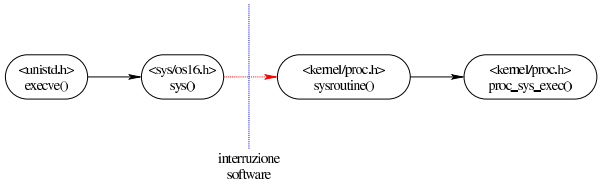
Caricamento ed esecuzione delle applicazioni

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



Caricamento in memoria

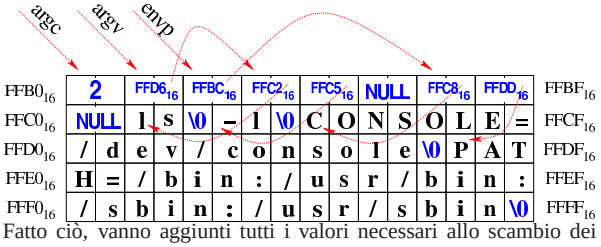
La funzione ***proc_sys_exec()*** (listato i160.9.21) del kernel è quella che svolge il compito di caricare un processo in memoria e di annotarlo nella tabella dei processi.

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

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

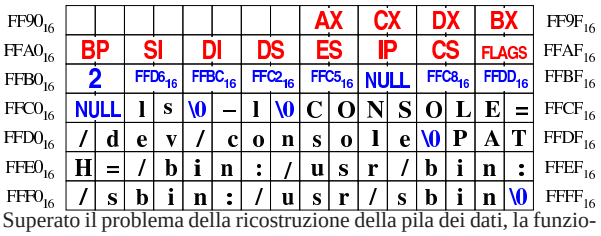
Terminato il caricamento del file, viene ricostruita in memoria la pila dei dati del processo. Prima si mettono sul fondo le stringhe delle variabili di ambiente e quelle degli argomenti della chiamata, quindi si aggiungono i puntatori alle stringhe delle variabili di ambiente, ricostruendo così l'array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

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



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

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



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

Il codice iniziale dell'applicativo

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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