CAPITOLO 4 - SCHEDULAZIONE E CAMBIO DI CONTESTO

Un sistema operativo realizza l'illusione dei processi concorrenti cambiando rapidamente un processore tra parecchie computazioni. Poiché la velocità di esecuzione è estremamente veloce rispetto a quella umana, l'effetto è impressionante - molte attività sembrano procedere simultaneamente.

Il cambio di contesto giace nel cuore delle azioni svolte sui processi. Esso arresta la corrente computazione, salvando abbastanza informazioni in modo da poter essere rieseguita, e fa ripartire un altro processo. Ciò che rende difficoltoso il cambio è il fatto che la CPU non può essere fermata del tutto - deve continuare ad eseguire il codice per passare al nuovo processo.

Questo capitolo descrive i meccanismi di base del cambio di contesto, mostrando esattamente come un processo salva l'informazione riguardante il suo stato, come sceglie un altro processo da eseguire tra quelli che sono pronti e il rilascio del controllo a questo processo. Vengono descritte anche le strutture dati che mantengono le informazioni dei processi non in esecuzione, e mostrate le modalità d'uso di queste strutture con il cambio di contesto. Per adesso ignoriamo le questioni di quando e perché i processi scelgono di cambiare contesto. I capitoli successivi affrontano questo problema, mostrando come le parti più alte del sistema usano il cambio di contesto costruito qui.

4.1 La tabella dei processi

Il sistema mantiene le informazioni dei processi in una struttura dati chiamata *tabella dei processi*. Esiste un elemento per processo in questa tabella. Poiché un solo processo è in esecuzione in un dato momento, uno di questi elementi corrisponde al processo attivo. Tutte le altre componenti della tabella dei processi contengono informazioni sui processi che sono temporaneamente sospesi. Per cambiare contesto il sistema operativo salva le informazioni del processo in esecuzione nel suo elemento della tabella dei processi e preleva i dati - sempre dalla tabella dei processi - del corrispondente processo da eseguire.

Quali indicazioni devono essere salvate esattamente nella tabella dei processi? Il sistema deve salvare ogni valore che sarebbe distrutto quando il nuovo processo viene caricato. Per esempio, in PC-Xinu ogni processo possiede la sua memoria stack separata, quindi una copia dello stack non deve essere salvata. (Il nuovo processo in esecuzione può cambiare i registri della macchina, ma questi saranno salvati sullo stack per quel processo.) In aggiunta ai dati che devono essere ricaricati quando un processo viene ripreso, il sistema tiene nella tabella dei processi anche informazioni che esso usa per controllare il processo e la descrizione delle risorse utilizzate. I dettagli diverranno più chiari dopo aver visto l'uso della tabella dei processi.

La tabella dei processi di PC-Xinu, *proctab*, è un array per al più *NPROC* processi. Essa è dichiarata nel file *proc.h* mostrato di seguito. Ogni elemento in *proctab* è una struttura denominata *pentry* che definisce le informazioni tenute per ogni processo.

```
/* proc.h - isbadpid */
/* 8086 version */

/* dichiarazione della tabella dei processi e costanti definite */

#ifndef     NPROC                   /* setta il numero di processi */
#define     NPROC       30          /* se non già fatto */
#endif
```

```
/* costanti dello stato di un processo */

#define PRCURR      '\01'   /* processo correntemente in esecuzione */
#define PRFREE      '\02'   /* elemento di processo libero */
#define PRREADY     '\03'   /* il processo è nella coda dei pronti */
#define PRRECV      '\04'   /* processo in attesa di un messaggio */
#define PRSLEEP     '\05'   /* processo in stato di sleeping */
#define PRSUSP      '\06'   /* processo in stato suspended */
#define PRWAIT      '\07'   /* il processo è sulla coda dei semafori */

/* definizioni di processo miscellaneo */

#define     PNMLEN            9          /* lunghezza del "nome" del processo */
#define     NULLPROC    0                /* identificatore del processo nullo */
                                         /* sempre eleggibile per l'esecuz. */

#define BADPID          (-1)    /* usato quando necessita pid invalido */

#define     isbadpid(x) (x<=0 || x>=NPROC)

/* elemento della tabella dei processi */

struct      pentry      {
      char pstate;      /* stato del processo: PRCURR, etc. */
      int   pprio;      /* priorità del processo */
      int   psem;       /* semaforo sul quale il processo sta attendendo */
      int   pwaitret;   /* valore di ritorno da una wait */
      long  pmsg;       /* messaggio mandato a questo processo */
      int   phasmsg;    /* diverso da zero se pmsg è valido */
      int   pimmortl;   /* diverso da zero se il processo è immortale */
      char  *pregs;     /* ambiente salvato per cambio di contesto */
      char  *pbase;     /* base dello stack in esecuzione */
      int     ssize;    /* dimenzione dello stack */
      word    plen;     /* lunghezza dello stack in byte */
      char  *pglob;     /* puntatore a dati globali privati */
      para  pmlist;     /* lista di memoria allocata  */
      word  pmalloc;    /* para della memoria allocata */
      int   ptarg;            /* argument to ptfn     */
      int   (*ptfn)();  /* funzione di trap */
      int   phastrap;   /* diverso da zero se una trap è pendente */
      char  pname[PNMLEN+1];  /* nome del processo */
      int   pargs;            /* numero iniziale di argomenti */
      int   (*paddr)();       /* indirizzo iniziale del codice */
      int   pnxtkin;          /* next-of-kin notificato a morire */
      Bool  ptcpumode;        /* processo in modo urgente TCP */
      int     pdevs[Nsio];    /* devices da chiudere dopo l'uscita */
      long    oldtime;        /* tempo dopo il quale resched lo rende corrente*/
      long    time;           /* tick di tempo CPU usato */
};

extern      struct      pentry proctab[];
extern      int   numproc;          /* processi correntemente attivi */
extern      int   nextproc;         /* cerca indicazioni per slot libero */
extern      int   currpid;          /* processo correntemente in esecuzione */
extern      char    *_pglob;        /* puntatore a dati globali privati */
```

