CAPITOLO 5 - ULTERIORE AMMINISTRAZ1ONE DEI PROCESSI

II capitolo 4 ha trattato il cambio di contesto e mostrato come i processi si muovono dallo stato *ready* a quello *current.* Questo capitolo mostra invece il modo in cui i nuovi processi vengono creati e come, eventualmente, escono di scena. Inoltre introduce il nuovo stato *suspended* ed esplora routine che muovono i processi negli stati current, ready e suspended.

5.1 Sospensione e ripresa di un processo

Avere un modo per arrestare temporaneamente un processo dall'esecuzione e poi farlo ripartire dimostra quanto sia completamente utile. Un processo fermato diremo che e stato posto nello stato di *«suspended unimation».* La *suspended animation* può essere usata, per esempio, quando un processo vuole attendere una delle parecchie condizioni di ripresa senza sapere quale di queste si verificherà per prima. Il primo passo nell'implementazione della *suspended animation* consiste nel definire operazioni che saranno usate per sospendere i processi. In questo caso, la scelta è ovvia perché abbiamo bisogno soltanto di due cose: *suspend,* per arrestare un processo e *resume* per reinizializzarlo. Poiché i processi sospesi non sono in grado di usare la CPU, necessitiamo un nuovo stato che distingua tali processi da quelli *ready* o *current.* Chiameremo il nuovo stato *suspended.* La figura 5.1 ricapitola le azioni che eseguono *suspend* e *resume,* mostrando il modo in cui i processi si muovono tra gli stati *ready, current* e *suspended.*

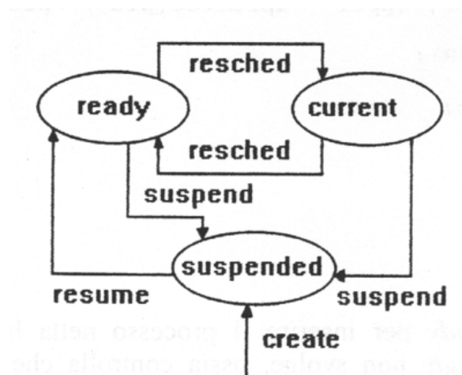

Figura 5.1 - Transizioni tra gli stati current, ready e suspended.

I dettagli di *suspend* e *resume* sono ovvi. *Suspend* richiede un argomento che specifichi il processo da sospendere. Esso deve anche verificare che il processo da sospendere sia *ready o current.* Anche *resume* richiede un argomento che specifichi il processo da far ripartire; esso deve inoltre verificare che il processo specificato sia nello stato *suspended.*

La ripresa di un processo e lineare. *Resume* deve soltanto riportare il processo nella lista dei pronti e cambiare il suo stato. La sospensione non e molto più complessa. Sospendere un processo implica cambiare lo stato attuale del rispettivo elemento della tabella dei processi e rimuoverlo dalla lista dei pronti; in questa maniera *resched* non lo selezionerà. Una procedura in esecuzione può sospendere se stessa passando a *suspend* il suo identificatore di processo:

suspend(getpid ( ));

Sospendere il processo corrente comporta metterlo nello stato sospeso e poi effettuare la schedulazione per permettere ad un altro processo di eseguire.

5.1.1 Implementazione di resume

La procedura *resume* rimette un processo sospeso nello stato pronto e quindi in possibilità di sfruttare il processore. Il codice relativo è contenuto nel file *resume.c.*

```
/ resume.c - resume */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
/* ------------------------------------------------------------------------------
 * resume -- passa un processo sospeso nello stato ready; ritorna la priorità
 * ------------------------------------------------------------------------------
 */
SYSCALL resume(pid)
int pid
{
      int ps;
      struct pentry *pptr;          /* puntatore all'elemento della tabella dei
processi */
      int prio;                     /* priorità da ritornare */

      disable(ps);
      if (isbadpid(pid) || (pptr = &proctab[pid]->pstate != PRSUSP) {
            restore(ps);
            return(SYSERR);
      }
      prio = pptr->pprio;
      ready (pid);
      resched();
      restore(ps);
      return(prio);
}
```

Sebbene *resume* chiami *ready* per inserire il processo nella lista dei pronti, esso esegue un importante controllo che *ready* non svolge, ossia controlla che L'argomento specificato sia un processo valido e che ciò che e riferito sia sospeso. Esso disabilita pure gli interrupt prima di chiamare *ready.* Queste azioni rendono *resume* chiamabile da un programma utente.

