CREAZIONE PROCESSI IN UNIX

STRUTTURE DATI PER PROCESSI

Un processo puo' essere in escuzione in *2 modi: kernel e utente.*
Un processo ha almeno 3 regioni: codice, dati e stack
Lo stack è allocato dinamicamente.
Unix usa 2 tipi di stack, user e kernel, a seconda del modo di esecuzione.Il meccanismo che opera il passaggio da user a kernel viene detto **trap** (o software interrupt).
Il kernel possiede una Tabella dei Processi contenente informazioni su ciascun processo in esecuzione.
Il **Contesto del processo**: è costituito dal contenuto dello spazio utente (le tre regioni), dei registri hardware, e dalle strutture dati del kernel relativi a quel processo.

MECCANISMO DI BIFORCAZIONE proc_id=**fork()**

La system call fork() duplica un processo. Dopo la fork, se questa ha avuto successo, esistono due processi: il genitore e il figlio.
I due processi avranno identificatori di processo diversi.
Poiche' il program counter e' lo stesso, entrambi i processi ritengono di aver eseguito la funzione fork. Entrambi ricevono un valore di ritorno, ma diverso:

proc_id > 0 per processo padre
== 0 per il figlio
< 0 per il padre (solo in caso di errore della fork)

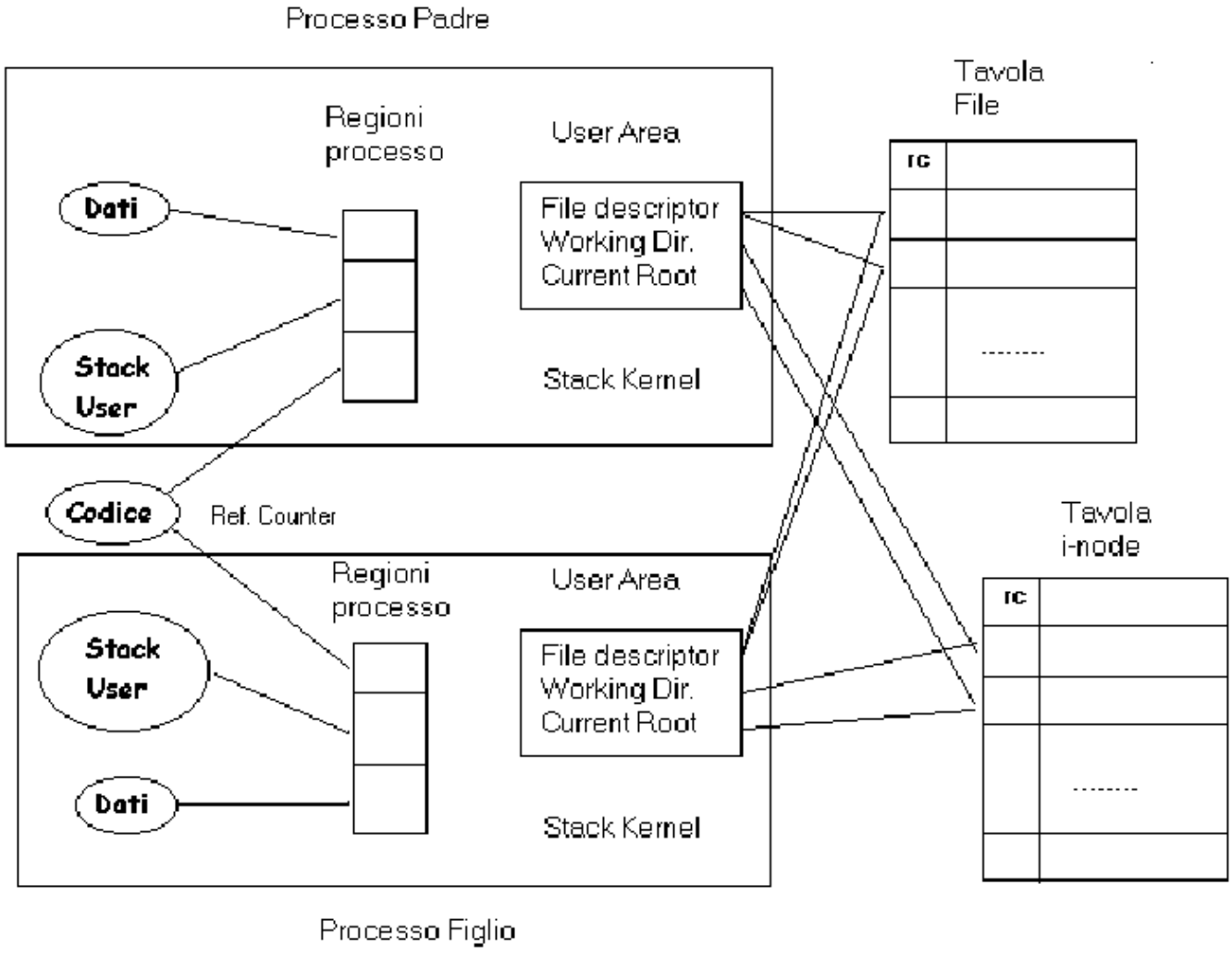
```
pid_t pid;
pid=fork();  // creo un nuovo processo
if(pid<0)    exit(1);  // errore, duplicazione non eseguita
else  {
      if(pid==0) {    .....   sono il processo figlio      }
      else       {    .....   sono il processo padre     }
}
```

