

Programmazione di Sistema in UNIX

Nicola Bombieri - Nicola Drago - Graziano Pravadelli

Università di Verona
Dipartimento di Informatica
Verona

Sommario

- System call:
 - Comunicazione tra processi: inter process communication (IPC)

System Call per la Comunicazione tra Processi (IPC)

Introduzione

- UNIX e IPC
- Pipe
- FIFO (*named pipe*)
- Code di messaggi (*message queue*)
- Memoria condivisa (*shared memory*)
- Semafori

UNIX e IPC

- `ipcs`: riporta lo stato di tutte le risorse, o selettivamente, con le seguenti opzioni:
 - `-s` informazioni sui semafori;
 - `-m` informazioni sulla memoria condivisa;
 - `-q` informazioni sulle code di messaggi.
- `ipcrm`: elimina le risorse (se permesso) dal sistema.
 - Nel caso di terminazioni anomale, le risorse possono rimanere allocate
 - Le opzioni sono quelle `ipcs`
 - Va specificato un ID di risorsa, come ritornato da `ipcs`

UNIX e IPC

- Esempio:

```
host:user> ipcs
```

```
IPC status from /dev/kmem as of Wed Oct 16 12:32:13 1996
```

```
Message Queues:
```

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

```
*** No message queues are currently defined ***
```

```
Shared Memory
```

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

m	1300		0 D-rw-----	root	system
---	------	--	-------------	------	--------

m	1301		0 D-rw-----	root	system
---	------	--	-------------	------	--------

m	1302		0 D-rw-----	root	system
---	------	--	-------------	------	--------

```
Semaphores
```

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

```
*** No semaphores are currently defined ***
```

Pipe

- Il modo più semplice di stabilire un canale di comunicazione *unidirezionale e sequenziale in memoria* tra due processi consiste nel creare una *pipe*:

```
#include <unistd.h>
```

```
int pipe (int fildes[2])
```

- La chiamata ritorna zero in caso di successo, -1 in caso di errore.
- Il primo descrittore ([0]) viene usato per leggere, il secondo [1] per scrivere.
- NOTA: L'interfaccia è quella dei file, quindi sono applicabili le system call che utilizzano file descriptor.

```

/*****
MODULO: pipe.c
SCOPO: esempio di IPC mediante pipe
*****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int argc, char *argv[]) {
    int status, p[2];
    char buf[64];

    pipe (p);
    if ((status=fork()) == -1) /* errore */
        syserr (argv[0], "fork() fallita");
    else if (status == 0) { /* figlio */
        close (p[1]);

```



```
    if (read(p[0],buf,BUFSIZ) == -1)
        syserr (argv[0], "read() fallita");
    printf ("Figlio - ricevuto: %s\n", buf);
    exit(0);
} else { /* padre */
    close(p[0]);
    printf("Padre - invio nella pipe 'In bocca al lupo'\n");
    write(p[1], "In bocca al lupo", 17);
    wait(&status);
    exit(0);
}
}
```

Pipe e I/O

- Non è previsto l'accesso random (no `lseek`).
- La dimensione fisica delle pipe è limitata (dipendente dal sistema – BSD classico = 4K).
- L'operazione di `write` su una pipe è **atomica**
- La scrittura di un numero di Byte superiore a questo numero:
 - Blocca il processo scrivente fino a che non si libera spazio
 - la `write` viene eseguita a “pezzi”, con risultati non prevedibili (es. più processi che scrivono)
- La `read` si blocca su pipe vuota e si sblocca non appena un Byte è disponibile (anche se ci sono meno dati di quelli attesi!)
- Chiusura prematura di un estremo della pipe:
 - scrittura: le `read` ritornano 0.
 - lettura: i processi in scrittura ricevono il segnale `SIGPIPE` (broken pipe)

Pipe e comandi

```
/******  
MODULO: pipe2.c  
SCOPO: Realizzare il comando "ps | sort"  
*****/  
#include <sys/types.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
main ()  
{  
    pid_t pid;  
    int pipefd[2];  
  
    pipe (pipefd);  
    if ((pid = fork()) == 0) { /* figlio */  
        close(1);          /* close stdout */
```

```
        dup (pipefd[1]);
        close (pipefd[0]);
        execlp ("ps", "ps", NULL);
    }
else if (pid > 0) { /* padre */
    close(0); /* close stdin */
    dup (pipefd[0]);
    close (pipefd[1]);
    execlp ("sort", "sort", NULL);
}
}
```