A volte, per il processo chiamante *resume,* risulta utile conoscere la priorità del processo da riprendere per poter restituire il suo valore. Dobbiamo prenderci cura di catturare la priorità prima della chiamata a *ready* perchè il processo ripreso potrebbe ricominciare ad eseguire quando *ready* chiama *re.vched* (prima che completi il processo che esegue *re.~ume).* L'intervallo di tempo esistente tra la chiamata a *ready* e l'istruzione seguente può essere arbitrariamente lungo perché un numero arbitrario di processi possono andare in esecuzione prima che il processo chiamante sia rischedulato. Per essere sicuri che il valore ritornato rifletta la priorità del processo ripreso, *resume* effettua una copia di tale valore nella variabile locale *può,* prima di chiamare *ready,* e ritorna tale valore.

5.1.2 I valori di ritorno SYSERR e OK.

Resume.c include i file *kernel*.h assieme a *conf*.h e *proc*.h. *Kernel*.h, mostrato più avanti in questo capitolo, definisce molte costanti usate, in pratica, dappertutto (come ad esempio gli interi SYSERR e OK). Noi abbiamo già incontrato SYSERR nel capitolo 2 e OK nel capitolo 3. Convenzionalmente, le procedure di PC-Xinu ritornano il valore SYSERR per indicare che l'argomento inserito era inaccettabile o che qualcos'altro ha impedito il completamento con successo delle operazioni che essi cercavano di portare a buon fine. Similarmente, routine come *ready* che non restituiscono valori al chiamante ritornano l'intero OK per indicare il completamento con successo.

5.2 Chiamate di sistema

Le precauzioni prese da *resume* per verificare che un operazione sia legale la rendono una routine di scopo generale che può essere invocata da ogni processo in qualsiasi momento. Esso e il nostro primo esempio di ciò che generalmente chiamiamo *system call* (chiamate di sistema). Le chiamate di sistema si collocano tra un ingenuo programma utente e la restante parte del sistema operativo; esse devono proteggere il sistema interno da usi illegali. Per il programmatore le *system call* definiscono 1'aspetto esteriore del sistema operativo provvedendo un'interfaccia attraverso il quale l'utente accede ai servizi di sistema.

Quando un processo esegue una *system call* come *resume* e le procedure che *resume* chiama, vengono modificate la tabella dei processi e altre strutture dati del sistema come la struttura *q*. Esiste una sola tabella dei processi nel sistema, condivisa da tutti i processi. Come può un processo essere sicuro che nessun altro interferirà cercando di cambiare la tabella dei processi nello stesso tempo? Per prima cosa non deve chiamare *resched* perché significherebbe un cambio di contesto ad un altro processo che modificherebbe le tabelle di sistema. Vedremo che ciò può avvenire ugualmente come risultato di un interrupt da parte di una device, perché le routine di interrupt a volte chiamano *resched.* Per prevenire gli interrupt, *resume* invoca la procedura *disable* per disabilitare gli interrupt del processore. *Disable* registra il corrente stato di interrupt (l'attuale contenuto del registro FLAGS), disabilita gli interrupt del processore e poi ritorna lo stato registrato. Proprio prima di lasciare *resume,* il processo chiama la procedura *restore* per riabilitare lo stato di interrupt al suo valore originale. *Resume* non può abilitare gli interrupt cosi semplicemente prima di ritornare al suo chiamante; infatti e probabile che il chiamante fosse in esecuzione con gli interrupt disabilitati. Quindi. prima di ritornare, *resume* ripristina lo stato di interrupt al valore originale.