Per tutto PC-Xinu, ogni processo è identificato da un intero. Le seguenti regole danno la relazione tra questi interi e la tabella di processi:

> *I processi sono riferiti dal loro identificatore di processo, che è l'indice dell'informazione di stato salvata in proctab.*

Solo alcuni campi contengono dati necessari per riprendere un processo; altri campi mantengono informazioni contabili e controlli di errore. Per esempio, il campo *pbase*, *plen*, *pargs* e *pname* contengono l'indirizzo dello stack del processo, la lunghezza dello stack, il numero di argomenti passati al processo quando è stato creato e una stringa di caratteri che identifica il processo. Alcuni di questi valori sono usati per liberare memoria quando un processo completa; altri sono utili soltanto per il debugging.

4.2 Stati di un processo

Il sistema usa il campo *pstate* della tabella dei processi per aiutare a tener traccia di ciò che sta facendo il processo e, conseguentemente, la validità delle operazioni eseguite su di esso. Il progettista di sistemi deve elaborare questo insieme di stati sin da principio. L'insieme deve essere ben definito prima che l'implementazione cominci perché molte routine che manipolano processi basano le loro azioni sullo stato del processo, richiedendo al programmatore di considerare con cura ogni caso.

In PC-Xinu sono usati in seguenti sei stati: *current, ready, receiving, sleeping, suspended* e *waiting*. Il file *proc.h* contiene costanti simboliche per ognuno di essi che sono usate nel codice dappertutto: PRCURR, PRREADY, PRRECV, PRSLEEP, PRSUSP, e PRWAIT. In aggiunta ai valori citati, il campo *pstate* contiene PRFREE quando nessun processo sta usando l'elemento della tabella dei processi. Più avanti esploreremo ogni stato in dettaglio vendendo il motivo per cui sono sorti e come un processo si muove attraverso essi. Solo lo stato current e ready ci concerne per il momento.

4.3 Selezionare un processo ready

Quasi ogni sistema ha bisogno dello stato *ready* e *current*. I processi sono classificati *ready* quando sono eleggibili per il servizio della CPU ma non sono correntemente in esecuzione; il singolo processo ricevente il servizio della CPU è classificato come corrente. Il cambio di contesto consiste in due passi: selezionare un processo tra quelli *ready* (o *current*) e dare il controllo alla CPU con il processo selezionato. Il software che implementa la politica di selezione del processo tra quelli pronti ad eseguire è chiamato *scheduler*. In PC-Xinu la procedura *resched* effettua la selezione in sintonia con la seguente politica di schedulazione:

> *In ogni momento, il processo eleggibile per la CPU con priorità maggiore viene eseguito. Tra processi aventi uguale priorità la schedulazione è di tipo round-robin.*

Round-robin significa che i processi sono selezionati uno dopo l'altro, cosicchè tutti i membri di un insieme hanno una opportunità di essere eseguiti prima che ogni altro membro ne abbia una seconda. Le priorità, tenute nel campo *pprio* dell'elemento della tabella dei processi, sono interi positivi che danno all'utente un controllo ulteriore per la selezione dei processi per l'esecuzione. (Sistemi più complessi aggiustano le priorità di volta in volta, in base al comportamento dei processi.)

