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 84.97. 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.
`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.
`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.
`void sb_print (void);`	Funzione diagnostica per la visualizzazione sullo schermo dello stato della tabella dei super blocchi.

84.8.2 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 nu-

mera 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.

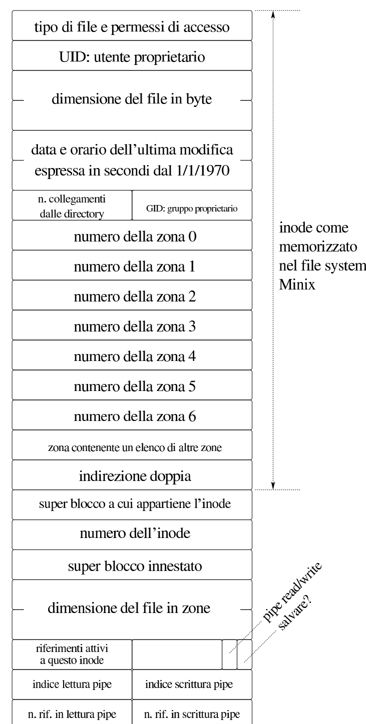
Tabella 84.99. Funzioni per la gestione delle zone, definite nei file '`kernel/fs/zone_...`'.

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***.
`void zone_print (sb_t *sb, zno_t zone);`	Funzione diagnostica per la visualizzazione dello stato di una zona.

84.8.3 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 84.100. Struttura del tipo '**`inode_t`**', corrispondente agli elementi dell'array ***inode_table[]***.



Listato 84.101. Struttura del tipo '**`inode_t`**', corrispondente agli elementi dell'array ***inode_table[]***.

```
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  : 1,
                  pipe_dir : 1;
    unsigned char pipe_off_read;
    unsigned char pipe_off_write;
    unsigned char pipe_ref_read;
    unsigned char pipe_ref_write;
};
```

Figura 84.102. Collegamento tra la tabella degli inode e quella dei super blocchi.

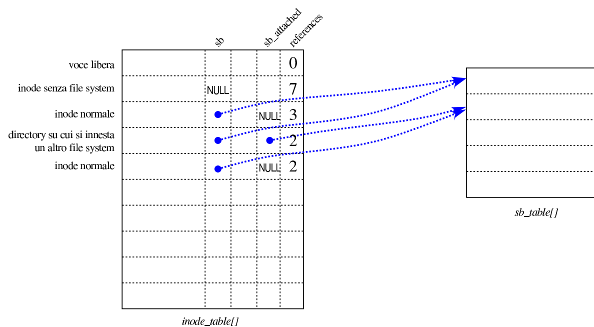


Tabella 84.103. Funzioni per la gestione dei file in forma di inode, definite nei file 'kernel/fs/inode_...'.

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.
`inode_t *inode_alloc (dev_t device, mode_t mode, uid_t uid gid_t gid);`	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 e al gruppo. Se la funzione riesce nel suo intento, restituisce il puntatore all'inode in memoria, il quale rimane così aperto e disponibile per ulteriori elaborazioni.
`int inode_free (inode_t *inode);`	Segna l'inode indicato come libero.
`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.
`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.
`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.
`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.
`inode_t * inode_stdio_dev_make (dev_t device, mode_t mode);`	Alloca un inode in memoria, riferito al dispositivo richiesto dal primo parametro, con i permessi del secondo parametro. Tale inode è adatto per essere utilizzato come flusso standard (standard input, standard output o standard error). Questa funzione viene usata solo da ***file_stdio_dev_make()***, la quale, a sua volta, viene usata solo da ***proc_sys_exec()***.
`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.

Funzione	Descrizione
`blkcnt_t inode_fzones_write (inode_t *inode, zno_t zone_start, void *buffer, blkcnt_t blkcnt);`	Svolge il compito opposto della funzione ***inode_fzones_read()*** e attualmente non viene utilizzata.
`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.
`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.
`int inode_check (inode_t *inode, mode_t type, int perm, uid_t uid, gid_t gid);`	Verifica che l'inode sia di un certo tipo e abbia i permessi di accesso necessari a un certo utente e gruppo. 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.
`inode_t *inode_pipe_make (void);`	Crea un condotto senza nome (*pipe*), restituendo il puntatore all'inode relativo.
`ssize_t inode_pipe_read (inode_t *inode, void *buffer, size_t count, int *eof);`	Legge il contenuto di un condotto, 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.
`ssize_t inode_pipe_write (inode_t *inode, void *buffer, size_t count);`	Scrive una certa quantità di byte nel condotto 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.
`void inode_print (void);`	Funzione diagnostica per la visualizzazione sintetica del contenuto della tabella degli inode.

84.8.4 Fasi dell'innesto di un file system

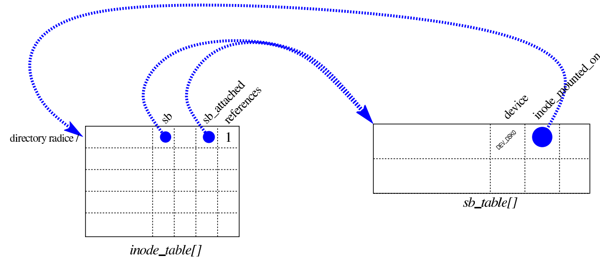
L'innesto e il distacco di un file system, coinvolge simultaneamente la tabella dei super blocchi e quella degli inode. Si distinguono due situazioni fondamentali: l'innesto del file system principale e quello di un file system ulteriore.