5.2.1 Disable e restore

Disable viene espansa dal preprocessore C in una chiamata a procedura assembler *sys_disable* che ha il compito di salvare i1 valore corrente del registro FLAGS, disabilitare gli interrupt e ritornare il valore del registro FLAGS appena memorizzato. L'accesso a questo registro e la disabilitazione degli interrupt sono operazioni di basso livello che devono essere attuate in linguaggio assembler. Similarmente, *restore* i in realtà una procedura assembler *sys_restor* che deposita il suo argomento ne1 registro FI.AGS. *Disable* e *restore* sono generalmente usate in coppia, delimitando la sezione critica del codice che non deve essere interrotto da altre attività di sistema. ll codice di *sys_disable* e *sys_re*store è contenuto nel file *eidi.asm.* Questo file contiene anche altre funzioni di utilità a basso livello: *sys_enabl* per abilitare gli interrupt, *sys_wait* per sospendere il

processore in attesa di un interrupt e *sys_hlt* per ritornare il controllo al sistema operativo ospite.

```
; eidi.asm - _sys_disabl, _sys_enabl, _sys_restor, _sys_wait, _sys_hlt

      include ..\h\dos.asm            ; macro di segmento
      dseg
; null data segment
      endds

      pseg

        public  _sys_disabl,_sys_restor,_sys_enabl,_sys_wait,_sys_hlt

;--------------------------------------------------------------------------------
; _sys_disabl  --  ritorna lo stato di interrupt e disabilita gli interrupt
;--------------------------------------------------------------------------------
; int sys_disabl()
_sys_disabl proc  far
      pushf             ; mete la word flag sullo stack
      cli               ; disabilita gli interrupt!
      pop   ax          ; deposita la word flag in un registro di ritorno
      ret
_sys_disabl endp

;--------------------------------------------------------------------------------
; _sys_restor  --  ristabilisce lo stato di interrupt
;--------------------------------------------------------------------------------
; void sys_restor(ps)
; int ps;
_sys_restor proc  far
      push    bp
      mov   bp,sp       ;
      push  [bp+6]
      popf              ; restituisce la word flag
      pop     bp
      ret
_sys_restor endp

;--------------------------------------------------------------------------------
; _sys_enabl  --  abilita gli interupt incondizionatamente
;--------------------------------------------------------------------------------
; void sys_enabl()
_sys_enabl  proc  far
      sti               ; abilita gli interrupt
      ret
_sys_enabl  endp

;--------------------------------------------------------------------------------
; _sys_wait  --  atende interrupt
;--------------------------------------------------------------------------------
; void sys_wait()
_sys_wait   proc  far
      pushf
      sti               ; gli interrupt devono essere abilitati qui
      hlt
      popf
      ret
```

```
_sys_wait   endp
;------------------------------------------------------------------------------
; _sys_hlt  --  arresta il programma corrente e ritorna all'ospite
;------------------------------------------------------------------------------
; void sys_hlt()
_sys_hlt    proc  far
      mov   ah,4ch            ; funzione di fine
      xor   al,al             ; codice di ritorno
      int     21h             ; chiamata di funzione MS-DOS
      ret
_sys_hlt    endp

      endps

      end
```

5.2.2 Implementazione di suspend

Sospendere un processo non è una operazione molto più complessa di una *resume*. Il file *suspend.c* contiene il codice inerente. Per prima cosa *suspend* controlla l'argomento *pid* passato affinché esso specifichi un processo valido, cioè in stato di *ready* o in esecuzione. Se il processo da sospendere e nello stato di pronto, esso deve essere rimosso dalla lista e messo nello stato *suspended*.

```
/* suspend.c - suspend */
/* 8086 version */

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

/*------------------------------------------------------------------------------
 *  suspend  --  sospende un processo, metendolo in ibernazione
 *------------------------------------------------------------------------------
 */
SYSCALL     suspend(pid)
      int   pid;              /* id del processo da sospendere */
{
      struct pentry *pptr;    /*   puntatore   all'elemento   della   tabella   dei
processi */
      int   ps;
      int   prio;             /* priorità ritornata */

      disable(ps);
      if (isbadpid(pid) || pid==NULLPROC ||
       ((pptr= &proctab[pid])->pstate!=PRCURR && pptr->pstate!=PRREADY)) {
            restore(ps);
            return(SYSERR);
      }
      if (pptr->pstate == PRREADY) {
            dequeue(pid);
            pptr->pstate = PRSUSP;
      } else {
            pptr->pstate = PRSUSP;
            resched();
      }
```

```
	prio = pptr->pprio;
	restore(ps);
	return(prio);
}
```