Pipe e I/O non bloccante

- E' possibile forzare il comportamento di `write` e `read` rimuovendo la limitazione del bloccaggio.
- Realizzato tipicamente con `fcntl` per impostare la flag `O_NDELAY` sul corrispondente file descriptor (0 o 1)
- Utile per implementare meccanismi di polling su pipe.
 - Se la flag `O_NDELAY` è impostata, una `write` su una pipe piena ritorna subito 0, e una `read` su una pipe vuota ritorna subito 0.

Pipe

- Limitazioni
 - possono essere stabilite soltanto tra processi *imparentati* (es., un processo ed un suo “progenitore”, o tra due discendenti di un unico processo)
 - Non sono permanenti e sono distrutte quando il processo che le crea termina
- Soluzione: assegnare un nome *unico* alla pipe: *named pipe* dette anche FIFO.
- Funzionamento identico, ma il riferimento avviene attraverso il nome anziché attraverso il file descriptor.
- Esistono fisicamente su disco e devono essere rimossi esplicitamente con `unlink`

Named Pipe (FIFO)

- Si creano con `mknod` (l'argomento `dev` viene ignorato)
`mknod (nome, S_IFIFO|mode, 0)`
 - valore di ritorno: 0 in caso di successo, -1 in caso di errore.
- apertura, lettura/scrittura, chiusura avvengono come per un normale file
- Possono essere usate da processi non in relazione, in quanto il nome del file è unico nel sistema.
- Le operazioni di I/O su FIFO sono atomiche
- I/O normalmente bloccante, ma è possibile aprire (con `open` e flag `O_NDELAY`) un FIFO in modo non bloccante. In tal modo sia `read` che `write` saranno non bloccanti.
- Utilizzabile anche la funzione `mkfifo()`

```
/*  
*****  
*/
```

```
MODULO: fifo.c
```

```
SCOPO: esempio di IPC mediante named pipe
```

USO: Lanciare due copie del programma su due shell separate, una con flag -0 e l'altra con flag -1.

Lanciare prima quella con flag -1 che crea la fifo.

La copia del programma con flag -0 leggerà quanto scritto dalla copia con flag -1.

```
*****  
*/
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <fcntl.h>
```

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

```
#include <sys/stat.h>
```

```
#include <unistd.h>
```



```

int main (int argc, char *argv[]) {
    int i,fd;
    char buf[64];

    if (argc != 2) {
        printf("Usage: fifo.x -[0|1]\n\t -0 to read\n\t -1 to write\n");
        exit(1);
    } else if (strncmp(argv[1], "-1", 2)==0){
        if (mknod("fifo",S_IFIFO|0777,0)== -1) {
            perror("mknod");
            exit(1);
        }
        if ((fd = open("fifo",O_WRONLY))== -1){
            perror("FIFO: -1");
            unlink("fifo");
            exit(1);
        }
    }
}

```

```

} else if (strncmp(argv[1], "-0", 2)==0){
    if ((fd = open("fifo",O_RDONLY))==-1){
        perror("FIFO: -1");
        unlink("fifo");
        exit(1);
    }
} else {
    printf("Wrong parameter: %s\n", argv[1]);
    unlink("fifo");
    exit(1);
}
for (i=0;i<20;i++) {
    if (strncmp(argv[1], "-1", 2)==0){
        write(fd,"HELLO",6);
        printf("Written HELLO %d \n", i);
    } else {
        read(fd,buf,6);
        printf("Read %s %d\n",buf,i);}}}}

```

Meccanismi di IPC Avanzati

- Cosiddette IPC SystemV:
 - Code di messaggi
 - Memoria condivisa
 - Semafori
- Disponibili API alternative (es. POSIX IPC).
Particolarmente usate quelle per semafori!

Meccanismi di IPC Avanzati – (cont.)

- Caratteristiche comuni:
 - Una primitiva “get” per creare una nuova entry o recuperarne una esistente
 - Una primitiva “ctl” (control) per:
 - * verificare lo stato di una entry,
 - * cambiare lo stato di una entry
 - * rimuovere una entry.

Meccanismi di IPC Avanzati – (cont.)