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);
    ...
```

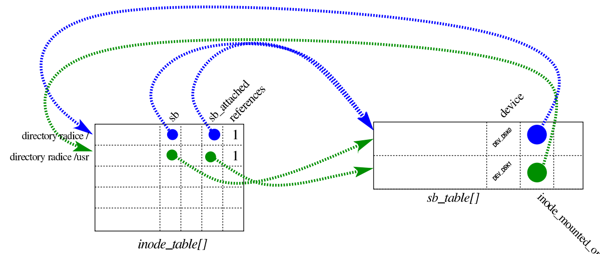
La funzione ***sb_mount()*** carica il super blocco nella tabella relativa, ma trovando il riferimento all'inode di innesto nullo, provvede a caricare l'inode della directory radice dello stesso dispositivo, creando un collegamento incrociato tra le tabelle dei super blocchi e degli inode, come si vede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

Figura 84.105. 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 84.106. Innesto di un file system nella directory '`/usr/`'.

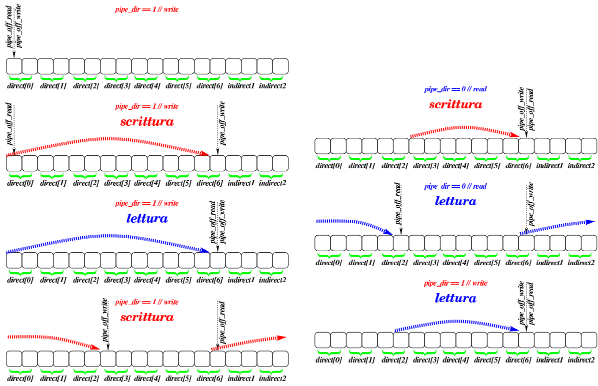


84.8.5 Condotti

I condotti sono gestiti da os32 nel modo tradizionale, sfruttando nell'inode la poca memoria che altrimenti servirebbe per i riferimenti ai blocchi di dati. In pratica, secondo la struttura del tipo '**`inode_t`**', si usa ***direct[]***, ***indirect1*** e ***indirect2***. Ciò comporta complessivamente la disponibilità di soli 18 byte, cosa che comunque sarebbe insufficiente per lo standard attuale dei sistemi Unix.

Lo spazio di questi 18 byte viene trattato come una coda e scandito attraverso due indici: quello di scrittura e quello di lettura. Durante l'accesso all'inode che rappresenta un condotto si distinguono due stati, individuati dal bit ***pipe_dir***.

Figura 84.107. Esempio di utilizzo di un condotto, attraverso varie fasi di scrittura e lettura.



La figura mostra un esempio che dovrebbe chiarire il meccanismo di funzionamento del condotto.

1. Inizialmente gli indici di scrittura e lettura si trovano ad avere lo stesso valore (nella figura si trovano nella posizione zero, ma qualunque altra posizione sarebbe equivalente), mentre il bit ***pipe_dir*** indica «scrittura». In questa situazione, si deve procedere con la scrittura, durante la quale l'indice di scrittura non può superare nuovamente quello di lettura.
2. Nel secondo disegno della figura si vede che è avvenuta una scrittura che ha occupato 13 byte, mentre l'indice di scrittura si trova sul quattordicesimo (quello successivo all'ultimo byte scritto).
3. A questo punto, si suppone che inizi la lettura del condotto: in tal caso, dato che il bit ***pipe_dir*** indica ancora «scrittura», la lettura non può superare la posizione che ha raggiunto l'indice di scrittura. Pertanto si suppone che la lettura raccolga la stessa quantità di byte occupati precedentemente dalla scrittura. Quando l'indice di lettura incontra quello di scrittura, il bit ***pipe_dir*** deve essere impostato a «scrittura», perché non c'è altro che si possa leggere. Tale bit era già impostato nel modo corretto e quindi non si notano variazioni.
4. Nel quarto disegno si vede l'inizio di una nuova fase di scrittura che raggiunge la fine dello spazio dei 18 byte previsti per riprendere dall'inizio. In questa fase di scrittura gli indici non si incontrano e nulla cambia nello stato di ***pipe_dir***.
5. Nel quinto disegno (all'inizio del lato destro), la scrittura riprende e raggiunge l'indice di lettura (che non può essere superato). Qui lo stato rappresentato da ***pipe_dir*** cambia, dal momento che non si può più procedere con la scrittura, adesso indica «lettura».
6. Nel sesto disegno si vede l'inizio di una lettura. Dopo di questa lettura, potrebbe esserci una fase di scrittura, ma senza poter superare l'indice di lettura. Comunque, questa scrittura non viene eseguita.
7. Nell'ultimo disegno si vede che la lettura continua, fino a raggiungere l'indice di scrittura, quando così il valore di ***pipe_dir*** viene invertito nuovamente.

In pratica, quando gli indici di lettura e scrittura coincidono, per sapere se si può procedere con una scrittura o una lettura, occorre chiederlo a ***pipe_dir***; diversamente, con indici diversi, la scrittura o la lettura può procedere indifferentemente, ma solo fino al raggiungimento dell'altro indice. Poi, se è l'indice di lettura che ha appena raggiunto quello di scrittura, ***pipe_dir*** deve essere impostato per richiedere la scrittura; al contrario, quando è l'indice di scrittura che raggiunge quello di lettura, ***pipe_dir*** deve richiedere la lettura successiva.

84.8.6 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 84.108. Struttura del tipo '**`file_t`**', corrispondente agli elementi dell'array ***file_table[]***.

riferimenti attivi a questo file provenienti da descrittori
indice interno di accesso al file
modalità di apertura
riferimento all'inode del file
riferimento al socket del file

```
typedef struct file file_t;

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

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***, oppure a quella dei socket, attraverso il membro ***sock***. Più voci della tabella dei file possono riferirsi allo stesso inode (o allo stesso socket), 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 84.109. Collegamento tra la tabella dei file e quella degli inode.

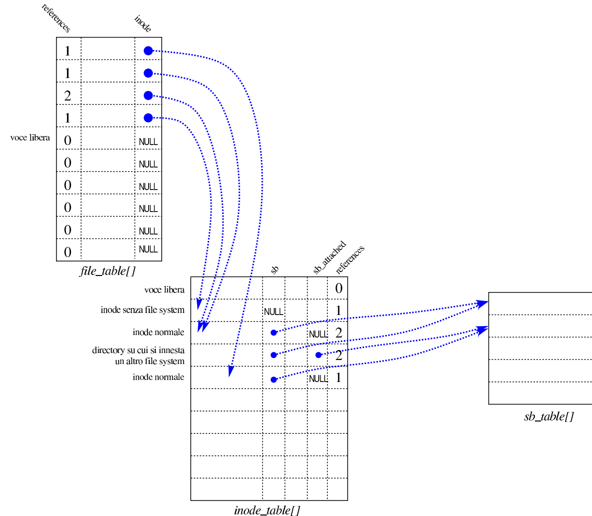


Tabella 84.110. Funzioni fatte esclusivamente per la gestione della tabella dei file ***file_table[]***.

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.

84.8.7 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 84.111. Struttura del tipo '**`fd_t`**', con cui si costituiscono gli elementi delle tabelle dei descrittori di file, una per ogni processo.

