Unix System V Interprocess Communication (IPC)

La comunicazione tra processi (IPC) è un aspetto importante di ogni sistema operativo che consente a processi arbitrari di scambiare dati e/o sincronizzarsi.

Alcuni semplici meccanismi di comunicazione tra processi sono stati già incontrati:

- fork (un processo padre comunica con il figlio mediante copia dell'immagine)
- kill (il messaggio scambiato può essere soltanto un segnale).
- pipe (possibile soltanto se i due processi hanno un antenato comune)

Unix System V offre meccanismi per la comunicazione tra processi molto più sofisticati, comunemente denominati "System V IPC".

I servizi di System V IPC sono di tre tipi:

- code di messaggi
- semafori
- memoria condivisa

Le modalità di impiego sono simili:

	Code di messaggi	Memoria condivisa	Semafori
include	`<sys/msg.h>`	`<sys/shm.h>`	`<sys/sem.h>`
create/open	`msgget`	`shmget`	`semget`
controllo	`msgctl`	`shmctl`	`semctl`
operazioni	`msgsnd`	`shmat`	`semop`
	`msgrcv`	`shmdt`	

Identificatori e chiavi

Il kernel associa a ciascuna struttura IPC (code di messaggi, semafori, o memoria condivisa) un intero non negativo chiamato **identificatore**. Qualunque accesso alla struttura deve avvenire mediante il corrispondente identificatore.

La più importante caratteristica comune fra i tre meccanismi IPC è il concetto di **chiave**. Le chiavi sono numeri che consentono ai processi di condividere la struttura IPC (in analogia con i nomi di file, che univocamente identificano un file).

Per garantire l'univocità nell'ambito dei soli processi cooperanti, si può utilizzare la funzione `ftok`:

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

key_t    keyval;

keyval = ftok(char *pathname, char project)
```

La funzione `ftok` accetta in ingresso il nome di fle e un numero intero che specifica un "progetto" e restituisce una chiave unica basata sull'i-node del file e del parametro `project`.

- I processi che cooperano utilizzano lo stesso file, che deve esistere
- Il parametro project differenzia più canali nell'ambito della stessa cooperazione

Operazioni `get`

I programmi usano la chiave sia per creare una struttura IPC, che per ottenere l'accesso ad una già esistente.

Il risultato di una operazione `get` è un identificatore (numero intero) che verrà in seguito usato per chiamare altre routine IPC.

In analogia con il file system, una operazione `get` è simile ad una operazione `creat` o `open`; l'identificatore restituito è simile al descrittore di file.

La struttura comune di una funzione get è la seguente:

```
int xxxget(key_t key, ..., int flags)
```

dove:

`xxx`	può essere `msg/shm/sem`
`key`	identificatore unico dell'oggetto IPC
`flags`	permessi di accesso

Code di messaggi

Un messaggio è semplicemente una sequenza di caratteri o byte.

I messaggi vengono passati tra processi per mezzo di code di messaggi, che sono create o aperte mediante la primitiva **`msgget`**.

Dopo aver creato o aperto una coda, i processi che hanno appropriati diritti di accesso possono mandare messaggi usando la primitiva **`msgsnd`**, o ricevere messaggi usando la primitiva **`msgrcv`** (la quale provvede anche a cancellare il messaggio dalla coda).

L'uso tipico della primitiva `msgget` è il seguente:

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

int msg_qid, permflags;
key_t key;
:
msg_qid = msgget( key, permflags );
:
```

Il parametro `permflags` determina precisamente l'azione da compiere. Due costanti tipicamente usate, contenute nel file `ipc.h`, sono:

`IPC_CREAT`	crea una coda relativa al valore key, se non ne esiste già una, o ne apre una esistente
`IPC_EXCL`	in congiunzione con `IPC_CREAT`, specifica la creazione di una coda nuova. Se una coda di messaggi con la chiave specificata esiste già, restituirà il valore -1

Esempio:

```
msgq_id = msgget((key_t)0100, 0644 | IPC_CREAT | IPC_EXCL)
```

L'operazione crea una nuova coda con chiave 0100 e diritti di accesso 0644 (il proprietario può leggere e scrivere, tutti gli altri soltanto leggere).

La struttura `msqid_ds`

Ad ogni coda di messaggi, il kernel associa una struttura dati chiamata `msqid_ds` che contiene informazioni riguardanti lo stato della coda.

La definizione della struttura si trova in `msg.h`.