Per rendere la selezione di un nuovo processo più veloce, tutti i processi pronti appaiono in una lista ordinata per priorità, cosicchè il processo con priorità più alta viene immediatamente eseguito. *Resched* usa il meccanismo delle code descritto nel capitolo 3 per esaminare e aggiornare la lista. Esso usa le priorità come chiave e tiene la lista ordinata in base a tale valore, perciò i processi con priorità maggiore si trovano in fondo alla lista. Le variabili globali *rdyhead* e *rdytail*

puntano alla testa e alla coda della lista dei processi pronti nella struttura *q*. Se il processo corrente dovrebbe essere tenuto nella lista dei pronti dipende principalmente dai dettagli di ogni implementazione, ma l'intero sistema deve essere progettato per obbedire alla stessa regola. In PC-Xinu

> *Il processo corrente non è presente nella lista dei pronti, ma il suo identificatore di processo è sempre mantenuto nella variabile globale currpid.*

Consideriamo ciò che accade al processo corrente durante un cambio di contesto. Spesso il processo in esecuzione rimane selezionabile all'uso della CPU sebbene deve temporaneamente passare il controllo a un altro processo. In tale situazione il cambio di contesto deve mutare lo stato del processo in PRREADY e muoverlo nella lista dei pronti per poter essere nuovamente considerato. Come decide lo schedulatore se muovere il processo corrente sulla lista dei pronti? Esso non riceve un parametro esplicito che dà le disposizioni sul processo corrente. Invece le routine di sistema cooperano nel salvare il processo nel seguente modo: se il processo in esecuzione non rimarrà eleggibile per l'uso della CPU, routine di sistema assegnano al campo *pfield* lo stato desiderato prima di chiamare *resched*. Ogniqualvolta *resched* prepara il cambio di contesto esso controlla *pstate* e lo mette nello stato di pronto solo se esso indica il valore PRCURR.

In alcune situazioni è necessario sospendere la schedulazione temporaneamente mentre attività di sistema critiche stanno avvenendosi. Tale sospensione rende possibile a un processo l'accesso esclusivo alla CPU anche se gli interrupt sono abilitati. La procedura *sys_pcxget* ritorna un valore diverso da zero se la schedulazione è permessa, zero altrimenti. Se il processo corrente chiama *resched* quando non permesso, la procedura ritorna immediatamente. Dato che ogni ritorno da *resched* deve lasciare un processo nello stato corrente, avviene un errore se un processo entra in *resched* quando la schedulazione è sospesa e il processo non è il corrente. Conseguentemente viene invocata la procedura di sistema *panic* per arrestare PC-Xinu se occorre tale errore. La sospensione della schedulazione verrà descritta con maggiori dettagli nel capitolo 9.

Oltre a muovere il processo nella lista dei pronti, *resched* completa ogni dettaglio di schedulazione e cambio di contesto eccetto il salvataggio e la restituzione dei registri macchina e il cambiamento dello stack (che non può essere effettuato direttamente in un linguaggio di alto livello come il C). Esso seleziona un nuovo processo da eseguire, cambia il relativo elemento della tabella dei processi, rimuove il processo dalla lista dei pronti, lo marca corrente e aggiorna *currpid*. Esso resetta pure il contatore di prelazione che considereremo più avanti. Infine chiama *ctxsw* per salvare i registri correnti, cambiare stack, e restituire i registri per il nuovo processo. Il codice è mostrato di seguito.

```
/* resched.c - resched */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <dos.h>
#include <q.h>
#include <sleep.h>

/*------------------------------------------------------------------------
 *  resched  --  schedulazione del processore al processo con priorità più alta
 *
 *  Notes:  Currpid dà l'identificatore del processo corrente.
 *          Proctab[currpid].pstate dà lo stato del processo.
 *------------------------------------------------------------------------
 */
int   resched()
{
```

```
register struct pentry  *optr;  /* puntatore all'elem. del vecchio processo */
register struct pentry  *nptr;  /* puntatore all'elem. del nuovo processo */

      optr = &proctab[currpid];
      if ( optr->pstate == PRCURR ) {
      /* nessun cambiamento necessitava se la priorità corrente era più alta */
      /* della successiva o se la schedulazione era disabilitata */
            if (sys_pcxget() == 0 || lastkey(rdytail) < optr->pprio)
                  return;
            /* forza il cambio di contesto */
            optr->pstate = PRREADY;
            insert(currpid,rdyhead,optr->pprio);
      } else if ( sys_pcxget() == 0 ) {
            kprintf("pid=%d state=%d name=%s",
                  currpid,optr->pstate,optr->pname);
            panic("Reschedule impossible in this state");
      }

      /* rimuove il processo con maggiore priorità alla fine della lista dei
pronti */

      nptr = &proctab[(currpid=getlast(rdytail))];
      nptr->pstate = PRCURR;        /* lo marca corrente */
      preempt = QUANTUM;            /* resetta il contatore di prelazione */
      ctxsw(&optr->pregs,&nptr->pregs);

      /* Il vecchio processo ritorna qui quando viene ripreso. */
      return;
}
```