- La primitiva “get” specifica due informazioni associate ad ogni entry:
 - Una *chiave*, usata per la creazione dell’oggetto:
 - * Valore intero arbitrario;
 - * Valore intero generato con la funzione `key_t ftok(char *path, char id);`
 - dato un nome di file esistente ed un carattere.
 - Utile per evitare conflitti tra processi diversi;
 - * `IPC_PRIVATE`, costante usata per creare una nuova entry
 - Dei *flag* di utilizzo:
 - * `IPC_CREAT`: si crea una nuova entry se la chiave non esiste
 - * `IPC_CREATE + IPC_EXCL`: si crea una nuova entry ad uso esclusivo da parte del processo
 - * Permessi relativi all’accesso (tipo `rw-rw-rw`)

Meccanismi di IPC Avanzati – (cont.)

- L'identificatore ritornato dalla “get” (se diverso da -1) è un descrittore utilizzabile dalle altre system call
- La creazione di un oggetto IPC causa anche l'inizializzazione di:
 - una struttura dati, diversa per i vari tipi di oggetto contenente informazioni su
 - * UID, GID
 - * PID dell'ultimo processo che l'ha modificata
 - * Tempi dell'ultimo accesso o modifica
 - una struttura di permessi ipc_perm, contenente:

```
    ushort cuid;    /* creator user id */
    ushort cgid;    /* creator group id */
    ushort uid; /* owner user id */
    ushort gid; /* owner group id */
    ushort mode; /* r/w permissions */
```

Code di Messaggi

- Un messaggio è una unità di informazione di dimensione variabile, senza un formato predefinito
- Vengono memorizzati nelle *code*, che vengono individuate dalla chiave

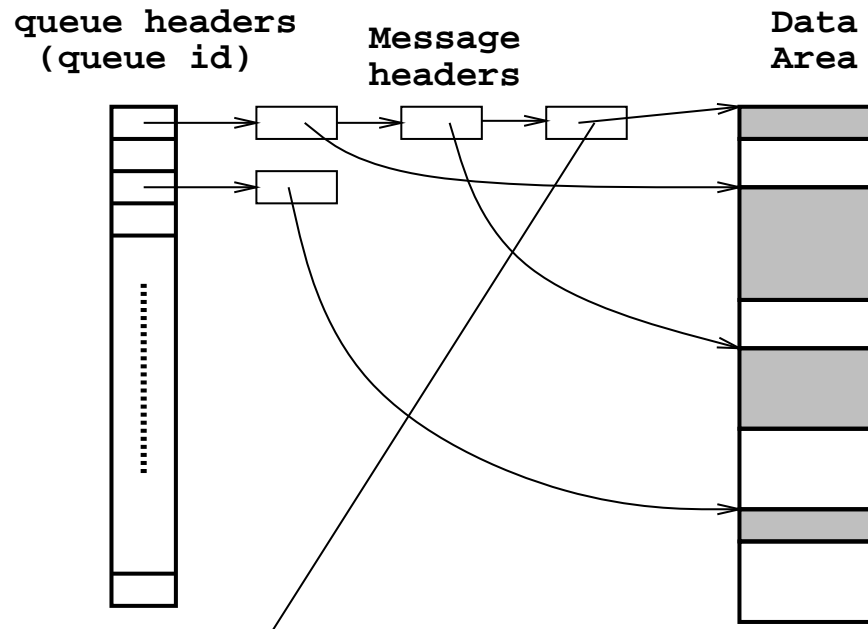
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgget (key_t key, int flag)
```

- Crea una coda di messaggi data la chiave *key* (di tipo `long`) se
 - `key = IPC_PRIVATE`, oppure
 - `key` non è definita, e `flag & IPC_CREAT` è vero.

Code di Messaggi – (cont.)

- I permessi associati ad una entry vengono specificati nei 9 LSb del campo flag (cfr. creat).
- Hanno significato solo i flag di lettura e scrittura.
- Struttura delle code di messaggi:



Code di Messaggi: Gestione

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

