Caricamento ed esecuzione del kernel

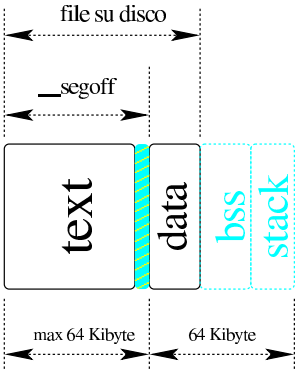
Dal file su disco alla copia in memoria 1327
File «kernel/main/crt0.s» 1328
File «kernel/main.h» e «kernel/main/*» 1330

`crt0.s` 1328 `main()` 1330

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

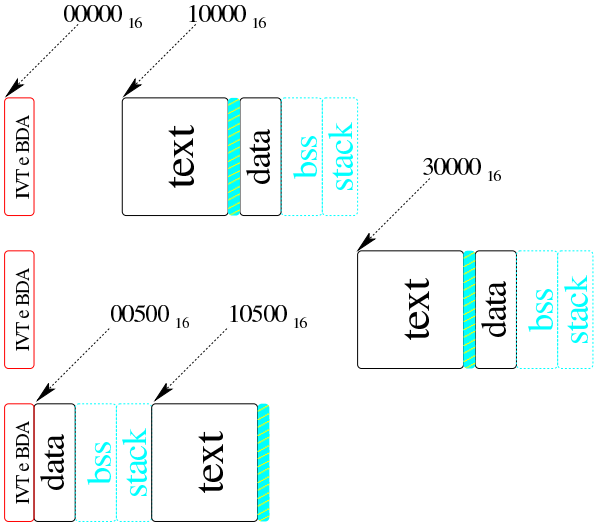
Dal file su disco alla copia in memoria

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



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

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



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

File «kernel/main/crt0.s»

Listato i160.7.2.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Utilizzando il compilatore Bcc per compilare ciò che descrive la funzione ***main()***, viene richiesta la presenza della funzione ***__mkargv()***

(il simbolo '**___mkargv**'), che in questo caso può limitarsi a non fare alcunché.

```
…
___mkargv:
    ret
…
```

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

« Listato u0.7 e successivi.

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

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

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

Dopo la dichiarazione delle variabili si inizializza la gestione del video della console con la funzione ***tty_init()***, si mostra un messaggio iniziale, quindi si passa alla predisposizione di ciò che serve, prima di poter avviare dei processi. In particolare va osservata la funzione ***heap_clear()***, la quale inizializza con il codice $FFFF_{16}$ lo spazio di memoria libero, tra la fine delle variabili «statiche» e il livello che ha raggiunto in quel momento la pila dei dati. Successivamente, avendo marcato in questo modo quello spazio, diventa possibile riconoscere empiricamente quanto spazio di quella porzione di memoria avrebbe potuto essere utilizzato, senza essere sovrascritto dalla pila dei dati. Il messaggio iniziale contiene la data di compilazione e la memoria libera (la macro-variabile ***BUILD_DATE*** viene definita dallo script '**makeit**', usato per la compilazione, creando il file '`kernel/main/build.h`' che viene poi incluso dal file '`kernel/main/main.c`').

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

```
…
int
main (int argc, char *argv[], char *envp[])
{
    …
    menu ();
    for (exit = 0; exit == 0;)
      {
        sys (SYS_0, NULL, 0);
        dev_io ((pid_t) 0, DEV_TTY, DEV_READ, 0L, &key, 1,
                NULL);
        …
        switch (key)
          {
            case 'h':
                menu ();
                break;
            …
            case 'x':
                exit = 1;
                break;
            case 'q':
                k_printf ("System halted!\n");
                return (0);
                break;
          }
      }
    }
…
```

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

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

Le funzioni principali disponibili in questa modalità diagnostica sono riassunte nella tabella successiva:

Tasto	Risultato
$[h]$	Mostra il menù di funzioni disponibili.
$[1]$	Invia il segnale '**SIGKILL**' al processo numero uno.
$[2]$...$[9]$	Invia il segnale '**SIGTERM**' al processo con il numero corrispondente.
$[A]$...$[F]$	Invia il segnale '**SIGTERM**' al processo con il numero da 10 a 15.
$[a]$, $[b]$, $[c]$	Avvia il programma '`/bin/aaa`', '`/bin/bbb`' o '`/bin/ccc`'.
$[f]$	Mostra l'elenco dei file aperti nel sistema.
$[m]$, $[M]$	Innesta o stacca il secondo dischetto dalla directory '`/usr/`'.
$[n]$, $[N]$	Mostra l'elenco degli inode aperti: l'elenco è composto da due parti.
$[l]$	Invia il segnale '**SIGCHLD**' al processo numero uno.
$[p]$	Mostra la situazione dei processi e altre informazioni.
$[x]$	Termina il ciclo e successivamente si passa all'avvio di '`/bin/init`'.
$[q]$	Ferma il sistema.

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

```
…
int
main (int argc, char *argv[], char *envp[])
{
    …
    menu ();
    for (exit = 0; exit == 0;)
      {
        …
      }
    exec_argv[0] = "/bin/init";
    exec_argv[1] = NULL;
    pid = run ("/bin/init", exec_argv, NULL);
    while (1)
      {
        sys (SYS_0, NULL, 0);
      }
    …
}
```

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

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

Figura u143.16. Aspetto di os16 in funzione mentre visualizza anche la tabella dei processi avviati (tasto [*p*]).

```
ababaaababaaababaabbaaababaaababap
pp  p pg
id id rp  tty  uid euid suid usage s iaddr isiz daddr dsiz sp   name
 0  0  0 0000    0    0    0 0.35 R 10500 eb7c 00500 0000 ffc8 os16 kernel
 0  1  0 0000    0    0    0 0.33 r 2f100 0600 2f700 aa00 a8e8 /bin/ccc
 0  2  0 0000    0    0    0 0.01 r 1f100 0600 84300 aa00 a8e8 /bin/ccc
 2  3  0 0000   10   10   10 0.01 r 1f100 0600 44b00 aa00 a8e8 /bin/ccc
 0  4  0 0000    0    0    0 0.17 r 21600 0600 3a100 aa00 a8e8 /bin/ccc
 4  5  0 0000   10   10   10 0.02 r 21c00 2400 6f100 a900 a86c /bin/aaa
 4  6  0 0000   11   11   11 0.02 s 24000 2500 59d00 a900 a8b6 /bin/bbb
 0  7  0 0000    0    0    0 0.13 r 26500 0600 64600 aa00 a8e8 /bin/ccc
 7  8  0 0000   10   10   10 0.02 r 26b00 2400 8ee00 a900 981e /bin/aaa
 7  9  0 0000   11   11   11 0.02 s 2bf00 2500 79a00 a900 a8b6 /bin/bbb
CS=1050 DS=0050 SS=0050 ES=0050 BP=ffe4 SP=ffe4 heap_min=878c etext=eb7c edata=1
b3c ebss=4c34 ksp=ffc8 clock=0000084b, time elapsed=00:01:57
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffe00000003fffffffffffffffffffffffffffffff
fffe000000000001fffffffff0007fffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff800
00000000000000000000000000000000000000007fffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffbfffffffffffffffffffffffffffffffffffffffffffe000000000000000000000000f

abbaaaaabababaaabaaabbaabbaabababbaabbbbbbbbbbbbbb
```

Figura u143.17. Aspetto di os16 in funzione con il menù in evidenza, dopo aver premuto il tasto [*x*] per avviare '**`init`**'.

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