Resched usa la procedura *ctxsw,* per salvare lo stato del processo e cambiare stack, perché registri e puntatori a stack non possono essere manipolati con linguaggi di alto livello. Il codice di *ctxsw*, naturalmente, è dipendente dall'architettura della macchina. Quando *ctxsw* cambia i processi, i registri FLAG devono essere salvati poichè contengono lo stato di interrupt del processo. Gli altri registri che bisogna salvare sono BP, SI e DI. *Ctxsw* salva questi registri sullo stack allo stessa maniera dei programmi assembler visti nel capitolo 2. Ovviamente il puntatore allo stack deve essere cambiato solo dopo aver protetto questi registri perché non appena cambia la CPU il contesto userà la stack del nuovo processo. Da questo punto in poi il processore lavora con lo stack del nuovo processo e tutte le sottosequenze di operazioni si riferiranno al nuovo stack. I registri DI, SI, FLAGS e BP del nuovo processo sono caricati dal suo stack. Al ritorno da *ctxsw* il puntatore all'istruzione sarà settato al codice del nuovo processo e la macchina riprenderà l'esecuzione delle istruzioni associate al nuovo processo.

I parametri passati a *ctxsw* sono puntatori al campo *pregs* del corrente e nuovo processo. Quello del nuovo è usato per ottenere l'ambiente stack, mentre quello del vecchio è usato per memorizzare l'ambiente stack.

```
; ctxsw.asm - _ctxsw

        include ..\h\dos.asm

      dseg
; null data segment
      endds

      pseg

      public      _ctxsw

;--------------------------------------------------------------------------
; _ctxsw  --  cambio di contesto
```

```
;-------------------------------------------------------------------------------
; void ctxsw(opp,npp)
; char *opp, *npp;
;-------------------------------------------------------------------------------
; Contenuto dello stack sull'entrata a ctxsw:
;     SP+8 => segment:offset, indirizzo della nuova area stack
;     SP+4 => segment:offset, indirizzo della vecchia area stack
;     SP   => indirizzo di ritorno
; Gli indirizzi del vecchio e nuovo contesto stack sono nel segmento di registro
; DS, il quale deve essere settato propriamente per accedere alle locazioni di
; salvataggio e ripristino.
;
; Lo stato salvato consiste nei registri BP, SI, DI e i FLAGS
;-------------------------------------------------------------------------------
_ctxsw      proc  far
      push  bp
      mov   bp,sp             ; frame pointer
      pushf                   ; condizioni di interrupt salvate nei flag
      cli                     ; disabilita gli interrupt per essere sicuri
      push  si
      push  di                ; preserva i registri nel vecchio stack
      mov   bx,[bp+6]         ; bx = offest del vecchio stack
      mov   es,[bp+8]         ; es = segmento del vecchio stack
      mov   es:[bx],sp        ; salva vecchio stack pointer
      mov   es:[bx+2],ss      ; salva vecchio stack segment
      mov   bx,[bp+10]        ; bx = offset del nuovo stack
      mov   es,[bp+12]        ; es = segmento del nuovo stack
      mov   sp,es:[bx]        ; sp = nuovo stack pointer
      mov   ss,es:[bx+2]      ; ss = nuovo stack segment
      pop   si
      pop   di
      popf                     ; ristabilisce lo stato di interrupt
      pop   bp
      ret
_ctxsw      endp

      endps

      end
```

Ecco lo stato dello stack prima del cambio di contesto:

