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

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

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

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

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

«

Listati i160.9.1 e i160.9.2.

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

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

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

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

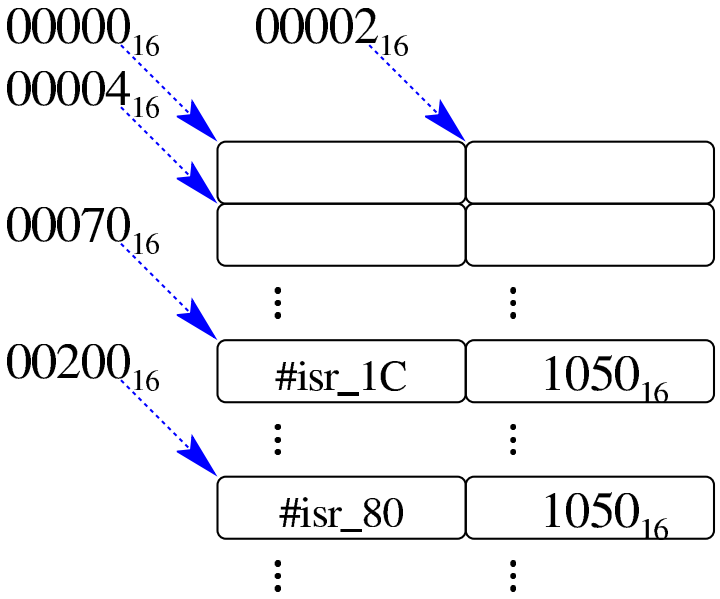
Nel file '`kernel/proc/_ivt_load.s`', la funzione ***_ivt_load()*** che inizia con il simbolo '**__ivt_load**', modifica la tabella IVT (*Interrupt vector table*) in modo che le interruzioni $1C_{16}$ e 80_{16} portino all'esecuzione del codice che inizia rispettivamente in corrispondenza dei simboli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