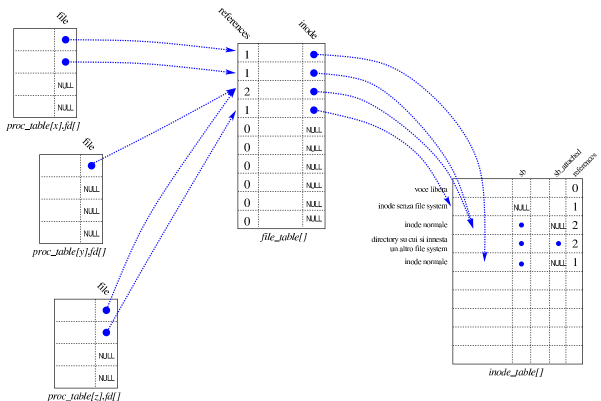
indicatori dello stato del file e delle modalità di accesso
indicatori del descrittore
riferimento alla tabella dei file di sistema

```
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 84.112. 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***.



84.8.8 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 84.113. 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 84.114. 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
`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.
`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_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.

Delle funzioni che, per affinità, farebbero parte di questo gruppo, si trovano nella directory '`kernel/lib_s/`', in quanto servono per attuare delle chiamate di sistema.

Tabella 84.115. 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, contenute nella directory 'kernel/lib_s/'.

Funzione	Descrizione
`int s_chdir (pid_t pid, const char *path);`	Cambia la directory corrente, utilizzando il nuovo percorso indicato. È l'equivalente della funzione standard ***chdir()*** (sezione 87.6).
`int s_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 87.7).
`int s_chown (pid_t pid, const char *path, uid_t uid, gid_t gid);`	Cambia l'utente e il gruppo proprietari del file. È l'equivalente della funzione standard ***chown()*** (sezione 87.8).
`int s_link (pid_t pid, const char *path_old, const char *path_new);`	Crea un collegamento fisico. È l'equivalente della funzione standard ***link()*** (sezione 87.30).
`int s_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 87.34).
`int s_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 87.35).
`int s_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. Le opzioni disponibili sono solo '**MOUNT_DEFAULT**' e '**MOUNT_RO**', come dichiarato nel file di intestazione 'lib/sys/os32.h'.
`int s_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 87.37).
`int s_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 87.55).
`int s_umount (pid_t pid, const char *path_mnt);`	Stacca l'unità innestata nella directory indicata, purché nulla al suo interno sia attualmente in uso.

Funzione	Descrizione
`int s_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 87.62).

84.8.9 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 84.116. Funzioni per la gestione dei file, a cui si fa riferimento attraverso il descrittore, relativamente a un certo processo elaborativo, le quali non rappresentano direttamente la realizzazione di una chiamata di sistema.

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_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***).

Delle funzioni che, per affinità, farebbero parte di questo gruppo, si trovano nella directory 'kernel/lib_s/', in quanto servono per attuare delle chiamate di sistema.

Tabella 84.117. Funzioni per la gestione dei file, a cui si fa riferimento attraverso il descrittore, tenendo conto del processo elaborativo per conto del quale si svolge l'operazione, contenute nella directory 'kernel/lib_s/'.

Funzione	Descrizione
`int s_dup (pid_t pid, int fdn_old);`	Duplica il descrittore ***fdn_old***, creandone un altro (utilizzando il primo numero di descrittore libero) il cui numero viene restituito dalla funzione. È l'equivalente della funzione standard ***dup()*** (sezione 87.12).
`int s_dup2 (pid_t pid, int fdn_old, int fdn_new);`	Duplica il descrittore ***s_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 87.12).
`int s_fchmod (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 87.7).
`int s_fchown (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 87.8).

Funzione	Descrizione

`int s_fcntl (pid_t` ***pid***`,`
`int` ***fdn***`,`
`int` ***cmd***`,`
`int` ***arg***`);`
	Svolge il compito della funzione standard ***fcntl()*** (sezione 87.18).

`int s_fstat (pid_t` ***pid***`,`
`int` ***fdn***`,`
`struct stat *`***buffer***`);`
	Svolge il compito della funzione standard ***fstat()*** (sezione 87.55).

`off_t s_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 87.33).

`int s_pipe (pid_t` ***pid***`,` `int` ***pipefd[2]***`);`	Crea un condotto senza nome (*pipe*), per il quale, i due descrittori necessari vengono salvati in ***pipefd[]***. Per la precisione, ***pipefd[0]*** individua il descrittore del lato di lettura, mentre ***pipefd[1]*** individua quello del lato di scrittura. È l'equivalente della funzione standard ***pipe()*** (sezione 87.38).

`ssize_t s_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 87.39).

`ssize_t s_write (pid_t` ***pid***`,`
`int` ***fdn***`,`
`const void *`***buffer***`,`
`size_t` ***count***`);`
	Scrive nel descrittore. È l'equivalente della funzione standard ***write()*** (sezione 87.64).

84.9 Gestione delle interfacce di rete

Il sistema os32 può gestire soltanto interfacce di rete Ethernet e l'interfaccia virtuale locale, nota con il nome *loopback*. Le interfacce di rete hanno tutte nomi del tipo '**net*n***', dove ***n*** è un numero intero, a partire da zero, e di norma l'interfaccia virtuale locale coincide con il nome '**net0**'.

84.9.1 Gestione dei dispositivi NE2K

Il kernel di os32 è in grado di gestire soltanto le interfacce di rete Ethernet NE2000, collocate nel bus PCI: NE2K. Ciò consente di conoscere la porta di I/O necessaria per accedervi, in modo automatico. Le funzioni per la gestione di queste interfacce sono contenute nei file della directory '`kernel/driver/nic/ne2k/`' e fanno capo al file di intestazione '`kernel/driver/nic/ne2k.h`' (listato 94.4.19 e successivi).

Le interfacce di rete NE2000 dispongono di una piccola memoria tampone interna per la ricezione; tuttavia, appena viene individuato un pacchetto ricevuto, os32 lo trasferisce immediatamente in una propria memoria tampone, contenuta nella tabella delle interfacce, descritta nella sezione successiva.

Per una maggiore semplicità progettuale, la trasmissione di un pacchetto avviene mettendo tutto il sistema in attesa, fino a che l'interfaccia dà un responso, positivo o negativo che sia. Tuttavia, ciò comporta anche il rischio di bloccare definitivamente il sistema, nel caso si dovessero manifestare dei problemi all'interfaccia.

Tabella 84.118. Funzioni per la gestione dell'interfaccia di rete NE2000.

Funzione	Descrizione
`int ne2k_check (uintptr_t` ***io***`);`	Verifica se l'interfaccia corrispondente alla porta di I/O specificata è veramente di tipo NE2000 [94.4.20].
`int ne2k_isr (uintptr_t` ***io***`);`	Verifica lo stato dell'interfaccia e ne acquisisce i dati, se disponibili. In caso di ricezione di una trama, viene chiamata la funzione ***ne2k_rx()*** per trasferirla nella memoria tampone della tabella delle interfacce [94.4.21].

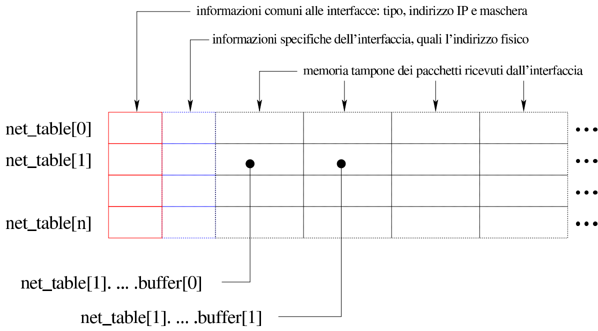
`int`
`ne2k_isr_expect`
`(uintptr_t` ***io***`,`
`unsigned int` ***isr_expect***`);`
	Rimane in attesa fino a che il registro ***ISR*** dell'interfaccia si attiva almeno un indicatore corrispondente a quanto richiesto con il parametro ***isr_expect***. Questa funzione viene usata, in particolare, nella trasmissione dei pacchetti, per i quali occorre verificare quando l'interfaccia ha completato il procedimento [94.4.22].

`int ne2k_reset (uintptr_t` ***io***`,` `void *`***address***`);`	Azzera l'interfaccia e ne estrae l'indirizzo fisico, collocandolo in corrispondenza di ****address*** [94.4.23].
`int ne2k_rx (uintptr_t` ***io***`);`	Copia tutte le trame accumulate nella memoria tampone interna dell'interfaccia, in quella della tabella delle interfacce [94.4.24].
`int ne2k_rx_reset (uintptr_t` ***io***`);`	Reinizializza il processo di ricezione [94.4.25].
`int ne2k_tx (uintptr_t` ***io***`,` `void *`***buffer***`, size_t` ***size***`);`	Trasmette una trama Ethernet, contenuta all'interno di ****buffer***, della lunghezza specificata da ***size***. La funzione attende il completamento dell'operazione, prima di concludere il proprio funzionamento [94.4.26].

84.9.2 Tabella delle interfacce e funzioni accessorie

Nei file '`kernel/net/net_public.c`' [94.12.31] e '`kernel/net.h`' [94.12] viene dichiarata la tabella delle interfacce, corrispondente all'array ***net_table[]***, con lo scopo di contenere la memoria tampone delle trame ricevute da ogni interfaccia. La struttura della tabella è definita dal tipo '**net_t**' e appare semplificata nella figura successiva.

Figura 84.119. Struttura semplificata della tabella delle interfacce.



Listato 84.120. Struttura di ogni elemento della tabella delle interfacce; i dettagli dei membri ***buffer*** non sono evidenziati, ma contengono sempre, a loro volta, i membri ***clock*** e ***size***