5.2.3 Sospendere il processo corrente

II codice che sospende il processo correntemente in esecuzione mette in luce due punti interessanti. Primo, il processo in esecuzione verrà fermato, al minimo temporaneamente, e bisognerà alternare un altro processo. Per fare ciò si marca semplicemente il processo usando la costante PRSUSP e si chiama *resched.* Se ricordi, *resched* controlla lo stato del processo per determinarne la disposizione. In questo caso cambierà contesto senza muovere il processo nella lista dei pronti. Secondo, *suspend,* come *resume,* ritorna la priorità del processo sospeso al suo chiamante. Tuttavia, *suspend* registra la priorità dopo aver sospeso il processo. Quando un processo sospende un altro, il codice ha perfetto senso perché niente può cambiare la priorità mentre *suspend* esegue con gli interrupt disabilitati. Invece, quando un processo sospende se stesso, chiama *resched* permettendo ad altri processi di eseguire. Essi potranno cosi cambiare la priorità del processo. Quando *suspend* riprende eventualmente l'esecuzione, riporterà la priorità.

Puoi aver notato che *suspend* e *resume* non mantengono una lista concatenata di processi sospesi come *ready.* La ragione e semplice. I processi pronti sono tenuti su una lista ordinata per velocizzare la ricerca del processo con maggiore priorità durante la schedulazione. Poiché il sistema non cerca mai tra i processi sospesi cercandone uno da riprendere, l'insieme dei processi sospesi non necessita di essere tenuto in una lista.

5.3 La conclusione di un processo

Suspend congela processi, ma li lascia nel sistema per poter essere successivamente ripresi. Un'altra system call denominata *kill* interrompe immediatamente un processo e lo rimuove dal sistema completamente. Una volta rimosso non potrà essere ricominciato perché *kill* sradica l'intero record liberando l'elemento della tabella dei processi.

Le azioni intraprese da *kill* dipendono dallo stato del processo. Prima di scrivere il codice, il progettista ha dovuto prendere in considerazione tutti i possibili stati e determinarne le conseguenze di terminazione per ogni caso. Per esempio, i processi *ready*, *sleeping* o *waiting* sono tenuti in liste concatenate nella struttura *q*, perciò *kill* dovrà rimuoverli da esse. Se il processo e in attesa su un semaforo, *kill* dovrà aggiustare anche il contatore del semaforo. Non tutti questi casi hanno senso completo fin quando non avrai maggiore conoscenza dello stato dei processi.

Il codice di *kill* appare nel file kill.c mostrato sotto. Consideriamo il suo operato su un processo *ready*. *Kill* controlla il suo argomento, il pid, per assicurarsi che corrisponda a un valido, attivo processo verificando che sia nella sua corretta portata e che 1'elemento della tabella dei processi non sia libera. Esso poi decrementa *numproc*, la variabile globale che registra il numero di processi utente attivi. Successivamente *kill* chiama la procedura *freestk* per liberare la memoria che il processo usa per lo stack. *Freestk* preleva il processo dalla lista dei pronti con la procedura *dequeue* e libera il relativo elemento della tabella dei processi assegnando al campo di stato il valore PRFREE. Non comparendo più nella lista esso non riotterrà più il controllo della CPU.

Adesso consideriamo cosa succede quando *kill* vuole terminare il processo correntemente in esecuzione. Come prima, esso convalida il suo argomento e decrementa il numero di processi attivi. Se il processo e 1'ultimo tra quelli utente, *numproc* va a zero e *kill* chiama la procedura xdone