```
int msgctl (int id, int command, struct msqid_ds *buffer)
```

- id è il descrittore ritornato da msgget
- command:
 - IPC_RMID Cancella la coda (buffer non usato)
 - IPC_STAT Ritorna informazioni relative alla coda nella struttura puntata da buffer (contiene info su UID, GID, stato della coda)
 - IPC_SET Modifica un sottoinsieme dei campi contenuti nella struct

Code di Messaggi: Gestione – (Cont.)

- `buffer` è un puntatore a una struttura definita in `sys/msg.h` contenente (campi utili):

```
struct msqid_ds
{
    struct ipc_perm msg_perm;      /* permissions (rwxrwxrwx) */
    __time_t msg_stime;           /* time of last msgsnd command */
    __time_t msg_rtime;          /* time of last msgrcv command */
    __time_t msg_ctime;          /* time of last change */
    unsigned long int __msg_cbytes; /* current number of bytes on queue */
    msgqnum_t msg_qnum;           /* number of messages currently on queue */
    msglen_t msg_qbytes;          /* max number of bytes allowed on queue */
    __pid_t msg_lspid;            /* pid of last msgsnd() */
    __pid_t msg_lrpid;            /* pid of last msgrcv() */
};
```

Code di Messaggi: Scambio di Informazione

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int id,struct msgbuf *msg,size_t size,int flag)
int msgrcv(int id,struct msgbuf *msg,size_t size,long type,int flag);
```

- id è il descrittore ritornato da msgget
- Struttura dei messaggi:

```
struct msgbuf {
    long    mtype;    /* message type */
    char    mtext[1]; /* message text */
};
```

- Da interpretare come "template" di messaggio!
- In pratica, si usa una struct costruita dall'utente

Code di Messaggi: Scambio di Informazione – (cont.)

- flag:

IPC_NOWAIT (msgsnd e msgrcv) non si blocca se non ci
sono messaggi da leggere

MSG_NOERROR (msgrcv) tronca i messaggi a size
byte senza errore

- type indica quale messaggio prelevare:

0 Il primo messaggio, indipendentemente dal tipo

> 0 Il primo messaggio di tipo type

< 0 Il primo messaggio con tipo più “vicino”
al valore assoluto di type

```

/*****
PROCESSO SERVER
*****/
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>

#define MSGKEY 75
#define MSGTYPE 1

int main (int argc, char **argv) {
    key_t msgkey;
    int msgid, pid;
    struct msg {

```

```

    int mtype;
    char mtext[256];
} Message;

if ((msgid = msgget(MSGKEY, (0666|IPC_CREAT|IPC_EXCL))) == -1) {
    perror(argv[0]);
}

/* leggo dalla coda */
msgrcv(msgid, &Message, sizeof(Message.mtext), MSGTYPE, 0); /* WAIT */
printf("Received from client: %s\n", Message.mtext);

/* scrivo in un messaggio il pid e lo invio*/
pid = getpid();
sprintf(Message.mtext, "%d", pid);
Message.mtype = MSGTYPE;
msgsnd(msgid, &Message, sizeof(Message.mtext), 0); /* WAIT */
}

```

```

/*****
PROCESSO CLIENT
*****/
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>

#define MSGKEY 75
#define MSGTYPE 1

int main (int argc, char **argv) {
    key_t msgkey;
    int msgid, pid;
    struct msg {

```

```

    int mtype;
    char mtext[256];
} Message;

if ((msgid = msgget(MSGKEY,0666)) == -1) {
    perror(argv[0]);
}
/* scrivo il PID in un messaggio e lo
   invio nella coda */
pid = getpid();
sprintf(Message.mtext,"%d",pid);
Message.mtype = MSGTYPE;
msgsnd(msgid,&Message,sizeof(Message.mtext),0); /* WAIT */

/* Ricevo dalla coda il messaggio del server */
msgrcv(msgid,&Message,sizeof(Message.mtext),MSGTYPE,0); /* WAIT */
printf("Received message from server: %s\n",Message.mtext);
}

```



```

/*****
MODULO: msgctl.c
SCOPO: Illustrare il funz. di msgctl()
*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <time.h>

void do_msgctl();
char warning_message[] = "If you remove read permission for
                          yourself, this program will fail frequently!";

int main(int argc, char* argv[]) {
    struct msqid_ds buf; /*buffer per msgctl()*/
    int cmd; /* comando per msgctl() */