```
typedef struct {
  clock_t                 clock;
  size_t                  size;
  …
} net_buffer_eth_t | net_buffer_lo_t;

typedef struct {
  unsigned int type;
  h_addr_t     ip;       // IPv4 address in host byte order.
  uint8_t      m;        // Short netmask.
  union {
    //
    // Ethernet type data:
    //
    struct {
      uint8_t                mac[6];
      uintptr_t              base_io;
      unsigned char          irq;
      net_buffer_eth_t buffer[NET_MAX_BUFFERS];
    } ethernet;
    //
    // Loopback type data:
    //
    struct {
      net_buffer_lo_t buffer[NET_MAX_BUFFERS];
    } loopback;
  };
} net_t;
```

Ogni pacchetto accumulato nella memoria tampone della tabella delle interfacce, oltre al contenuto del pacchetto, include l'orario in cui questo è stato ricevuto (in unità '**`clock_t`**') e la sua dimensione effettiva.

La scansione della tabella richiede generalmente due indici: il numero che individua l'interfaccia e il numero che rappresenta la trama memorizzata (PDU di livello 2 nel caso di interfaccia Ethernet, oppure di livello 3 nel caso di interfaccia virtuale locale), assieme a delle informazioni accessorie. Per esempio, '**`net_table[0].loopback.buffer[`*f*`].clock`**' individua l'orario di ricevimento di un pacchetto con indice *f* dell'interfaccia locale '**`net0`**' (*loopback*), mentre '**`net_table[1].ethernet.buffer[`*f*`].size`**' individua la dimensione del pacchetto *f* dell'interfaccia Ethernet '**`net1`**'.

I pacchetti, a livello della rete fisica, vengono depositati nella memoria tampone della tabella, in corrispondenza dell'interfaccia da cui provengono; da qui, poi, attraverso la funzione ***net_rx()***, i pacchetti vengono passati ai gestori appropriati, cancellandoli dalla tabella originaria.

Tabella 84.121. Funzioni per la gestione della tabella delle interfacce, contenute nella directory '`kernel/net/`', e altre accessorie relative alla gestione Ethernet.

Funzione	Descrizione
`net_buffer_eth_t *` `net_buffer_eth (int` ***n***`);`	Restituisce il puntatore a un elemento libero, o utilizzabile, della memoria tampone dell'interfaccia '**`net`*n***', purché questa sia di tipo Ethernet [94.12.23].
`net_buffer_lo_t *` `net_buffer_lo (int` ***n***`);`	Restituisce il puntatore a un elemento libero, o utilizzabile, della memoria tampone dell'interfaccia '**`net`*n***', purché questa sia di tipo *loopback*, ossia l'interfaccia locale virtuale [94.12.24].
`int net_index (h_addr_t` ***ip***`);`	Restituisce l'indice della tabella delle interfacce, corrispondente all'indirizzo IPv4 fornito come argomento [94.12.27].

`int net_index_eth (h_addr_t` ***ip***`,`
`uint8_t` ***mac[6]***`,`
`uintptr_t` ***io***`);`
	Restituisce l'indice della tabella delle interfacce, corrispondente a uno dei dati forniti come argomento (i valori nulli vengono ignorati), purché si tratti di un'interfaccia Ethernet [94.12.28].

`void net_init (void);`	Inizializza la gestione della rete, utilizzando le informazioni attraverso le opzioni di avvio per configurare anche le interfacce Ethernet [94.12.29].
`void net_rx (void);`	Scandisce i pacchetti memorizzati nella tabella delle interfacce, passandoli al gestore appropriato e rimuovendoli poi dalla tabella [94.12.32].

`int net_eth_ip_tx (h_addr_t` ***src***`,`
`h_addr_t` ***dst***`,`
`const void *`***packet***`,`
`size_t` ***size***`);`
	A partire da un pacchetto IPv4 completo e dagli indirizzi IPv4 di origine e di destinazione, viene assemblata e spedita una trama Ethernet. La funzione richiede separatamente l'indicazione degli indirizzi IPv4 di origine e destinazione, per semplificare il codice, evitando di estrapolarli dal pacchetto IPv4 stesso [94.12.25].

`int net_eth_tx (int` ***n***`,`
`void *`***buffer***`,`
`size_t` ***size***`);`
	Provvede a trasmettere una trama Ethernet attraverso l'interfaccia ***n*** (ovvero ***net_table[n]***), la quale deve essere di tipo Ethernet [94.12.26].

84.9.3 Tabella ARP

Per mantenere memoria delle corrispondenze tra indirizzi IPv4 e indirizzi Ethernet, si utilizza la tabella ARP, descritta nel file '`kernel/net/arp.h`' [94.12.1] e dichiarata nel file '`kernel/net/arp/arp_public.c`' [94.12.6].

Le voci della tabella sono valide per un tempo limitato, definito dalla macro-variabile ***ARP_MAX_TIME*** e periodicamente vengono scandite e cancellate le voci troppo vecchie.

Figura 84.122. Struttura della tabella ARP, costituita da elementi di tipo '**`arp_t`**'.

	time	MAC	IPv4
mac_table[0]			
mac_table[1]			
mac_table[n]			