mostrata successivamente. Dopo aver liberato i1 campo di stato la chiamata a *resched* passerà il controllo ad un altro processo pronto.

```
/* kill.c - kill */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <sem.h>
#include <dos.h>

/*------------------------------------------------------------------------------
 * kill  --  uccide un processo e lo rimuova dal sistema
 *------------------------------------------------------------------------------
 */
SYSCALL kill(pid)
int   pid;              /* processo da uccidere */
{
      struct      pentry      *pptr;
      int   ps;
      int   die();

      disable(ps);
      if (isbadpid(pid) || (pptr= &proctab[pid])->pstate==PRFREE) {
            restore(ps);
            return(SYSERR);
      }
      if (--numproc == 0)
            xdone();
      freestk(pptr->pbase, pptr->plen);
      switch (pptr->pstate) {

      case PRCURR:      pptr->pstate = PRFREE;  /* suicidio */
                        resched();

      case PRWAIT:      semaph[pptr->psem].semcnt++;

      case PRSLLE:
      case PRREADY:     dequeue(pid);

      default:          pptr->pstate = PRFREE;
      }
      restore(ps);
      return(OK);
}


/* xdone.c - xdone */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <io.h>
#include <disk.h>
#include <tty.h>

extern char *memptr;
```

```
extern int run;

/*------------------------------------------------------------------------
 *  xdone  --  stampa messaggio di fine sistema e termina PC-Xinu
 *------------------------------------------------------------------------
 */
int xdone()
{
      int   kprintf();
      int   ps, i;

#ifdef Ndsk
      for ( i=0; i<Ndsk; i++ )
            control(dstab[i].dnum,DSKSYNC);     /* sincronizza il disco */
#endif
      sleep(1);                                 /* pausa per l'output */
      disable(ps);
      maprestore();
      restore(ps);
      kprintf("\n\n-- system halt --\n\n");
      if (numproc==0)
            kprintf("All user processes have completed\n");
      else
        kprintf("terminated with %d process%s active\n",
                  numproc, ((numproc==1) ? "" : "es"));
      kprintf("Returning to DOS . . .\n\n");
      halt();
```