Le informazioni contenute in `msqid_d` sono le seguenti:

```
struct msqid_ds  {
  struct ipc_perm msg_perm;  /* info su proprietari,
                                gruppo, etc */
  struct msg *msg_first;     /* ptr a primo messaggio */
  struct msg *msg_last;      /* ptr a ultimo messaggio */
  ulong       msg_cbytes;    /* # byte correnti in coda */
  ulong       msg_qnum;      /* # messaggi in coda */
  ulong       msg_qbytes;    /* max # byte in coda */
  pid_t       msg_lspid;     /* pid di ultima msgsnd() */
  pid_t       msg_lrpid;     /* pid di ultima msgrcv() */
  time_t      msg_stime;     /* tempo ultima msgsnd() */
  time_t      msg_rtime;     /* tempo ultima msgrcv() */
  time_t      msg_ctime;     /* tempo ultima modifica */
};
```

Il nucleo utilizza alcune costanti che definiscono i limiti associati ad una (o più) code:

`MSGMAX`	numero massimo di byte di un messaggio
`MSGMNB`	numero massimo di byte di una coda
`MSGMNI`	numero massimo di code di messaggi nel sistema
`MSGTQL`	numero massimo di messaggi nel sistema

L'operazione `msgsnd`

Dopo aver creato una coda, è possibile scrivere messaggi mediante la primitiva `msgsnd`.

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

int msg_qid, size, flags, retval;
struct my_msg {
     long mtype;
     char mtext[SOMEVALUE];
     } message;
:
:
retval = msgsnd(msg_qid, &message, size, flags );
```

Il significato dei parametri è il seguente:

`msg_qid` è l'identificatore della coda

`message` contiene il messaggio. Il campo `mtype` è impiegato per raggruppare i messaggi in classi diverse (sono ammessi solo valori positivi);
il campo `mtext` conterrà il testo del messaggio.

`size` è la lunghezza del messaggio realmente inviato

`flags` può assumere solo un valore significativo:
`IPC_NOWAIT`. Se il flag non è settato, il processo chiamante verrà sospeso se non ci sono sufficienti risorse per inviare il messaggio
Se, invece, `IPC_NOWAIT` è settato la funzione ritornerà immediatamente anche se il messaggio non è stato inviato, e il suo valore di ritorno (`retval`) sarà -1.

Il valore di ritorno sarà -1 anche quando il chiamante non ha il permesso di scrivere sulla coda.

L'operazione `msgrcv`

Per ricevere messaggi si usa la primitiva `msgrcv`.

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

long msg_type;
int msg_qid, size, flags, retval;
struct my_msg {
     long mtype;
     char mtext[SOMEVALUE];
     } message;
:
retval=msgrcv(msg_qid,&message,size,msg_type,flags);
```

Il significato dei parametri è il seguente:

`msg_qid` è l'identificatore della coda

`message` conterrà il messaggio letto

`size` è il numero di byte che si intende leggere

`msg_type` ci sono tre casi:

- `=0` viene letto il primo messaggio in coda
- `>0` viene letto il primo messaggio avente il tipo uguale a `msg_type`
- `<0` viene letto il primo messaggio il cui tipo è il più piccolo valore minore o uguale del valore assoluto di `msg_type`

`flags` può assumere solo due valore significativi, da soli o combinati insieme mediante l'or.

- `IPC_NOWAIT:` stabilisce se ritornare o meno quando un messaggio del tipo desiderato non è disponibile
- `MSG_NOERROR:` se settato, il messaggio viene troncato se più lungo di `size`

Il valore di ritorno (`retval`) conterrà il numero di byte effettivamente letti (o -1 in caso di errore).