RELAZIONE PADRE FIGLIO

Il figlio condivide il codice con il genitore, mentre la memoria, i registri e le informazioni sui files (tabelle) vengono duplicate.

Quindi, dopo la fork, ogni processo può modificare le variabili contenute nel proprio spazio utente senza alterare le analoghe variabili dell'altro processo. I file descriptor sono duplicati.
Le tabelle del kernel dei file aperti non sono duplicate.
Il figlio eredita i permessi di accesso ai file aperti.
I reference counter nelle tabelle del kernel sono incrementati per:

1. area codice
2. entry nella tabella file del kernel (per file aperti)
3. entry nella tabella i-node in memoria



Terminazione Sincrona di Processo

void exit(int status)

La system call void exit(int status) viene chiamata implicitamente dalla libreria C all'uscita dal main.

Fra le operazioni eseguite dal kernel c'e' la chiusura dei file aperti, la liberazione della memoria e il cambiamento dello stato del processo in **zombie**.

Un processo zombie non e' piu' schedulato anche se e‘ ancora presente inella Tavola dei processi.

Infine viene inviato al processo genitore un segnale di avvenuta terminazione.
Sono comunque mantenuti e aggiornati (nella Tavola dei processi) i tempi di esecuzione relativi al processo terminato.

Insieme al segnale di avvenuta terminazione, al padre viene anche passato un valore intero di terminazione, lo status.

Attesa Terminazione Figlio

pid_t **wait(int *status)**

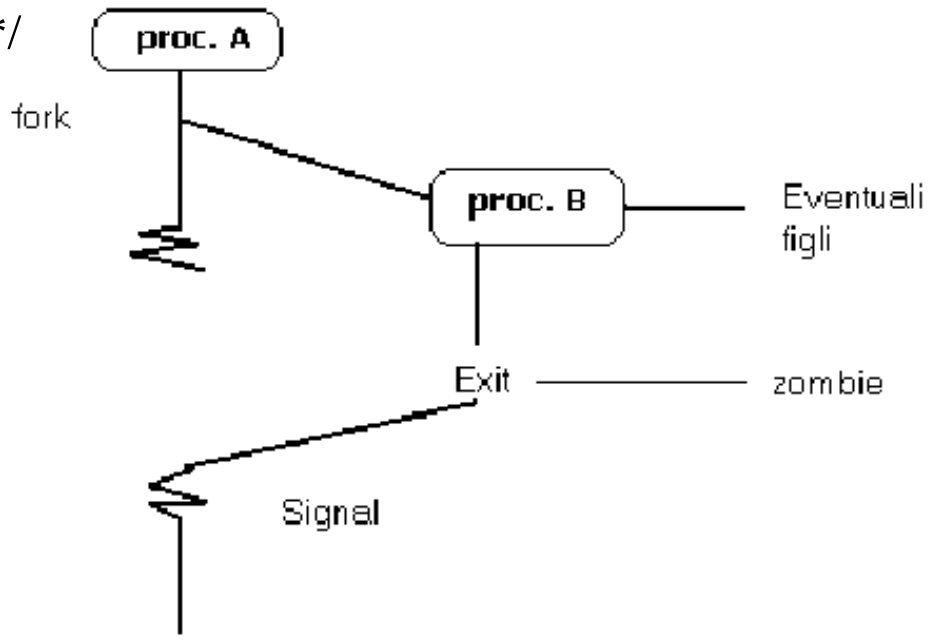
Il processo che chiama la system call pid = **wait(&status)** rimane sospeso (non piu' schedulato, quindi in stato waiting) fino alla morte di uno dei suoi figli, in particolare fino a che uno dei figli esegue la funzione exit. Il genitore raccoglie il signal emesso durante la exit.