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

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

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

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



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

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

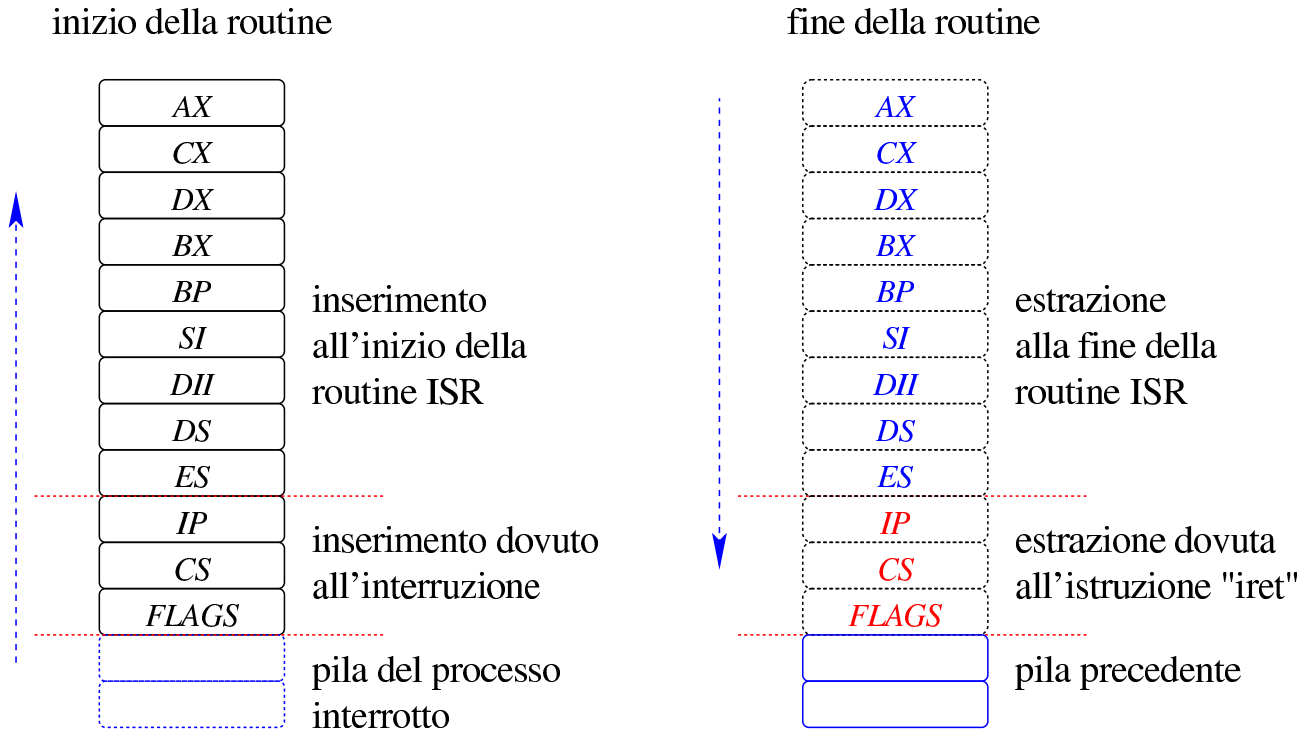
```
...
isr_1C:|isr_80:
```

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

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

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

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

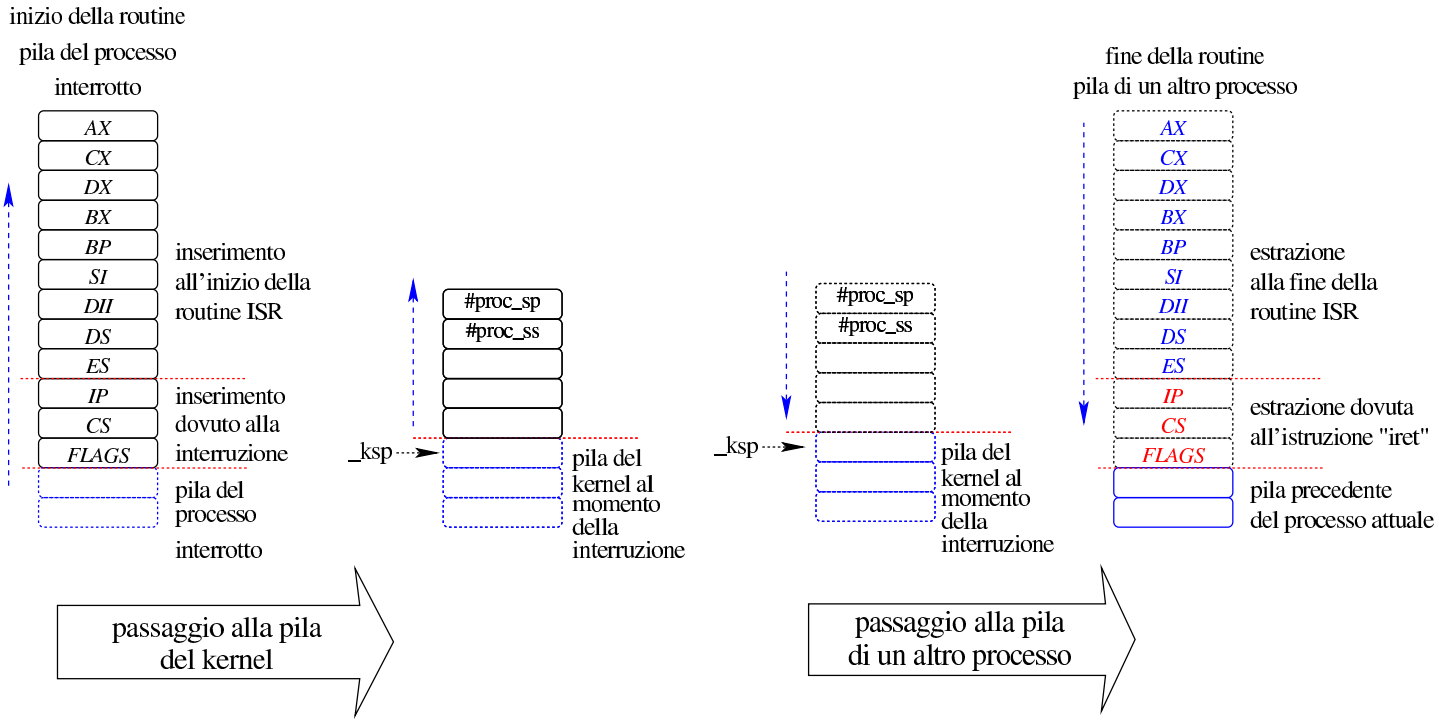


Dopo il salvataggio nella pila dei registri principali, viene modificato il valore dei registri ***DS*** e ***ES***, per consentire l'accesso alle variabili dichiarate all'inizio del file '`kernel/_isr.s`'. Il valore che si attribuisce a tali registri è 0050_{16}, perché il segmento dati del kernel inizia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

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

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

Figura u149.6. Scambi delle pile.



Routine «isr_1C»

«

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

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

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

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

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

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

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

```
    …
    mov dx, proc_ss_0
```

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

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

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

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

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

```
    …
```

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

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

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

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

Routine «isr_80»

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

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

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

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

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

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

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

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

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

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

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

diverso da quello interrotto inizialmente.

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

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

La tabella dei processi

«

Listato u0.9.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chiamate di sistema

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

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

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

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

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

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

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

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

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

File «kernel/proc/...»

«

Listati successivi a u0.9.

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

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

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

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

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

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

Funzione «proc_init()»

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

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

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

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

Funzione «sysroutine()»

«

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

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

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

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

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

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

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

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

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

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

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

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

```
msg.tipo_chiamata
```

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

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

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

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

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

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

Funzione «proc_scheduler()»

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

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

Inizialmente, la funzione acquisisce il numero del processo interrotto:

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

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

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

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

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

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

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

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

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

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