```
typedef struct {
  time_t    time;
  uint8_t   mac[6];
  h_addr_t  ip;
} arp_t;
```

Tabella 84.123. Funzioni per la gestione della tabella ARP, contenute nella directory '`kernel/net/arp/`'.

Funzione	Descrizione
`void arp_init (void);`	Azzera completamente la tabella ARP: si usa una volta sola all'avvio della gestione della rete [94.12.4].
`void arp_clean (void);`	Azzera le voci della tabella ARP che risultano troppo vecchie e che devono essere rinnovate [94.12.2].
`int arp_index (unsigned char` ***mac[6]***`, h_addr_t` ***ip***`);`	Restituisce l'indice della tabella ARP, corrispondente all'indirizzo Ethernet o all'indirizzo IPv4 fornito [94.12.3].
`arp_t *arp_reference (void);`	Restituisce il puntatore a un elemento della tabella ARP contenente la voce più vecchia, allo scopo presumibile di riutilizzarla per un indirizzo nuovo [94.12.7].
`void arp_request (h_addr_t` ***ip***`);`	Invia una richiesta ARP, preparando il pacchetto relativo e inviandolo attraverso la funzione ***ethernet_tx()*** [94.12.8].
`int arp_rx (int` ***n***`, int` ***f***`);`	Legge dalla tabella delle interfacce il pacchetto individuato dall'indice ***n*** per l'interfaccia e dall'indice ***f*** per la trama relativa. Il pacchetto in questione deve essere relativo al protocollo ARP: se si tratta di una richiesta, provvede a inviare una risposta, se invece si tratta di una risposta, allora aggiorna la tabella ARP [94.12.9].

84.10 Gestione di IPv4

La gestione di IPv4, da parte di os32, è estremamente limitata, per semplificare il codice e la sua comprensione. In particolare non si considerano le opzioni che potrebbero essere contenute tra l'intestazione minima e il contenuto del pacchetto IPv4.

Tabella 84.124. Funzioni per la gestione dei pacchetti a livello IP.

Funzione	Descrizione
`uint16_t ip_checksum (uint16_t *`***data1***`, size_t` ***size1***`, uint16_t *`***data2***`, size_t` ***size2***`);`	Produce il codice di controllo usato nel protocollo IPv4, partendo da due blocchi di dati [94.12.16].
`int ip_rx (int` ***n***`, int` ***f***`);`	Si occupa di acquisire un pacchetto IPv4, dalla tabella ***net_table[]***, per copiarlo nella tabella ***ip_table[]*** e trattarlo per le questioni urgenti [94.12.21].

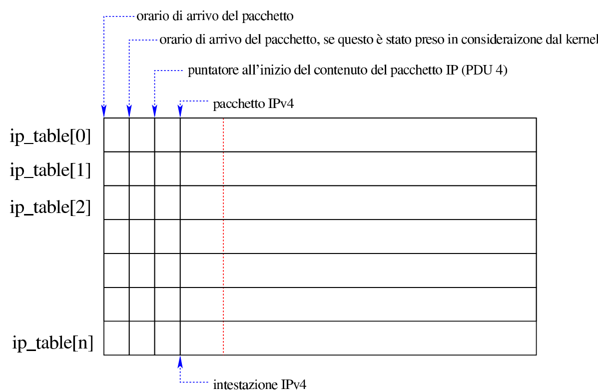
Funzione	Descrizione
`ip_t *ip_reference (void);`	Restituisce il puntatore a un elemento della tabella ***ip_table[]*** che possa essere riutilizzato, perché mai usato prima oppure perché contenente il pacchetto IPv4 ricevuto che risulta essere più vecchio di tutti gli altri [94.12.20].
`ssize_t ip_header (h_addr_t` ***src***`, h_addr_t` ***dst***`, uint16_t` ***id***`, uint8_t` ***ttl***`, uint8_t` ***protocol***`, void *`***buffer***`, size_t` ***length***`);`	Scrive, in corrispondenza di ***buffer***, un'intestazione IPv4, sulla base dei dati contenuti negli altri parametri [94.12.17].
`int ip_tx (h_addr_t` ***src***`, h_addr_t` ***dst***`, int` ***protocol***`, const void *`***buffer***`, size_t` ***size***`);`	Produce e trasmette un pacchetto IPv4, partendo dal contenuto che deve avere e dai dati necessari a costruire l'intestazione IPv4 [94.12.22].
`h_addr_t ip_mask (int` ***m***`);`	A partire da un numero che rappresenta la dimensione di una maschera di rete, si ottiene il valore a 32 bit della maschera stessa. Per esempio, dal valore 16, si ottiene 255.255.0.0 [94.12.18].

In varie situazioni si usa il tipo '**`h_addr_t`**', il quale rappresenta un indirizzo IPv4, a 32 bit, espresso però secondo l'architettura dell'elaboratore (*host byte order*). Questo tipo derivato si contrappone a quello standard, denominato '**`in_addr_t`**', il quale rappresenta lo stesso indirizzo, ma secondo l'ordinamento adatto alla trasmissione in rete (*network byte order*).

84.10.1 Tabella IPv4

Quando un pacchetto viene ricevuto ed è riconosciuto dalla funzione ***net_rx()*** come riguardante IPv4, questa chiama la funzione ***ip_rx()*** che lo copia nella tabella ***ip_table[]***, dove rimane fino a quando viene rimpiazzato da un nuovo pacchetto, secondo il criterio per cui i pacchetti più vecchi lasciano il posto a quelli più recenti.

Figura 84.125. Struttura della tabella dei pacchetti IPv4.



Listato 84.126. Struttura di ogni elemento della tabella dei pacchetti IPv4.