Se il processo non ha figli e' ritornato -1 e settato errno.
In caso contrario, viene restituito il pid del figlio terminato.

Se un figlio termina prima che il genitore esegua la wait, ne viene tenuta traccia. Quando il genitore chiamerà la wait() non verrà quindi sospeso, ma continuerà immediatamente la sua esecuzione.

I figli che hanno chiamato la exit, ma il cui padre non ha ancora chiamato la wait vanno in uno stato detto zombie e rimangono nella tabella dei processi del kernel. Nel momento in cui il padre chiama la wait i figli zombie sono cancellati dalla tavola dei processi.

```
int status; pid_t pid;
if ((pid = fork()) == 0)
{
      /* processo figlio */
      ..... exit(1);
}
/* processo padre */
pid = wait (&status);
.....
```



Esecuzione di Programmi

- **Famiglia di funzioni int exec...(**diversi parametri**)**

E' possibile mandare in esecuzione altri programmi all'interno di un processo, mediante la famiglia di funzioni “exec“.
Tali funzioni differiscono solo per i tipo di parametri con cui possono essere chiamate. Esistono ad esempio:

int **execv** (char *nomefile, char *argv[]);
int **execl** (char *nomefile,char *arg1,char *arg2,.,char *argn,0);
e altre ancora .... definite in <unistd.h>
dove:
nomefile e' il nome del file che contiene l'eseguibile
argv e' il puntatore ad un array di puntatori a caratteri, ognuno dei quali punta ad una stringa che verrà passata come argomento al processo chiamato
arg1, arg2, argn puntano ciascuno ad una stringa che verrà passata come argomento al programma

Il nuovo programma si sostituisce interamente, come dati e codice , a quello vecchio, che non e' piu' "raggiungibile". mantre restano inalterate le tavole file (file aperti, posizionamento all'interno di essi, relazioni con altri processi ecc.).

ESEMPIO..... editare il file augusto.txt con l’editor /usr/bin/vi

```
#include <unistd.h>
void main(void) {
        execl(“/usr/bin/vi”, “augusto.txt”,0);
        printf("errore in chiamata a /usr/bin/vi\n");
}
```

La funzione printf viene eseguita solo in caso di errore della funzione exec, cioe' solo se l'operazione fallisce, per cui il processo continua con lo stesso programma, e non chiama l’altro.

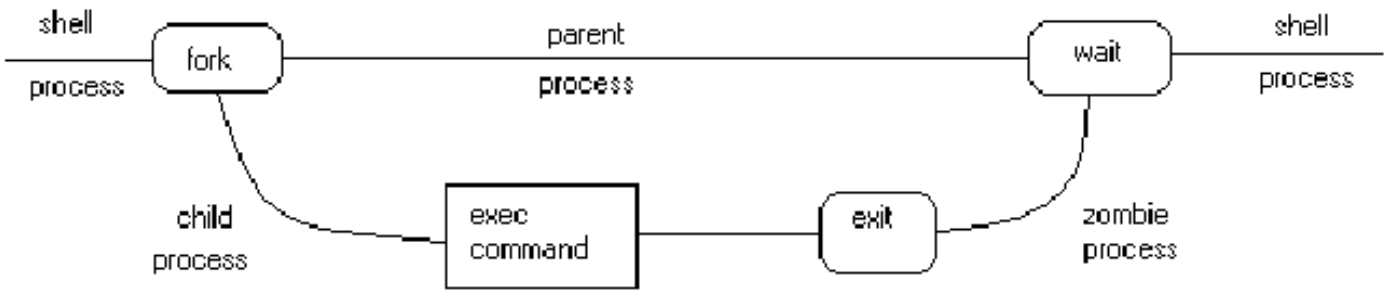
Esecuzione di più Programmi

Le system call di tipo ***exec*** possono essere associate alla fork per ottenere l'esecuzione del programma chiamato senza interrompere l'esecuzione del programma chiamante.