5.4 Dichiarazioni del kernel

Il file *kernel.h* definisce macro tra le quali *disable* e *restore* menzionate sopra e altre variabili e costanti simboliche usate in tutto PC-Xinu. Sebbene non tutti i nomi che appaiono hanno ancora senso, essi lo avranno alla fine del capitolo.

```
/* kernel.h - isodd */
/* 8088 PC-Xinu version - for IBM PC and Clones */

/* Costanti simboliche usate ovunque in Xinu */

typedef int             Bool;       /* tipo booleano */
typedef unsigned int    word;       /* tipo word */
typedef unsigned short  para;       /* tipo paragrafo */

#define     FALSE       0           /* costanti booleane */
#define     TRUE        1
#define     EMPTY       (-1)        /* un gpq illegale */
#define     NULL        0           /* puntatore nullo per liste concatenate */
#define     LSYSCALL    long        /* System call che ritornano un long */
#define     SYSCALL     int         /* dichiarazione di system call */
#define     LOCAL       static      /* dichiarazione di procedura locale */
#define     INTPROC     int         /* procedura di interrupt */
#define     PROCESS     int         /* dichiarazione di processo */
#define     WORD        word        /* word di 16 bit */
#define     MININT      0100000     /* minimo intero (-32768) */
#define     MAXINT      0077777     /* massimo intero (+32767) */
```

```
#define     MINSTK      0x400       /* minima dimensione dello stack-processo */
#define     NULLSTK     0x400       /* stack del processo 0 */
#define     OK          1           /* ritornato quando la system call è ok */
#define     SYSERR      -1          /* ritornato quando la sys. Call fallisce */

#define     EOF         -2          /* End-of-file (usu. from read)     */
#define     TIMEOUT     -3L         /* time out (usu. recvtime) */
#define     TMSGINT     -4L         /* "intr" su pressione tasto tastiera */
                                    /* (usu. defined as ^B)       */
#define     TMSGKILL    -5L         /* tasto per "uccidere processo" */
                                    /* (usu. defined as ^C) */

#define     NULLCH      '\0'        /* carattere null */
#define     NULLSTR     ""          /* puntatore a stringa vuota */
#define     COMMAND     int         /* dischiarazione dei comandi Shell */
#define     BUILTIN     int         /* Shell builtin "" */

#define     LOWBYTE     0377        /* maschera per gli 8 bit bassi */
#define     HIBYTE      0177400     /* maschera per gli 8 bit alti dei 16 */
#define     LOW16       0177777     /* maschera per i 16 bit bassi */

#define     MAGIC       0125252     /* insolito valore per il top dello stack */


/* inizializzazione di costanti */

#define     INITARGC    1           /* argc iniziale del processo */
#define     INITSTK     0x1000      /* stack iniziale del processo */
#define     INITPRIO    20          /* priorità del processo iniziale */
#define     INITNAME    "xmain"     /* nome di processo iniziale */
#define     INITRET     userret     /* indirizzo di ritorno dei processi */
#define     INITREG     0           /* contenuto iniziale di registro */
#define     QUANTUM     1           /* tick di clock prima del preemption */
#define     INITARGS    1,0         /* contatore/argomento iniziale */

/* funzioni utility miscellanee */

#define     isodd(x)    (01&(int)(x))
#define     disable(x)  (x)=sys_disabl()  /* salva lostato di interrupt */
#define     restore(x)  sys_restor(x)     /* ripristina lo stato di interrupt */
#define     enable()    sys_enabl()       /* abilita interrupt */
#define     pause()     sys_wait()        /* attesa della macchina per interr.*/
#define     halt()      sys_hlt()         /* arresta PC-Xinu */
#define xdisable(x)     (x)=sys_xdisabl() /* salva lo stato di int & dosflag */
#define xrestore(x)     sys_xrestor(x)    /* rende lo stato di int & dosflag */
#define min(a,b)        (((a) < (b)) ? (a) : (b))
#define max(a,b)        (((a) > (b)) ? (a) : (b))

/* furzioni e variabili specifiche di sistema */

extern      int   sys_disabl();           /* ritorna int flags & disable */
extern      void  sys_restor();           /* rende il registro dei flag */
extern      void  sys_enabl();            /* abilita interrupt */
extern      void  sys_wait();             /* attesa di un interrupt */
extern      void  sys_hlt();              /* arresta il processore */
extern      int   sys_xdisabl();          /* ritorna gli interrupt a MS-DOS */
extern      void  sys_restor();           /* Interrupt back to Xinu */
extern      word  sys_cs;                 /* valore del reg. di code segment */
```

```
/* variabili per la gestione dei processi */

extern      int   rdyhead, rdytail;
extern      int   preempt;
extern      long receive();
extern      long recvclr();
extern      long recvtim();
extern      send(int, long);
extern      sendf(int, long);
extern      sendn(int, long);
```

5.5 Creazione di un processo

La system call *create* crea un nuovo, indipendente processo. L'idea e di mettere giù una esatta immagine del processo come se fosse stato fermato dall'esecuzione; in tal maniera ctxsw può considerarlo. *Create* trova uno slot libero nella tabella dei processi, alloca spazio per lo stack del nuovo processo e riempie il relativo elemento della tabella dei processi.

Uno sguardo al codice del file *create.c* mette in risalto maggiori dettagli. La procedura *newpid* cerca uno slot libero nella tabella dei processi e ritorna il valore SYSERR se non lo trova. *Create* usa la *macro roundew* per arrotondare la specificata dimensione dello stack alla successiva word pari e chiama *getmem* per allocare lo spazio necessario (il capitolo 8 illustra entrambe le routine).

Ci riferiamo allo stack del processo iniziale come una pseudo-call poiché create inserisce valori con cura su di esso per simulare una chiamata di procedura. In C la pseudo-call consiste di argomenti e indirizzi di ritorno. Quando e partito, il nuovo processo inizia eseguendo il codice per la procedura designata, obbedendo alle normali convenzioni di chiamata per accedere agli argomenti e allocando variabili locali. In breve, esso si comporta come se fosse stato chiamato da un'altra procedura.

