Memoria condivisa in Unix System V

La memoria condivisa consente a due o più processi di condividere una particolare area di memoria centrale.

Questa rappresenta la forma più efficiente di comunicazione tra processi, in quanto i dati non devono essere copiati tra client e server.

In genere è richiesto un supporto hardware, e perciò non sempre questo servizio è disponibile.

L'unico problema creato dalla memoria condivisa è la sincronizzazione tra client e server (tipicamente realizzata mediante semafori). Se un processo sta scrivendo nell'area dati comune, gli altri non possono accedere all'area prima che abbia terminato.

Il kernel gestisce la seguente struttura dati:

```
struct shmid_ds {
  struct ipc_perm shm_perm;     /* info su proprietari
                                   permessi, etc */
  int shm_segsz;                /* dimensione segmento */
  pid_t           shm_lpid;     /* pid ultima shmop */
  pid_t           shm_cpid;     /* pid creatore */
  shmatt_t        shm_nattch;   /* # attaches */
  time_t          shm_atime;    /* tempo ultima shmat */
  time_t          shm_dtime;    /* tempo ultima shmdt */
  time_t          shm_ctime;    /* ultima modifica */
};
```

Alcuni limiti associati alla memoria condivisa:

SHMMAX	Dimensione massima del segmento
SHMMIN	Dimensione minima del segmento
SHMMNI	Numero massimo di segmenti nel sistema
SHMSEG	Numero massimo di segmenti per prcesso

La primitiva **`shmget`**

La primitiva `shmget` (analoga alla `msgget`) consente di "aprire" un segmento di memoria ed ottenerne l'identificatore.

Uso:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget( key_t key, int size, int flag );
```

Il significato dei parametri è il seguente:

`key`	è la chiave del segmento di memoria che si intende aprire
`size`	specifica la dimensione in byte del segmento
`flag`	specifica i diritti di accesso (come in `msgget/semget`)

Il valore di ritorno di `shmget` è l'identificatore del segmento aperto (oppure -1 in caso di errore).

Quando si crea un segmento nuovo, alcuni campi della struttura `shmid_ds` sono inizializzati come segue:

- nella struttura ipc_perm si scrivono informazioni relative al proprietario, diritti di accesso (contenuti in `flag`), etc.
- `shm_lpid`, `shm_nattach`, `shm_atime`, etc. sono posti a 0
- `shm_ctime` (ultima modifica) è posto uguale al tempo corrente

Nel caso in cui si crei un segmento nuovo, il parametro `size` deve essere specificato. Se, invece, si apre un semaforo già creato si può usare `nsems=0`.

La primitiva `shmctl`

La `shmctl` viene impiegata per svolgere varie funzioni:

Uso:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl( int shmid, int cmd, struct shmid_ds *buf );
```

Il significato dei parametri è il seguente:

`shmid`	è l'identificatore del segmento
`cmd`	specifica l'azione da eseguire, fra le 5 possibili
`buf`	puntatore a struttura `shmid_ds`

Il parametro cmd può essere:

`IPC_STAT`	copia le informazioni riguardanti il segmento (memorizzate in `shmid_ds`) in `*buf`
`IPC_SET`	consente di modificare il valore di alcuni parametri associati al segmento (user-id, group-id, modalità di accesso) E' consentita solo al proprietario della coda o al super-user.
`IPC_RMID`	rimuove il semaforo dal sistema (consentito solo al proprietario e il super-user).
`SHM_LOCK`	blocca l'accesso al segmento (solo super-user)
`SHM_UNLOCK`	sblocca l'accesso al segmento (solo super-user)

La primitiva `shmat`

La `shmat` viene impiegata per "agganciare" un segmento di memoria all'area dati (logica) di un processo.

Uso:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat( int shmid, void *addr, int flag );
```

Il significato dei parametri è il seguente:

`shmid` è l'identificatore del segmento

`addr` consente di specificare l'indirizzo di allocazione del segmento

`flag` due possibili flag sono: `SHM_RDONLY` (segmento utilizzabile solo in letura) e `SHM_RND` (per "arrotondare" l'indirizzo di allocazione in memoria)

Il parametro `addr` viene interpetato come segue:

`addr=0`: il segmento è allocato dal kernel al primo indirizzo utile (è la tecnica raccomandata)

`addr≠0`: il segmento verrà allocato all'indirizzo specificiato, esattamente (se `SHM_RND` non è specificato) oppure mediante arrotondamento (se `SHM_RND` è settato).

Il valore di ritorno di `shmat` è l'indirizzo del segmento di memoria, oppure -1 in caso di errore.

La primitiva `shmdt`

La `shdt` viene impiegata per "distaccare" un segmento di memoria condivisa dall'area dati di un processo.

Svolge la funzione opposta a quella realizzata da `shmat`.

Uso:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt( void *addr );
```

Il parametro addr specifica l'indirizzo del segmento da sganciare.

Il valore di ritorno è -1 in caso di errore, 0 altrimenti.