```

```

int msqid; /* ID della coda da passare a msgctl() */

if (argc!=2){
    printf("Usage: msgctl.x <msgid>\n");
    exit(1);
}

msqid = atoi(argv[1]);

fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
fprintf(stderr, "\nScegliere l'opzione desiderata: ");

scanf("%i", &cmd);

switch (cmd) {
case IPC_SET:

```

```

fprintf(stderr, "Prima della IPC_SET,
                controlla i valori correnti:");
/* notare: non e' stato inserito il break, quindi di seguito
vengono eseguite le istruzioni del case IPC_STAT */

case IPC_STAT:
do_msgctl(msqid, IPC_STAT, &buf);
fprintf(stderr, "msg_perm.uid = %d\n", buf.msg_perm.uid);
fprintf(stderr, "msg_perm.gid = %d\n", buf.msg_perm.gid);
fprintf(stderr, "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
fprintf(stderr, "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
fprintf(stderr, "msg_perm.mode = %#o, ", buf.msg_perm.mode);
fprintf(stderr, "access permissions = %#o\n",
        buf.msg_perm.mode & 0777);
fprintf(stderr, "msg_cbytes = %d\n", buf.msg_cbytes);
fprintf(stderr, "msg_qbytes = %d\n", buf.msg_qbytes);
fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
fprintf(stderr, "msg_lspid = %d\n", buf.msg_lspid);

```

```

fprintf(stderr, "msg_lrpid = %d\n", buf.msg_lrpid);
if (buf.msg_stime) {
    fprintf(stderr, "msg_stime = %s\n", ctime(&buf.msg_stime));
}
if (buf.msg_rtime) {
    fprintf(stderr, "msg_rtime = %s\n", ctime(&buf.msg_rtime));
}
fprintf(stderr, "msg_ctime = %s", ctime(&buf.msg_ctime));

/* se il comando originario era IPC_STAT allora esco
dal case, altrimenti proseguo con le operazioni
specifiche per la modifica dei parametri della coda.
*/
if (cmd == IPC_STAT)
    break;

/* Modifichiamo alcuni parametri della coda */
fprintf(stderr, "Enter msg_perm.uid: ");

```

```

scanf ("%hi", &buf.msg_perm.uid);
fprintf(stderr, "Enter msg_perm.gid: ");
scanf("%hi", &buf.msg_perm.gid);
fprintf(stderr, "%s\n", warning_message);
fprintf(stderr, "Enter msg_perm.mode: ");
scanf("%hi", &buf.msg_perm.mode);
fprintf(stderr, "Enter msg_qbytes: ");
scanf("%hi", &buf.msg_qbytes);
do_msgctl(msqid, IPC_SET, &buf);
break;

case IPC_RMID:
default:
    /* Rimuove la coda di messaggi. */
    do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
    break;
}
exit(0);

```

```

}

void do_msgctl(int msqid, int cmd, struct msqid_ds* buf) {
    int rtrn; /* per memorizzare il valore di ritorno della msgctl() */

    fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d, %s)\n",
             msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
    rtrn = msgctl(msqid, cmd, buf);

    if (rtrn == -1) {
        perror("msgctl: msgctl failed");
        exit(1);
    } else {
        fprintf(stderr, "msgctl: msgctl returned %d\n", rtrn);
    }
}
}

```

Analisi esempi

Per provare gli esempi precedenti:

- lanciare il programma server
- lanciare il programma client su una shell separata
 - client e server si scambieranno un messaggio e termineranno lasciando la coda di messaggi attiva.
- eseguire il comando `ipcs -q` per verificare che effettivamente esista una coda attiva.
- lanciare il programma `msgctl.c` passandogli come parametro il valore *msquid* visualizzato dal comando `ipcs -q`
- provare le varie opzioni del programma `msgctl.c`, in particolare usare `IPC_SET` per variare le caratteristiche della coda e `IPC_RMID` per rimuovere la coda

Memoria Condivisa

- Due o più processi possono comunicare anche condividendo una parte del loro spazio di indirizzamento (virtuale).
- Questo spazio condiviso è detto *memoria condivisa* (*shared memory*), e la comunicazione avviene scrivendo e leggendo questa parte di memoria

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

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

```
shm_id shmget(key_t key, int size, int flags);
```

- I parametri hanno lo stesso significato di quelli utilizzati da `msgget`.
- `size` indica la dimensione in byte della regione condivisa.

Memoria Condivisa – (cont.)