```
typedef struct {
  clock_t   clock;
  clock_t   kernel_serviced;
  uint8_t  *pdu4;
  union {
    uint8_t      octet[NET_IP_MAX_PACKET_SIZE];
    struct iphdr header;
  } packet;
} ip_t;
```

Il membro ***kernel_serviced*** contiene inizialmente il valore zero, per poi ottenere una copia dell'orario di arrivo del pacchetto, appena questo risulta essere stato considerato dal kernel, ai fini del protocollo ICMP (in quanto il protocollo ICMP viene gestito internamente). Quando il kernel trova un pacchetto che ha l'orario di arrivo uguale a quello di elaborazione, sa così che l'ha già preso in considerazione nella propria gestione interna e non deve farci altro.

Il membro ***pdu4*** contiene un puntatore all'inizio del contenuto del pacchetto IPv4, ovvero a ciò che c'è nel pacchetto, dopo l'intestazione IPv4 e dopo le opzioni eventuali.

84.10.2 Ricezione di un pacchetto IPv4

«

La ricezione di un pacchetto IPv4 avviene per opera della funzione ***ip_rx()***, la quale viene avviata da ***net_rx()***, quando si accorge di avere a che fare con un pacchetto di questo tipo.

La funzione ***ip_rx()*** riceve due argomenti, ***n*** e ***f***, i quali rappresentano, rispettivamente, l'indice dell'interfaccia che ha ricevuto la trama e l'indice della trama stessa. Con questi indici, la funzione ***ip_rx()*** è in grado di estrapolare il pacchetto IPv4 dalla tabella ***net_table[]***.

Avendo individuato l'inizio del pacchetto IPv4, verifica l'integrità del contenuto dell'intestazione con il codice di controllo relativo: se la verifica ha successo, e se non si tratta di un frammento (in quanto os32 non gestisce pacchetti frammentati), il pacchetto viene accolto e copiato nella prima posizione disponibile della tabella ***ip_table[]***, annotando l'orario di arrivo.

Il pacchetto ricevuto in questo modo, dovrebbe risultare destinato a un'interfaccia del proprio sistema. Se però l'indirizzo IPv4 di destinazione non è abbinato ad alcuna interfaccia, viene trasmesso un pacchetto ICMP con il messaggio di destinazione irraggiungibile.

Se il pacchetto ricevuto risulta includere informazioni su porte UDP o TCP, viene verificato se nella tabella ***sock_table[]*** è prevista la ricezione nella porta che questo pacchetto dovrebbe raggiungere. Se non è così, viene trasmesso un pacchetto ICMP con il messaggio di porta non raggiungibile.

La tabella ***sock_table[]*** è dichiarata nel file '`kernel/fs.h`' (94.5), perché le connessioni TCP e UDP, a cui si riferisce, hanno un trattamento affine a quello dei file comuni.

Alla fine, se il pacchetto risulta essere di tipo ICMP, viene avviata la funzione ***icmp_rx()*** perché se ne occupi; diversamente viene semplicemente copiato l'orario di ricevimento del pacchetto nel campo che rappresenta l'elaborazione dello stesso a livello IP.

84.10.3 Instradamenti

«

Nella directory '`kernel/net/route/`' si trovano i file delle funzioni che consentono la gestione degli instradamenti, raccolte nel file di intestazione '`kernel/net/route.h`' (listato 94.12.33 e successivi). Per la limitazione di os32, gli instradamenti servono in pratica solo per la trasmissione, in quanto non è previsto il funzionamento in qualità di router (quindi non si pone il problema di reindirizzare i pacchetti ricevuti).

Figura 84.127. Struttura della tabella ***route_table[]*** per la gestione degli instradamenti.

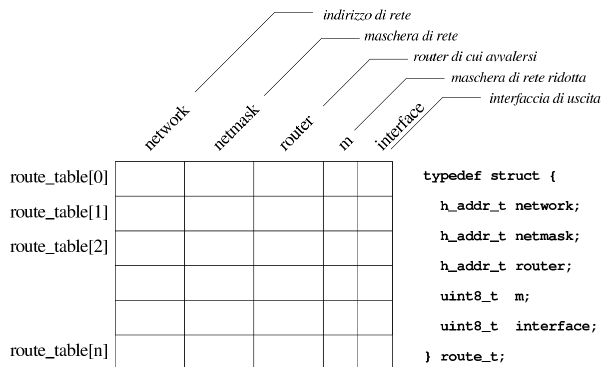


Tabella 84.128. Funzioni per la gestione della tabella degli instradamenti.

Funzione	Descrizione
`void route_init (void);`	Inizializza la tabella ***route_table[]***, predisponendo la voce ***route_table[0]*** per l'interfaccia locale *loopback* [94.12.34].
`void route_sort (void);`	Riordina la tabella degli instradamenti [94.12.39].
`h_addr_t route_remote_to_local (h_addr_t remote);`	Restituisce l'indirizzo IPv4 locale, più adatto per intrattenere una connessione con l'indirizzo remoto fornito come argomento. Questo tipo di analisi viene determinato partendo dalla tabella degli instradamenti, per determinare l'indirizzo IPv4 locale dell'interfaccia interessata dal collegamento [94.12.37].
`h_addr_t route_remote_to_router (h_addr_t remote);`	Restituisce l'indirizzo IPv4 del router da utilizzare per raggiungere l'indirizzo IPv4 remoto specificato. Se l'indirizzo restituito è pari a −1 significa che non è stata ottenuta alcuna voce corrispondente, mentre se si ottiene zero significa che non c'è bisogno di router per raggiungere la destinazione [94.12.38].

84.11 Gestione del protocollo ICMP

«

Listato 94.12.10 e successivi.

Quando viene ricevuto un pacchetto IPv4 che contiene un messaggio ICMP, la funzione ***ip_rx()*** chiama la funzione ***icmp_rx()*** per il trattamento di questa informazione. La funzione ***icmp_rx()*** verifica il tipo di messaggio e si comporta di conseguenza: a una richiesta di eco risponde con la trasmissione di un pacchetto appropriato, attraverso la funzione ***icmp_tx_echo()***; a un messaggio di destinazione irraggiungibile, comunica l'informazione nella tabella ***sock_table[]***, dopo aver trovato lì dentro la voce di una connessione con le caratteristiche appropriate.

Tabella 84.129. Funzioni per la gestione del protocollo ICMP.

Funzione	Descrizione
`int icmp_rx (int i);`	Recepisce un pacchetto ICMP dalla tabella dei pacchetti IPv4, conoscendo l'indice di questa tabella, trattando le richieste di eco e i messaggi di destinazione irraggiungibile [94.12.11].
`int icmp_tx (h_addr_t src, h_addr_t dst, int type, int code, icmp_packet_t *icmp, size_t size);`	Completa l'intestazione di un pacchetto ICMP, con il tipo e il codice del messaggio, oltre al codice di controllo; questo pacchetto poi viene imbustato in un pacchetto IPv4 per la trasmissione [94.12.12].
`int icmp_tx_echo (h_addr_t src, h_addr_t dst, int type, int code, int identifier, int sequence, uint8_t *data, size_t size);`	Prepara un pacchetto ICMP di eco, partendo dal contenuto di questo pacchetto e dalle informazioni da aggiungere nell'intestazione ICMP; l'intestazione di questo pacchetto viene completata attraverso la funzione ***icmp_tx()***, la quale poi provvede a imbustarlo in un pacchetto IPv4 per la trasmissione [94.12.13].
`int icmp_tx_unreachable (h_addr_t src, h_addr_t dst, int type, int code, icmp_packet_t *icmp, size_t size);`	Prepara un pacchetto ICMP con un messaggio di destinazione irraggiungibile, partendo dal contenuto di questo pacchetto e dalle informazioni da aggiungere nell'intestazione ICMP; l'intestazione di questo pacchetto viene completata attraverso la funzione ***icmp_tx()***, la quale poi provvede a imbustarlo in un pacchetto IPv4 per la trasmissione [94.12.14].

84.12 Gestione dei protocolli UDP e TCP

«

Listati 94.5, 94.12.53 e successivi, 94.12.40 e successivi.

Per la gestione dei protocolli UDP e TCP, os32 definisce una tabella in cui ogni voce descrive una connessione. Si tratta della tabella ***sock_table[]*** le cui voci sono collegate dalla tabella ***file_table[]***. In pratica, quando si utilizza la funzione ***socket()*** per creare una connessione UDP o TCP, si ottiene un descrittore di file che fa riferimento a una voce nella tabella ***sock_table[]***, invece che alla tabella degli inode.

Figura 84.130. 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***.

La struttura degli elementi della tabella ***sock_table[]*** è molto articolata. In generale si distingue una prima parte, comune ai protocolli UDP e TCP, rispetto a una seconda specifica per il protocollo TCP.

Listato 84.131. Struttura di ogni elemento della tabella dei socket.