Esempio

Realizzare un programma (`shmcopy`) che copi il suo standard input sul suo standard output, utilizzando la memoria condivisa.

```
/***
*
*  share_ex.h -- header file for shared memory example
*
***/


#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define SHMKEY1      (key_t) 010     /* shared mem key */
#define SHMKEY2      (key_t) 020     /* shared mem key */
#define SEMKEY       (key_t) 050     /* semaphore key */

#define SIZ          500             /* buffer size */


/*--- will hold data & read count ---*/
struct databuf {
  int d_nread;
  char d_buf[SIZ];
  };
```

```
/***
*
*  inizialization routines for shared memory example
*
***/

#include "share_ex.h"

#define IFLAGS       (IPC_CREAT | IPC_EXCL)
#define ERR          ((struct databuf *) -1)

int shmid1, shmid2, semid;


/*--- create + attach shared memory segment ---*/

getseg( struct databuf **p1, struct databuf **p2 )
{

/* create chared memory segments */

if((shmid1 =
    shmget(SHMKEY1,sizeof(struct databuf),0600|IFLAGS))<0){
   perror("shmget failed");
   exit(1);
   }
if((shmid2 =
    shmget(SHMKEY2,sizeof(struct databuf),0600|IFLAGS))<0){
   perror("shmget failed");
   exit(1);
   }

/* attach shared memory segments */

if((*p1=(struct databuf *) shmat(shmid1,0,0)) == ERR) {
   perror("shmat failed");
   exit(1);
  }
if((*p2=(struct databuf *) shmat(shmid2,0,0)) == ERR) {
   perror("shmat failed");
   exit(1);
  }

} /* end pf getseg */
```

```
/*--- get semaphore set ---*/

int getseg()
{

/* create two semaphore sets */

if((semid = semget(SEMKEY1,2,0600|IFLAGS)) < 0) {
  perror("semget failed");
  exit(1);
  }

/* set initial values */

if( semctl(semid,0,SETVAL,0) < 0 ) {
  perror("semctl failed");
  exit(1);
  }

if( semctl(semid,1,SETVAL,0) < 0 ) {
  perror("semctl failed");
  exit(1);
  }

return(semid);

} /* end of getseg */



/*--- remove shared memory segment + sem set ---*/

remove()
{

if( shmctl(shmid1,IPC_RMID,(struct shmid_ds *)0) < 0) {
  perror("shmctl failed");
  exit(1);
  }

if( shmctl(shmid2,IPC_RMID,(struct shmid_ds *)0) < 0) {
  perror("shmctl failed");
  exit(1);
  }

if( semctl(semid,IPC_RMID,(struct semid_ds *)0) < 0) {
  perror("semctl failed");
  exit(1);
  }

} /* end of remove */
```

```
/***
*
*  shmcopy -- main function
*
***/


#include "share_ex.h"


main()
{

int semid, pid;
struct databuf *buf1, *buf2;

/* initialize semaphore set */
semid = getsem();

/* create and attach shared memory segments */
getseg(&buf1,&buf2);

switch( pid=fork() ) {

  case -1:      /* something went wrong */
     perror("fork");
     exit(1);

  case 0:       /* I'm the child: the writer */
     writer(semid,buf1,buf2);
     remove();
     break;

  default:      /* I'm the parent: the reader */
     reader(semid,buf1,buf2);
     break;

  }

exit(0);

} /* end of main */
```

```
/***
*
*  reader -- handle reading of file
*
***/

#include "share_ex.h"

#define STDIN   0
#define STDOUT  1


/* these define p() and v() for two semaphores */
struct sembuf p1={0,-1,0}, p2={1,-1,0};
struct sembuf v1={0,1,0}, v2={1,1,0};


reader( int semid,
        struct databuf *buf1, struct databuf *buf2 )

{

for(;;) {

  /* read into buffer buf1 */
  buf1->d_nread = read(STDIN,buf1->d_buf,SIZ);

  /* synchronization point */
  semop(semid, &v1, 1);
  semop(semid, &p2, 1);

  /* test here to avoid writer sleeping */
  if(buf1->d_nread <= 0)
    return;


  buf2->d_nread = read(STDIN,buf2->d_buf,SIZ);

  semop(semid, &v2, 1);
  semop(semid, &p1, 1);

  if(buf2->d_nread <= 0)
    return;


  }

} /* end of reader */
```

```
/***
*
*  writer -- handle writing of file
*
***/

#include "share_ex.h"

#define STDIN   0
#define STDOUT  1


/* already defined in reader.c */
extern struct sembuf p1, p2;
extern struct sembuf v1, v2;


writer( int semid,
        struct databuf *buf1, struct databuf *buf2 )

{

for(;;) {

  semop(semid, &p1, 1);
  semop(semid, &v2, 1);

  if(buf1->d_nread <= 0)
    return;

  write(STDOUT, buf1->d_buf, buf1->d_nread);


  semop(semid, &p2, 1);
  semop(semid, &v1, 1);

  if(buf2->d_nread <= 0)
    return;

  write(STDOUT, buf2->d_buf, buf2->d_nread);

  }

} /* end of writer */
```

Dopo averlo compilato, il programma può essere eseguito con un comando del tipo:

```
$ shmcopy < big > /tmp/big
$ _
```

I comandi `ipcs` e `ipcrm`

Unix mette a disposizione due comandi shell per gestire strutture IPC.

Il comando **`ipcs`** stampa informazioni sullo stato corrente delle strutture IPC.

Esempio:

```
$ ipcs

Message Queues:
T       ID      KEY        MODE         OWNER    GROUP
q        0 1191574507 --rw-------       root    system

Shared Memory:
T       ID      KEY        MODE         OWNER    GROUP
*** No shared memory segments are currently defined ***

Semaphores:
T       ID      KEY        MODE         OWNER    GROUP
s        0 1090534506 --ra-------       rossi    users
```

Il comando **`ipcrm`** serve per rimuovere dal sistema una struttura IPC.

Esempio:

```
$ ipcrm -s 0    ... rimuove semaforo con identificatore 0

$ ipcrm -Q 200  ... rimuove coda messaggi con key 200
```