- Una volta creata, l'area di memoria non è subito disponibile
- Deve essere *collegata* all'area dati dei processi che vogliono utilizzarla.

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

```
char *shmat (int shmid, char *shmaddr, int flag)
```

- `shmaddr` indica l'indirizzo virtuale dove il processo vuole attaccare il segmento di memoria condivisa.
- Il valore di ritorno rappresenta l'indirizzo di memoria condivisa effettivamente risultante

Memoria Condivisa – (cont.)

- In base ai valori di `flag` e di `shmaddr` si determina il punto di attacco del segmento:
 `shmaddr = 0 && (flag & SHM_RND)` al primo indirizzo disponibile
 `shmaddr != 0 && !(flag & SHM_RND)` all'indirizzo indicato da `shmaddr`
- Il segmento è attaccato in lettura se `flag & SHM_RDONLY` è vero, contemporaneamente in lettura e scrittura altrimenti.
- Un segmento attaccato in precedenza può essere “staccato” (*detached*) con `shmdt`

```
int shmdt (char *shmaddr)
```

dove `shmaddr` è l'indirizzo che individua il segmento di memoria condivisa.

- Non viene passato l'ID della regione perchè è possibile avere più aree di memoria identificate dallo stesso ID (cioè attaccate ad indirizzi diversi);

Memoria Condivisa: Gestione

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

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

- shmid è il descrittore ritornato da shmget
- Valori di cmd:
 - IPC_RMID Cancella il segm. di memoria condivisa
 - IPC_STAT Ritorna informazioni relative all'area di memoria condivisa nella struttura puntata da buffer (contiene info su UID, GID, permessi, stato della memoria)
 - IPC_SET Modifica un sottoinsieme dei campi contenuti nella struct (UID, GID, permessi)
 - SHM_LOCK Impedisce che il segmento venga swappato o paginato

Memoria Condivisa: Gestione – (Cont.)

- `buffer` è un puntatore a una struttura definita in `sys/shm.h` contenente:

```
struct shmid_ds {
    struct ipc_perm shm_perm;           /* operation permission struct */
    size_t shm_segsz;                  /* size of segment in bytes */
    __time_t shm_atime;                /* time of last shmat() */
    __time_t shm_dtime;                /* time of last shmdt() */
    __time_t shm_ctime;                /* time of last change by shmctl() */
    __pid_t shm_cpid;                  /* pid of creator */
    __pid_t shm_lpid;                  /* pid of last shmop */
    shmatt_t shm_nattch;                /* number of current attaches */
};
```

```

/*****
MODULO: shmctl.c
SCOPO: Illustrare il funz. di shmctl()
USO: Lanciare il programma e fornire l'ID
      di un segmento di memoria condivisa
      precedentemente creato.
      Usare il comando della shell ipcs per vedere
      i segmenti di memoria condivisa attivi
*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>

void do_shmctl();

int main(int argc, char *argv[]) {
    int cmd; /* comando per shmctl() */

```

```

int shmid; /* ID dell'area di memoria condivisa*/
struct shmid_ds shmid_ds; /* struttura per il controllo
                           dell'area di memoria condivisa*/

fprintf(stderr, "Inserire l'ID del segmento di memoria condiviso: ");
scanf("%i", &shmid);

fprintf(stderr, "Comandi validi:\n");
fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
fprintf(stderr, "Scegliere il comando desiderato: ");
scanf("%i", &cmd);

switch (cmd) {
    case IPC_STAT:
        /* Le informazioni sullo stato della memoria condivisa
           vengono recuperate con la chiamata alla funzione

```

```

do_shmctl(shmid, cmd, &shmid_ds) eseguita al termine
del case. Le informazioni saranno inserite nella
struttura shm_ds*/
break;
case IPC_SET:
/*visualizzazione dello stato attuale della memoria */
do_shmctl(shmid, IPC_STAT, &shmid_ds);
/* Lettura da tastiera dei valori di UID, GID, e permessi da settare */
fprintf(stderr, "\nInserire shm_perm.uid: ");
scanf("%hi", &shmid_ds.shm_perm.uid);
fprintf(stderr, "Inserire shm_perm.gid: ");
scanf("%hi", &shmid_ds.shm_perm.gid);
fprintf(stderr, "N.B.: Mantieni il permesso di lettura per te stesso!\n");
fprintf(stderr, "Inserire shm_perm.mode: ");
scanf("%hi", &shmid_ds.shm_perm.mode);
break;
case IPC_RMID: /* Rimuove il segmento */
break;
case SHM_LOCK: /* Esegui il lock sul segmento */
break;