ESEMPIO..... editare il file augusto.txt con l'editor /usr/bin/vi

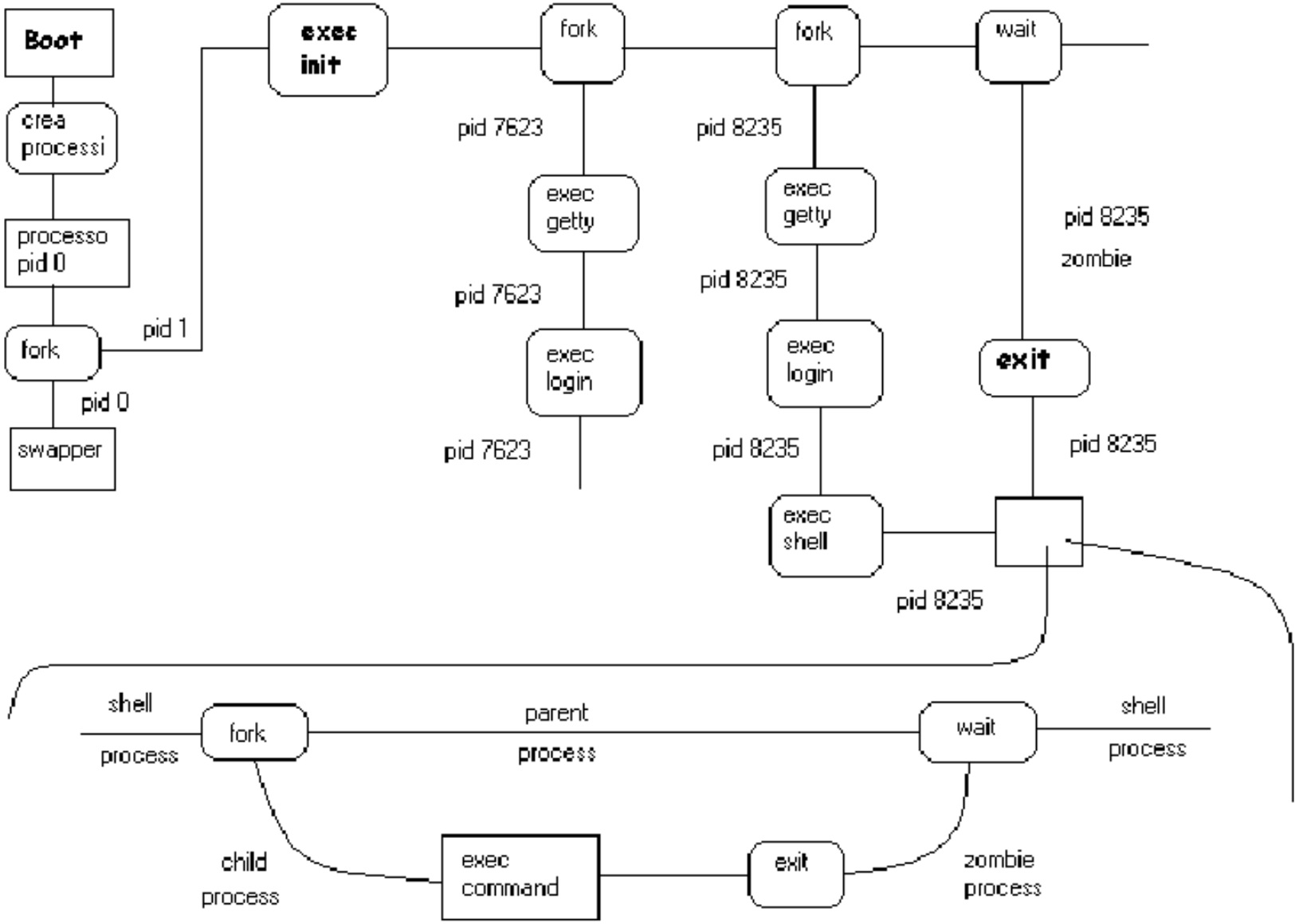
```
#include <unistd.h>
void main(void) {
        pid_t pid;  int status;
        pid=fork();
        if(pid==0) ) { /* figlio */
                execl("/usr/bin/vi", "augusto.txt",0);
                printf("errore in chiamata a /usr/bin/vi\n");
                exit(1);
        }
        wait (&status);
         .............
}
```

