`addr_t` 1326 `diag.h` 1323 `directory_t` 1326 `dsk_chs_t` 1326 `dsk_t` 1326 `fd_t` 1326 `file_t` 1326 `inode_t` 1326 `memory_t` 1326 `offset_t` 1326 `sb_t` 1326 `segment_t` 1326 `tty_t` 1326 `zno_t` 1326

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

os16 contiene uno schedulatore banale e molto limitato, un'organizzazione dei processi ad albero e una funzionalità limitata di amministrazione dei segnali, una gestione del file system Minix 1 (ma di unità intere, senza partizioni), una shell banale e qualche programma di servizio di esempio.

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

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

Organizzazione

Tutti i file di os16 dovrebbero essere disponibili a partire da *allegati/os16*. In particolare i file '`floppy.a`' e '`floppy.b`' sono le immagini di due dischetti da 1440 Kibyte, contenenti un file system Minix 1: il primo predisposto attraverso Bootblocks (sezione u0.5) per l'avvio di un kernel denominato '`boot`'; il secondo usato per essere innestato nella directory '`/usr/`' del primo.

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

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

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

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

```
local TAB=" "
```

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

Le directory

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

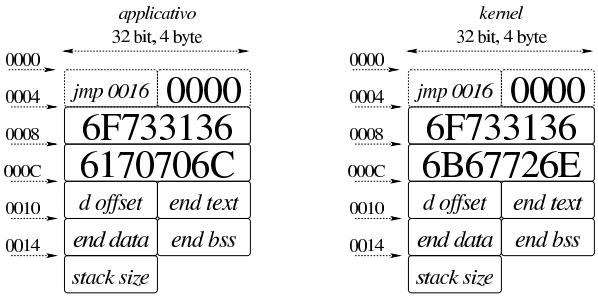
Directory	Contenuto
‘`applic/`’	File delle applicazioni da usare con os16.
‘`kernel/`’	File per la realizzazione del kernel, inclusi i file di intestazione specifici.
‘`lib/`’	File di intestazione generali, file della libreria C per le applicazioni e, per quanto possibile, anche per il kernel.
‘`ported/`’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

La libreria C non è completa, limitandosi a contenere ciò che serve per lo stato di avanzamento attuale del lavoro. Si osservi che nella directory ‘`lib/bcc/`’ si collocano file contenenti una libreria di funzioni in linguaggio assemblatore, necessaria al compilatore Bcc per compiere il proprio lavoro correttamente con valori da 32 bit. Tale libreria è scritta dall’autore originale di Bcc, Bruce Evans, e viene inclusa in modo da garantire che la compilazione non richieda alcun file estraneo.

La struttura degli eseguibili

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

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



Nella figura si mettono a confronto la parte iniziale dell’eseguibile di un applicativo con quella del kernel di os16. Nei primi quattro byte c’è un’istruzione di salto al codice che si trova subito dopo l’intestazione, quindi appare un’impronta di riconoscimento che occupa quattro byte. Tale impronta è la rappresentazione esadecimale della stringa «os16». Successivamente appare un’altra impronta, con cui si distingue se si tratta di un applicativo o del kernel; si tratta in pratica della sequenza di «appl», oppure di «kern». Tuttavia, a causa dell’inversione dell’ordine dei byte, in pratica, se si visualizza il file binario si legge «61so», «lppa» e «nrek».

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

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

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

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

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

Caricamento del kernel

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

Informazioni diagnostiche

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

Tabelle

«

Nel codice del kernel si utilizzano spesso delle informazioni organizzate in memoria in forma di tabella. Si tratta precisamente di array, le cui celle sono costituite generalmente da variabili strutturate. Queste tabelle, ovvero gli array che le rappresentano, sono dichiarate come variabili pubbliche; tuttavia, per facilitare l'accesso ai rispettivi elementi e per uniformità di comportamento, viene abbinata loro una funzione, con un nome terminante per '`…_reference()`', con cui si ottiene il puntatore a un certo elemento della tabella, fornendo gli argomenti appropriati. Per esempio, la tabella degli inode in corso di utilizzazione viene dichiarata così nel file '`kernel/inode/inode_table.c`':

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

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

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

Guida di stile

«

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

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

Tipo	Nome	Utilizzo
`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.
`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.

Tipo	Nome	Utilizzo
`struct tm *`	`timeptr` `timeptr_…`	Puntatore a una variabile strutturata contenente le componenti di un orario.

Tipi derivati speciali

«

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

File di intestazione	Tipo speciale	Descrizione
‘`kernel/memory.h`’	`addr_t`	Variabile scalare, in grado di rappresentare un indirizzo efficace di memoria (un indirizzo che vada da 00000_{16} a $FFFFF_{16}$).
‘`kernel/memory.h`’	`segment_t`	Variabile scalare, a 16 bit, usata per rappresentare il valore di un registro di segmento.
‘`kernel/memory.h`’	`offset_t`	Variabile scalare, a 16 bit, usata per rappresentare lo scostamento di memoria, a partire dall’inizio di un segmento. Questo tipo di variabile non va confuso con il tipo ‘**`off_t`**’, il quale è un valore con segno e di rango maggiore rispetto a questo ‘**`offset_t`**’.
‘`kernel/memory.h`’	`memory_t`	Variabile strutturata, adatta a contenere tutte le coordinate utili a individuare una certa area di memoria, secondo l’architettura prevista da os16.
‘`kernel/tty.h`’	`tty_t`	Variabile strutturata, adatta a contenere le informazioni e lo stato di un terminale.
‘`kernel/ibm_i86.h`’	`dsk_t`	Variabile strutturata, adatta a contenere le informazioni hardware di una certa unità di memorizzazione.
‘`kernel/ibm_i86.h`’	`dsk_chs_t`	Variabile strutturata, adatta a contenere le coordinate di un certo settore (cilindro, testina e settore) in un’unità di memorizzazione.
‘`kernel/fs.h`’	`zno_t`	Variabile scalare, per rappresentare un numero di una zona, secondo la terminologia del file system Minix.
‘`kernel/fs.h`’	`sb_t`	Variabile strutturata, adatta a contenere tutte le informazioni di un super blocco, relativo a un dispositivo di memorizzazione innestato.
‘`kernel/fs.h`’	`inode_t`	Variabile strutturata, adatta a contenere tutte le informazioni di un inode aperto nel sistema.
‘`kernel/fs.h`’	`file_t`	Variabile strutturata, adatta a contenere i dati di un file di sistema.
‘`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.