Esempio

Sviluppare un sistema per il passaggio di messaggi tra processi, in cui l'accesso ai messaggi avviene mediante priorità.

```
/***
*
*  q.h -- header for message facility example
*
***/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

extern int errno;

#define   QKEY       (key_t)0105     /* identifying key */
#define   QPERM      0660            /* permissions */
#define   MAXOBN     50              /* max len obj name */
#define   MAXPRIOR   10              /* max priority lev */

struct q_entry {
     long mtype;
     char mtext[MAXOBN+1];
};
```

```
/***
*
*     enter -- place an object into queuee
*
***/


#include   "q.h"

static int s_qid = -1;     /* message queue identifier */

int enter( char *objname, int priority )
{
int len;
struct q_entry s_entry;   /* struct to hold message */

/*--- validate name length & priority level ---*/
if( (len = strlen(objname)) > MAXOBN ) {
      warn("name too long);
      return(-1);
      }

if(priority > MAXPRIOR || priority < 0) {
      warn("invalid priority level");
      return(-1);
      }

/*--- initialize message queue as necessary ---*/
if( s_qid = init_queue() == -1 )
      return(-1);

/*--- initialize s_entry ---*/
s_entry.mtype = (long)priority;
strncpy(s_entry.mtext, objname, MAXOBN);

/*--- send message; waiting if necessary ---*/
if( msgsnd(s_qid, &s_entry, len, 0) == -1 ) {
      perror("msgsnd failed");
      return(-1);
      }
else
      return(0);

} /* end of enter */


/***
*  warn -- print warning messages
***/

warn (char *s)
{
fprintf( stderr, "Warning: %s\n", s ;);
}
```

```
/***
*
*    init_queue -- get queue identifier
*
***/

#include  "q.h"

int init_queue()
{
int queue_id

/*--- attempt to create messaage queue ---*/
if( (queue_id = msgget(QKEY, IPC_CREAT | QPERM)) == -1 )
     perror("msgget failed");

return( queue_id );
}


/***
*
*    server -- serve object with highest priority on queue
*
***/

#include "q.h"

static int r_qid = -1;

int serve()
{
struct q_entry r_entry;
int mlen;

/*--- initialize message queue as necessary ---*/
if( r_qid = init_queue() == -1 )
     return(-1);

/*--- get and process next message, waiting if necessary */
for(;;) {
     if( (mlen = msgrcv(r_qid, &r_entry, MAXOBN,
               (long) -1*MAXPRIOR, MSG_NOERROR ) == -1 ) {
          perror("msgrcv failed");
          return(-1);
          }
     else {
          r_entry.mtext[mlen] = '\0';  /* to get string*/

          /*--- process object name ---*/
          proc_obj(&r_entry);
          }
     }
} /* end of serve */
```

```
/***
*
*    etest -- enter object names in the queue
*
***/

#include  <stdio.h>
#include  "q.h"

main( int argc, *argv[] )
{
int priority;

if( argc != 3 ) {
  fprintf( stderr, "Usage: %s objname priority\n");
  exit(1);
  }

if(priority = atoi(argv[2]) <= 0 || priority > MAXPRIOR){
  warn("invalid priority level");
  exit(2);
  }

if( enter(argv[1], priority) < 0 ) {
  warn("enter failure");
  exit(3);
  }

exit(0);
} /* end of main */
```

```
/***
*
*     stest -- simple server for queue
*
***/

#include   <stdio.h>
#include   "q.h"

main()
{

int pid;

switch( pid = fork() ) {

  case 0:       /* child */
    setpgrp();       /* break with terminal */
    serve();
    break;

  case -1:      /* error */
    warn("fork to start server failed");
    break;

  default:
    printf("server process pid is %d\n", pid );
  }

exit( pid != -1 ? 0 : 1 );
} /* end of main */


/***
*     proc_obj -- do something with objects
***/

proc_obj( struct q_entry msg )
{
     printf("priority: %ld, name: %s\n", msg->mtype,
          msg->mtext);
}
```

Esempio :

```
$ etest objname1 3
$ etest objname2 4
$ etest objname3 1
$ etest objname4 9
$ stest
server process pid is 2456
$
priority: 1, name: objname3
priority: 3, name: objname1
priority: 4, name: objname2
priority: 9, name: objname4
```

La primitiva `msgctl`

La primitiva `msgctl` realizza tre funzioni:

- ottenere informazioni riguardo lo stato di una coda di messaggi
- cambiare alcuni limiti associati ad una coda
- rimuovere una coda di messaggi dal sistema

Uso

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
:
int msg_qid, command, retval;
struct msqid_ds msq_stat;
:
retval = msgctl(msg_qid, command, &msq_stat);
```

dove:

`msg_qid`	è l'identificatore della coda
`command`	specifica l'azione da compiere
`msq_stat`	conterrà informazioni sullo stato della coda
`retval`	valore di ritorno: 0 se tutto OK; -1 altrimenti

Il parametro `command` può assumere uno dei seguenti tre valori:

`IPC_STAT`	copia le informazioni riguardanti la coda in `msq_stat`
`IPC_SET`	consente di modificare il valore di alcuni parametri associati alla coda (user-id, group-id, modalità di accesso, numero max di byte). E' consentita solo al proprietario della coda o al super-user. Solo il super-user può modificare `msg_qbytes` (max byte in coda) .
`IPC_RMID`	rimuove la coda dal sistema (consentito solo al proprietario e il super-user)