Questo meccanismo viene applicato anche nelle shell di comandi per eseguire dei programmi: prima con una fork si duplica la shell, e il padre viene messo in attesa della terminazione del figlio.
Il figlio chiama la exec per eseguire il programma voluto, tale programma si sostituisce al figlio e alla fine dell'esecuzione avvisa il padre, la shell, che riprenderà il controllo.



Esecuzione di più Programmi

Lo stesso meccanismo (fork + exec) è utilizzato a Linux, per creare i processi iniziali del s.o.



INTERPROCESS COMMUNICATION 27

- Le comunicazioni tra processi (IPC, Intreprocess Communication) sono realizzate mediante strutture dati rese disponibili dal kernel.

Sono disponibili 3 tipologie di comunicazioni tra processi:
- memoria condivisa (**shared memory segments**)
- semafori (**semaphore arrays**)
- code di messaggi (**message queues**)

- Ciascuna di queste strutture dati sono indicate da un identificatore, che ha lo stesso significato dell'identificatore di file aperto.
- Mediante tale identificatore i processi possono acquisire, utilizzare e rimuovere le strutture.
- Tale identificatore viene ottenuto da una system call, specificando alcuni parametri per permettere ad insiemi di processi diversi di condividere strutture diverse.

- Esiste un eseguibile **/usr/bin/ipcs** che permette di visualizzare le strutture allocate, e che mostra anche l'identificatore di ciascuna struttura e l'utente proprietario.

- Qualora i processi non abbiano rimosso le strutture allocate (ad es. in caso di terminazione anomala), per rimuoverle si può utilizzare il pro_ gramma **/usr/bin/ipcrm** per rimuovere una data struttura noto il suo identificatore.

Output del Programma ***ipcs***

```
[vittorio@poseidon prova]$ ipcs

------ Shared Memory Segments --------
key            shmid      owner      perms      bytes      nattch     status
0x00000001 3969       vittorio    600        100        0

------ Semaphore Arrays --------
key            semid      owner      perms      nsems      status

------ Message Queues --------
key            msqid      owner      perms      used-bytes  messages
```

Shared Memory Segments

Normalmente i processi Unix non condividono la memoria;
nemmeno i processi 'parenti':
infatti il figlio eredita UNA COPIA dei dati del padre;
se i due processi modificano quei dati dopo la fork, ognuno modifica la propria copia.
La possibilita' di condividere memoria e' stata aggiunta (come le altre funzioni IPC) nella versione System V e poi utilizzata anche nelle altre versioni di Unix.

Il segmento di memoria condivisa è caratterizzato da un identificatore (detto chiave, key).

Quindi se piu' processi vogliono usare lo stesso segmento di memoria condivisa, questi dovranno richiederla esplicitamente, specificando tutti la stessa key (devono sapere il valore di key).

Oppure, se i processi che devono condividere il segmento di memoria sono tutti figli di uno stesso processo, il padre prima di creare i figli potrà mappare la memoria condivisa, ed i figli avranno automaticamente una copia della chiave di accesso alla memoria condivisa.

Questi due modalità sono realizzate con un uso diverso delle stesse system call.