Come sceglie il progettista il valore dell'indirizzo di ritorno da usare nella pseudo-call? Fortunatamente esiste una linea-guida per ciò che accade quando un processo ritorna dalla sua procedura di inizio: esso dovrebbe terminare. *Create* crea l'indirizzo di ritorno nella pseudo-call con quello di *userret.* Se il processo non permette il ritorno dalla procedura iniziale il controllo passa a *userret.* Questa procedura termina il processo chiamante con *kill.*

Create riempie anche l'elemento della tabella dei processi. Sapendo che *ctxsw* alterna tra processi prelevando il contenuto dei registri dal suo stack (puntato dal campo *pregs), crea (e* riempie i valori sullo stack e setta l'elemento *pregs* per puntare al "top dello stack". I valori di SI, DI e HP sono immateriali mentre il valore del registro FLAGS e importante. Poiché il processo dovrebbe iniziare L'esecuzione con gli interrupt abilitati, il registro FLAGS dovrebbe avere il bit di interrupt settato. *Create* pone lo stato del processo creato al valore di PRSUSP, lasciandolo sospeso. Finalmente ritorna l'identificatore di processo che deve essere passato a *resume* per avviare all'esecuzione il nuovo processo.

Molti dettagli della inizializzazione di un processo dipendono dall'ambiente di esecuzione C - non è semplice avviare un processo senza affrontare tali dettagli. Per esempio *create* inserisce gli argomenti sullo stack del processo in modo tale che il primo argomento sia vicino alla cima. Il codice che inserisce tali argomenti e difficile da comprendere perché *create* li copia direttamente dal suo stack di esecuzione a quello che ha allocato per il nuovo processo. Per effettuare ciò, esso trova l'indirizzo degli argomenti sul proprio stack e li muove usando puntatori aritmetici. Questo, ovviamente, e un trucco dipendente dalla macchina (e dal compilatore).

```
/* create.c - create, newpid */

#include <conf.h>
#include <kernel.h>
#include <io.h>
#include <proc.h>
#include <dos.h>
#include <mem.h>

typedef int (*fptr)();

#define     INITF 0x0200      /* regisro flag inziale */
/* setta il flag di interrupt, clear direction e trap */

extern      int   INITRET();  /* locazione da restituire al termine */

/*------------------------------------------------------------------------------
 *  create  --  crea un process oda avviare eseguendo una procedura
 *------------------------------------------------------------------------------
 */
SYSCALL create(procaddr,ssize,priority,namep,nargs,args)
int (*procaddr)();            /* indirizzo di procedura */
word ssize;                   /* dimensione stack in word */
short priority;               /* priorità di processo > 0 */
char *namep;                  /* nome (per il debugging) */
int nargs;                    /* numero di argomenti che seguono */
int args;                     /* argomenti (trattati come un array) */
{
      int   pid;              /* memorizza il nuovo identificatore di processo*/
      struct pentry *pptr;    /* puntatore all'elem. Della tabella dei proc. */
      int i;                  /* variabile di ciclo */
      int *a;                 /* puntatore alla lista degli argomenti */
      char *saddr;            /* indirizzo di partenza dello stack */
      int *sp;                /* puntatore allo stack */

      disable(ps);
      ssize = roundp(ssize);
      if (ssize < MINSTK || priority < 1 ||
            (pid=newpid()) == SYSER || ((saddr=getstk(ssize)) == NULL )) {
                  restore(ps);
                  return(SYSERR);
      }
      numproc++;
      pptr = &proctab[pid];
      pptr->pstate = PRSUSP;
      for (i=0 ; i<PNMLEN ; i++)
            pptr->pname[i] = (*namep ? *namep++ : ' ');
      pptr->pname[PNMLEN]='\0';
      pptr->pprio = priority;
      pptr->phasmsg = FALSE;        /* nessun messagio */
      pptr->pbase = saddr;
      pptr->plen = ssize;
      sp = (int *) (saddr+ssize);   /* simula il puntatore allo stack */
      sp -= 4;                      /* lascia un piccolo spazio */
      pptr->pargs = nargs;
      a = (&args) + nargs;          /* punta all'ultimo argomento passato */
      for ( ; nargs > 0; nargs--)   /* dipendente dalla macchina; copia */
            *(--sp) = *(--a);       /* argoment sullo stack creato */
```

```
        *(--sp) = (int)INITRET;         /* push dell'indirizzo di ritorno */
        *(--sp) = (int)procaddr;        /* simula un cambio di contesto */
        --sp;                           /* una word per bp */
        *(--sp) = INITF;                /* valore del registo FLAGS */
        sp -=2;                         /* 2 word per  SI e DI */
        pptr->pregs = (char *)sp;       /* salva per il cambio di contesto */
        restore(ps);
        return(pid);
}

/*------------------------------------------------------------------------
 *  newpid  --  ottiene un nuovo (libero) identificatore di processo
 *------------------------------------------------------------------------
 */
LOCAL newpid()
{
        int pid;                    /* identificatore di processo da ritornare */
        int   i;

        for (i=0 ; i<NPROC ; i++) {    /* controlla tutti gli NPROC slot */
              if ( (pid=nextproc--) <= 0)
                    nextproc = NPROC-1;
              if (proctab[pid].pstate == PRFREE)
                    return(pid);
        }
        return(SYSERR);
}

/* userret.c - userret */

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

/*------------------------------------------------------------------------
 *  userret  --  entered when a process exits by return
 *------------------------------------------------------------------------
 */
userret()
{
        int   pid;

        kill(pid=getpid());
        kprintf("Fatal system error - unable to kill process %d",pid);
}
```