Il codice in *ctxsw* svela come risolvere il dilemma causato dal cercare di salvare i registri mentre un processo li sta ancora usando. Pensiamo ad un processo in esecuzione che ha chiamato *resched* che, al suo volgere, chiama *ctxsw*. Invece di provare a salvare i registri esplicitamente

durante l'esecuzione, ctxsw cattura il valore del puntatore allo stack precisamente quando i registri (che includono il puntatore alle istruzione e i FLAGS) sono già sullo stack come risultato del codice in *ctxsw*. Ciò congela lo stack del processo come se fosse in mezzo all'esecuzione di una normale procedura. Poi *ctxsw* restaura il puntatore allo stack di un altro processo che era stato congelato. *Ctxsw* ritorna i registri e riprende normalmente l'esecuzione dell'altro processo.

E' interessante notare che tutti i processi chiamano *resched* per eseguire il cambio di contesto, e *resched* chiama *ctxsw*; così tutti i processi sospesi saranno ripresi nello stesso punto - proprio dopo la chiamata a *ctxsw* in *resched*. Ogni processo ha nel proprio stack chiamate a procedure, tuttavia, il ritorno da *resched* li porterà in varie direzioni. Notiamo anche che, se i due puntatori passati a *ctxsw* sono uguali - per esempio, se un processo potesse eseguire un cambio di contesto con se stsso - *ctxsw* ritornerà semplicemente al chiamante senza cambiamenti.

Costruire tutte le procedure di ritorno al loro chiamante è un ingrediente chiave nel tenere la progettazione del sistema chiaro. Sarebbe stato impossibile a meno che lo scheduler ritornasse al suo chiamante. Così, sia *resched* che *ctxsw* sono stato disegnati per comportarsi proprio come ogni altra procedura - essi eventualmente ritornano. Naturalmente, ci può essere un considerevole ritardo prima che la chiamata a *resched* ritorni, perché la CPU potrebbe eseguire altri processi arbitrariamente lunghi prima di far ripartire il processo chiamante (dipendente dalla priorità).

4.4 Il processo NULL

Resched alterna il processore solo tra i processi pronti e il corrente; esso non crea nuovi processi. Inoltre assume che al minimo un processo sia disponibile e non si preoccupa di verificare se la lista dei pronti sia vuota. C'è una forte conseguenza:

> *Resched può cambiare contesto solo da un processo a un altro, perciò, al minimo, un processo deve sempre rimanere pronto ad eseguire.*

Per assicurarsi che almeno un processo esista, PC-Xinu crea un processo extra chiamato il processo *null* quando inizializza il sistema. In processo *null* ha come identificatore il valore zero e priorità zero; il suo codice, che sarà mostrato nel capitolo 13, e consiste di un ciclo infinito. Poiché i processi utenti devono avere priorità maggiore di zero, lo scheduler sceglie il suddetto processo quando nessun processo utente rimane pronto ad eseguire.

4.5 Rendere un processo ready

Quando resched necessitava di muovere il processo corrente sulla lista dei pronti, esso manipolava la lista direttamente. Rendere un processo eleggibile per il servizio della CPU avviene così frequentemente che abbiamo inventato una procedura che faccia proprio questo. Essa si chiama ready:

```
/* ready.c - ready */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>

/*------------------------------------------------------------------------
 *  ready  --  rende un processo eleggibile per il servizio della CPU
```

```
 *------------------------------------------------------------------------
 */
int   ready (pid)
      int   pid;              /* identificatore del processo da rendere ready*/
{
      register struct   pentry      *pptr;

      if ( isbadpid(pid) )
            return(SYSERR);
      pptr = &proctab[pid];
      pptr->pstate = PRREADY;
      insert(pid,rdyhead,pptr->pprio);
      return(OK);
}
```

Generalmente, una procedura che chiama *ready* necessita anche una chiamata a *resched* dopo aver piazzato il processo sulla lista dei pronti; ciò assicura che la CPU sta eseguendo il processo pronto con più alta priorità. Quando il chiamante deve muovere parecchi processi da qualche altra lista alla lista dei pronti, la schedulazione dopo ogni chiamata a *ready* può sciupare tempo. La risposta consiste nel sospendere temporaneamente la chiamata a *resched* e chiamare *ready* parecchie volte. Poi, dopo che tutti i processi sono stati mossi e la manipolazione della lista è stata completata, una chiamata a *resched* reintegra la politica di esecuzione del processo con priorità maggiore. Vedremo un esempio di schedulazione rinviata nel capitolo 6.