Shared Memory Segments

Ad un segmento di memoria condivisa si accede tramite le seguenti system calls

int **shmget** (key_t **key**, int **size**, int **shmflg**);
void ***shmat** (int **shmid**, void ***shmaddr**, int **shmflg**);

definiti negli header

sys/types.h , sys/ipc.h e sys/shm.h

shmget restituisce un identificatore di un segmento di memoria condivisa di dimensione size;

a seconda del valore di key, il segmento viene creato o no.
Il segmento viene CREATO se
- key vale IPC_PRIVATE (0),

oppure se
- non esiste il segmento associato a key, e semflg&IPC_CREAT e' vero,

Quindi se si vuole che piu' processi acquisiscano la stessa struttura si puo':

1) far partire indipendentemente diversi processi e fare chiamare a tutti la *semget* con la stessa chiave key diversa da IPC_PRIVATE.
   Tale key deve essere DIVERSA da chiavi usate da altri processi anche di altri utenti
2) far acquisire le strutture da un processo padre e farle 'ereditare' dai processi figli, che quindi non devono fare semget, ma solo usare l'identificatore restituito dalla semget al padre. In questo caso il padre puo' usare IPC_PRIVATE come chiave.

il parametro shmflg definisce i permessi di lettura scrittura

La system call

void ***shmat** (int **shmid**, void ***shmaddr**, int **shmflg**);

collega shmget restituisce un identificatore di un segmento di memoria

- shmat esegue il collegamento del segmento di memoria condivisa, ottenuto dalla chiamata alla shmget, ad un indirizzo nello spazio di indirizzi del processo chiamante.
- Praticamente, restituisce un indirizzo che punta all'area di memoria condivisa, che potrà essere utilizzato per i successivi accessi.
- Se il parametro shmaddr e' NULL (consigliato), l'indirizzo viene scelto dal sistema, altrimenti dall'utente.
- Il parametro shmflg, se posto a zero, assegna capacità di leggere e scrivere nel segmento di memoria condivisa.

La system call

int **shmdt** (char *shmaddr);

scollega l'area di memoria condivisa puntata da shmaddr dallo spazio di memoria indirizzabile dall'utente, ma non rimuove l'area condivisa.

La system call

int **shmctl** (int shmid, int cmd, struct shmid_ds

serve per rimuovere un segmento di memoria condivisa.
Si specifica cmd = IPC_RMID
e terzo parametro nullo.

Shared Memory: Esempio Padre/Figlio

```
#include <malloc.h>  #include <stdio.h> #include <sys/types.h>
#include <sys/ipc.h> #include <sys/sem.h>
main()  {
int shmid1; char *addr1;
/* alloca memoria condivisa per un array di 100 caratteri */
shmid1 = shmget(IPC_PRIVATE,100*sizeof(char),0600);
if (shmid1 == -1)
      { perror("Creazione memoria condivisa"); exit(1); }
// ottengo un puntatore all'area condivisa
addr1 = (char *) shmat ( shmid1, 0, 0);
/*    E SE AVESSI ALLOCATO in maniera canonica ?
      addr1= malloc(100);
      if(addr1==NULL) printf("errore nella malloc"); exit(0);
      */
// assegno un valore al primo carattere
*addr1='A';
if (fork() == 0)
  {   // processo figlio, cambia valore al primo carattere
      *addr1='C';
      shmdt(addr1);
      printf("figlio: carattere %c\n", *addr1);
   }
else  {  // processo padre, attendo che il figlio abbia scritto
      sleep(5);
      printf("padre: carattere: %c\n", *addr1);
      // attendo che il figlio termini
      wait(NULL);
      shmctl(shmid1,IPC_RMID,NULL);
   }
}
```

lancio prima il processo 1, e immediatamente il processo 2

```
// processo 1
main()  { int shmid1; char *addr1;
        int key=1;
        shmid1 = shmget(key,100*sizeof(char),0600);
        addr1 = (char *) shmat ( shmid1, 0, 0);
        // assegno un valore al primo carattere
        *addr1='A';
        printf("proc1: prima, carattere: %c\n", *addr1);
        // attendo che l'altro processo scriva
        sleep(10);
        // dovrei stampare B
        printf("proc1: dopo, carattere: %c\n", *addr1);
        shmctl(shmid1,IPC_RMID,NULL);
}

// processo 2
main()  {
        int shmid1; char *addr1;
        int key=1;
        shmid1 = shmget(key,100*sizeof(char),0600);
        addr1 = (char *) shmat ( shmid1, 0, 0);
        // LEGGO
        printf("proc2: prima carattere: %c\n", *addr1);
        *addr1='B';
        shmdt(addr1);
}
```