5.6 Procedura di utilità

Tre addizionali system call aiutano a gestire i processi: *getpid, getprio* e *chprio*. *Getpid* permette di ottenere l'identificatore di un processo. *Userret* mostra una ragione per cui una procedura necessita di sapere l'identificatore del processo che la sta eseguendo. *Getprio* invece dà la possibilità di conoscere la priorità di schedulazione. Un'altra chiamata di sistema utile e *chprio* la quale abilita un processo a cambiare la propria priorità. L'implementazione di queste tre routine è estremamente semplice.

```
/* getprio.c - getprio */

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

/*------------------------------------------------------------------------
 *  getprio  --  ritorna la priorità di schedulazione di un dato processo
 *------------------------------------------------------------------------
 */
SYSCALL getprio(pid)
	int	pid;
{
	struct	pentry	*pptr;
	int	ps;

	disable(ps);
	if (isbadpid(pid) || (pptr = &proctab[pid])->pstate == PRFREE) {
		restore(ps);
		return(SYSERR);
	}
	restore(ps);
	return(pptr->pprio);
}
```

Dopo aver controllato il suo argomento, *getprio* estrae la priorità di schedulazione per il processo specificato da1 relativo elemento della tabella dei processi e ritorna la priorità al suo chiamante.

```
/* getpid.c - getpid */

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

/*------------------------------------------------------------------------
 *  getpid  --  ottiene l'identificatore del processo in esecuzione
 *------------------------------------------------------------------------
 */
SYSCALL getpid()
{
	return(currpid);
}
```

Può sembrare che la procedura *getpid* sia inutile perché ritorna il valore della variabile currpid che potrebbe essere ottenuta direttamente dal processo senza ulteriore sovraccarico. Perché i processi non hanno accesso a *currpid*? Se PC-Xinu venisse trasportato su una macchina nella quale i processi utente non possono accedere allo spazio di indirizzamento occupato dal sistema. non sarebbe possibile ottenere i1 valore di *currpid* direttamente.

```
/* chprio.c - chprio */

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

/*------------------------------------------------------------------------
 *  chprio  --  cambia la priorità di schedulazione di un processo
 *------------------------------------------------------------------------
 */
SYSCALL chprio(pid,newprio)
	int	pid;
	int	newprio;		/* newprio > 0 */
{
	int	oldprio;
	struct	pentry	*pptr;
	  int ps;

	disable(ps);
	if (isbadpid(pid) || newprio<=0 ||
	    (pptr = &proctab[pid])->pstate == PRFREE) {
	      restore(ps);
	      return(SYSERR);
	}
	oldprio = pptr->pprio;
	pptr->pprio = newprio;
	restore(ps);
	return(oldprio);
}
```

Questa implementazione di *chprio* sembra fare esattamente ciò che gli è stato richiesto. Essa controlla che il processo specificato esista prima di cambiare il campo priorità nel suo elemento della tabella dei processi. Tuttavia contiene un serio difetto.