```
struct sock
{
  int         family;
  int         type;
  int         protocol;
  h_addr_t    laddr;          // indirizzo locale
  h_port_t    lport;          // porta locale
  h_addr_t    raddr;          // indirizzo remoto
  h_port_t    rport;          // porta remota
  struct {
    clock_t   clock[IP_MAX_PACKETS];
  } read;
  uint8_t     active      : 1, // socket utilizzato?
              unreach_net : 1, // rete irraggiungibile?
              unreach_host: 1, // destinazione
                               //  irraggiungibile?
              unreach_prot: 1, // protocollo
                               //  irraggiungibile?
              unreach_port: 1; // porta irraggiungibile?
  struct
  {
    uint16_t  conn        : 4, // stato della connessione
              can_write   : 1, // si può scrivere in
                               //  'send_data[]'?
              can_read    : 1, // si può leggere a
                               //  partire da
                               //   '*recv_index'?
              can_send    : 1, // si possono inviare
                               //  dati?
              can_recv    : 1, // si possono ricevere
                               //  dati?
              send_closed : 1, // il canale di
                               //  trasmissione
                               //   è chiuso?
              recv_closed : 1; // il canale di ricezione
                               //  è chiuso?
    //
    uint32_t  lsq[16];        // array della sequenza locale
    uint32_t  lsq_ack;        // numero di sequenza in attesa
                              //  di conferma
    uint32_t  rsq[16];        // array della sequenza remota
    uint8_t   lsqi      : 4,  // indice dell'array della
                              //  sequenza locale
              rsqi      : 4;  // indice dell'array della
                              //  sequenza remota
    //
    clock_t   clock;          // istante dell'ultima
                              //  trasmissione
    //
    uint8_t   send_data[TCP_MSS - sizeof (struct tcphdr)];
                              // dati da trasmettere
```

```
    size_t    send_size;     // dimensione dei dati da
                             //  trasmettere
    int       send_flags;
    uint8_t   recv_data[TCP_MAX_DATA_SIZE];
                             // dati ricevuti
    size_t    recv_size;     // dimensione dei dati
                             //  ricevuti
    uint8_t  *recv_index;    // indice per la lettura dei
                             //  dati ricevuti
    pid_t     listen_pid;    // processo in ascolto della
                             //  porta locale, in attesa di
                             //  connessioni
    int       listen_max;    // numero massimo di richieste
                             //  di connessione accettabili
    int       listen_queue[SOCK_MAX_QUEUE];
                             // descrittori di connessioni
                             //  realizzate
  } tcp;
};
```

Tabella 84.132. Funzioni per la gestione dei protocolli TCP e UDP.

Funzione	Descrizione
`int tcp_tx_raw (h_port_t sport, h_port_t dport, uint32_t seq, uint32_t ack_seq, int flags, h_addr_t saddr, h_addr_t daddr, const void *buffer, size_t size);`	Costruisce un pacchetto TCP, utilizzando i dati forniti come argomenti della chiamata; quindi lo trasmette attraverso la funzione ***ip_tx()*** che a sua volta provvede a imbustarlo in un pacchetto IP prima della trasmissione effettiva. Si tratta comunque di una funzione usata soltanto per fare dei test di funzionamento [94.12.50].
`void tcp_show (h_addr_t src, h_addr_t dst, const struct tcphdr *tcphdr);`	Funzione diagnostica realizzata per visualizzare alcune informazioni su un pacchetto TCP, di cui si conosce il contenuto e gli indirizzi IPv4. Questa funzione viene usata prevalentemente da ***tcp()***, quando si attiva la macro-variabile ***DEBUG*** [94.12.46].
`int tcp_tx_rst (void *ip_packet);`	Sulla base di un pacchetto IP ricevuto con un contenuto TCP, trasmette un pacchetto TCP di azzeramento (RST) [94.12.51].
`int tcp_tx_sock (void *sock_item);`	Trasmette quanto contenuto nella coda del socket indicato come argomento, ammesso che il socket sia nella condizione di poter trasmettere [94.12.52].
`int tcp_tx_ack (void *sock_item);`	Trasmette un pacchetto TCP vuoto contenente la conferma di quanto ricevuto in precedenza, sulla base dello stato attuale del socket indicato come argomento [94.12.49].
`int tcp_rx_ack (void *sock_item, void *packet);`	Verifica che il pacchetto indicato come secondo parametro, contenga una conferma valida per il socket specificato come primo parametro [94.12.44].

Funzione	Descrizione
`int tcp_rx_data (void *sock_item, void *packet);`	Legge il contenuto di un pacchetto TCP e lo copia all'interno della memoria tampone del socket rappresentata da ***sock_table[s].tcp.recv_data*** (dove ***s*** è l'indice del socket considerato) [94.12.45].
`int tcp_connect (void *sock_item);`	Fa in modo di mettere il socket in connessione, ammesso che ciò sia possibile. Questa funzione serve a ***s_connect()*** [94.12.43].
`int tcp_close (void *sock_item);`	Fa in modo di mettere il socket nello stato di chiusura, ammesso che ciò sia possibile. Questa funzione serve a ***s_close()*** [94.12.42].
`int tcp (void);`	Viene chiamata da ***proc_sch_net()*** e si occupa di gestire lo stato di tutte le connessioni TCP in essere in quel momento [94.12.41].
`int udp_tx (h_port_t sport, h_port_t dport, h_addr_t saddr, h_addr_t daddr, const void *buffer, size_t size);`	Assembla un pacchetto UDP e lo trasmette (dopo aver costruito a sua volta il pacchetto IPv4 complessivo) [94.12.54].

84.12.1 UDP

Quando viene ricevuto un pacchetto IPv4 che contiene dati del protocollo UDP, dopo aver verificato che esiste effettivamente una porta UDP in attesa di ricevere nella tabella ***sock_table[]***, questo viene semplicemente lasciato nella tabella ***ip_table[]***, annotando soltanto che il kernel lo ha già preso in considerazione.

L'acquisizione effettiva del pacchetto UDP avviene attraverso la funzione ***s_recvfrom()***, la quale costituisce la versione interna al kernel di ***recvfrom()***. La funzione ***s_recvfrom()***, chiamata per leggere da un socket UDP, cerca nella tabella ***ip_table[]*** un pacchetto corrispondente alle caratteristiche del socket, che non sia già stato preso in considerazione dal socket stesso (il membro ***read.clock[i]***, dove ***i*** corrisponde all'indice del pacchetto trovato nella tabella ***ip_table[]***, contiene l'orario di un pacchetto già letto in quella posizione: se l'orario del pacchetto contenuto nella tabella ***ip_table[]*** è più recente, allora deve essere letto). Se il pacchetto viene accettato, si aggiorna nel socket il valore del membro ***read.clock[i]*** con l'orario di ricevimento del pacchetto (per evitare di rileggerlo un'altra volta), quindi viene copiato il contenuto del pacchetto nella destinazione specificata dagli argomenti della funzione.

Figura 84.133. Semplificazione dei punti principali del procedimento di lettura di un pacchetto UDP, attraverso la funzione ***s_recvfrom()***.

```
s_recvfrom ( pid_t pid, int sfdn, void *buffer, size_t length, int flags, struct sockaddr *addrfrom, socklen_t *addrlen );

individua il socket
sock = &(proc_table[pid].fd[sfdn]->file->sock)
si suppone che si tratti di un socket UDP
sock->type == DGRAM
sock->protocol == IPPROTO_UDP
si scandisce la tabella dei pacchetti IP (con l'indice i) alla ricerca di un pacchetto UDP non alcora letto
ip_table[i].packet.header.protocol == IPPROTO_UDP
ip_table[i].clock < sock->read.clock[i]
si scandisce la tabella dei pacchetti IP alla ricerca di un pacchetto UDP non alcora letto
udp = (struct udphdr *) &ip_table[i].packet.octet[sizeof (stuct iphdr)]);
il pacchetto deve essere destinato alla stessa porta in cui  in ascolto il socket, e deve essere diversa da zero
udp->dest != 0
udp->dest == htons (sock->lport)
il pacchetto deve provenire dalla porta remota che il socket ha già annotato, oppure la porta remota non deve essere ancora conosciuta
udp->source == htons (sock->rport) || sock->rport == 0
gli indirizzi IP devono coincidere o non devono essere conosciuti dal socket
ip_table[i].packet.header.daddr == htonl (sock->laddr) || sock->laddr == 0
ip_table[i].packet.header.saddr == htonl (sock->raddr) || sock->raddr == 0
si annota che il pacchetto è stato letto
sock->read.clock[i] = ip_tabe[i].clock
individua il contenuto del pacchetto UDP e lo copia dove richiesto
data = (((uint8_t *) udp) + sizeof (struct udphdr))

memcpy (buffer, data, length)
```

Va osservato che la lettura del pacchetto UDP, così come viene fatta da os32, si limita alla porzione specificata dalla dimensione massima della memoria tampone; se poi si esegue una nuova lettura, si cerca semplicemente un altro pacchetto, senza terminare eventualmente la lettura del precedente.

La trasmissione avviene attraverso la funzione ***s_send()*** (corrispondente a ***send()*** dal lato utente). Questa funzione, dopo aver determinato che si tratta di un socket UDP, si avvale a sua volta della funzione ***udp_tx()*** per costruire e spedire effettivamente il pacchetto. Come nel caso della ricezione, la trasmissione riguarda un solo pacchetto, e le informazioni eccedenti sono semplicemente eliminate.

84.12.2 TCP

«

La gestione del TCP è estremamente più complessa rispetto a UDP, in quanto richiede un proprio sistema di gestione delle connessioni. La funzione ***tcp()*** ha il compito di scandire la tabella dei pacchetti ***ip_table[]*** alla ricerca di quelli che riguardano il protocollo TCP e che non sono ancora stati considerati, aggiornando lo stato delle connessioni relative. La funzione ***tcp()*** è chiamata a ogni interruzione da ***proc_sch_net()***.

La funzione ***tcp()***, quando individua nella tabella ***ip_table[]*** un pacchetto TCP ancora da prendere in considerazione, deve valutare le caratteristiche del pacchetto trovato in relazione allo stato della connessione eventualmente già in corso, agendo di conseguenza. Ciò significa che la funzione ***tcp()*** può trovarsi nella necessità di trasmettere a sua volta pacchetti TCP alla controparte, per il fine della gestione della connessione.

È importante osservare che gli indicatori ***sock_table[].tcp.can_read*** e ***sock_table[].tcp.can_write***, necessari a controllare la lettura e la scrittura del socket con le funzioni ***s_recvfrom()*** e ***s_send()***, sono aggiornati dalla funzione ***tcp()***.

Le funzioni ***s_recvfrom()*** e ***s_send()***, se si trovano nell'impossibilità di leggere o scrivere il socket richiesto, in condizioni normali mettono il processo relativo in pausa, in attesa di un cambiamento. Inoltre, la funzione ***s_recvfrom()*** deve aggiornare l'indice di lettura interno al socket, in modo che la lettura successiva riprenda da quella posizione. In pratica, lettura e scrittura avvengono qui in modo analogo a quello di un file, in un flusso continuo di byte.

Il risveglio dei processi in attesa di leggere o scrivere un socket avviene per opera della funzione ***proc_sch_net()*** dopo aver avviato ***tcp()*** per aggiornare lo stato dei socket TCP, in base al fatto che sia stato ricevuto qualcosa o che ci sia motivo di ritenere che sia possibile scrivere attraverso un socket bloccato precedentemente in scrittura.

Esiste un grosso limite di os32, relativo alla gestione del TCP: la chiusura di una connessione elimina le informazioni relative al socket, mentre la controparte potrebbe non essere ancora pronta per recepire tale conclusione. Questa semplificazione serve a far sì che ci sia sempre corrispondenza tra il descrittore di file e il socket, mentre in un sistema reale, il socket deve poter continuare a esistere per un certo tempo, benché chiuso, anche dopo la chiusura del descrittore di file relativo.

Va poi considerato che os32 gestisce finestre TCP pari a un solo pacchetto, per cui si attende la conferma di ogni singola trasmissione dalla controparte.