```

```

    case SHM_UNLOCK: /* Esegui unlock sul segmento */
        break;
    default: /* Comando sconosciuto passato a do_shmctl */
        break;
}
/* La funzione do_shmctl esegue il comando scelto dall'utente */
do_shmctl(shmid, cmd, &shmctl_ds);
exit(0);
}

void do_shmctl(int shmid, int cmd, struct shmctl_ds* buf) {
    int rtrn; /* valore di ritorno della shmctl */

    fprintf(stderr, "shmctl: Chiamo shmctl(%d, %d, buf)\n", shmid, cmd);
    if (cmd == IPC_SET) {
        fprintf(stderr, "\tbuf->shm_perm.uid == %d\n", buf->shm_perm.uid);
        fprintf(stderr, "\tbuf->shm_perm.gid == %d\n", buf->shm_perm.gid);
        fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n", buf->shm_perm.mode);
    }
    if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {

```



```

    perror("shmctl: shmctl fallita.");
    exit(1);
} else {
    fprintf(stderr, "shmctl: shmctl ha ritornato %d\n", rtrn);
}
if (cmd != IPC_STAT && cmd != IPC_SET)
    return; /* ritorno perche' il comando e' stato eseguito e non
            devo visualizzare nessuna informazione sullo stato */

/* Stampa lo stato corrente del segmento */
fprintf(stderr, "\nCurrent status:\n");
fprintf(stderr, "\tshm_perm.uid = %d\n", buf->shm_perm.uid);
fprintf(stderr, "\tshm_perm.gid = %d\n", buf->shm_perm.gid);
fprintf(stderr, "\tshm_perm.cuid = %d\n", buf->shm_perm.cuid);
fprintf(stderr, "\tshm_perm.cgid = %d\n", buf->shm_perm.cgid);
fprintf(stderr, "\tshm_perm.mode = %#o\n", buf->shm_perm.mode);
fprintf(stderr, "\tshm_perm.key = %#x\n", buf->shm_perm.__key);
fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);

```

```
fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
if (buf->shm_atime)
    fprintf(stderr, "\tshm_atime = %s", ctime(&buf->shm_atime));
if (buf->shm_dtime)
    fprintf(stderr, "\tshm_dtime = %s", ctime(&buf->shm_dtime));
fprintf(stderr, "\tshm_ctime = %s", ctime(&buf->shm_ctime));
}
```

```

/*****
NOME: shm1.c
SCOPO: ‘‘attaccare’’ due volte un’area di memoria condivisa
        Ricordarsi di rimuovere la memoria condivisa al termine
        del programma lanciando shmctl.c oppure tramite il comando
        della shell ipcrm.
*****/
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define K 1
#define SHMKEY 75
#define N 10

int shmids;
int main (int argc, char **argv) {
    int i, *pint;
    char *addr1, *addr2;

```

```

/* Creao il segmento condiviso di dimensione 128*K byte */
shmmid = shmget(SHMKEY, 128*K, 0777|IPC_CREAT);
/* Attacco il segmento in due zone diverse */
addr1 = shmat(shmmid,0,0);
addr2 = shmat(shmmid,0,0);

printf("Address1 = 0x%x\t Address2 = 0x%x\t\n", addr1,addr2);

/* scrivo nella regione 1 */
pint = (int*)addr1;
for (i=0;i<N;i++) {
    *pint = i;
    printf("Writing: Index %4d\tValue: %4d\tAddress: 0x%x\n",
           i,*pint,pint);
    pint++;
}
/* leggo dalla regione 2 */
pint = (int*)addr2;
for (i=0;i<N;i++) {

```

```
    printf("Reading: Index %4d\tValue: %4d\tAddress: 0x%x\n",  
          i,*pint,pint);  
    pint++;  
}  
}
```

```
/******
```

```
NOME: shm2.c
```

```
SCOPO: ‘‘attaccarsi’’ ad un area di memoria condivisa
```

```
USO: lanciare prima il programma shm1.c per creare la memoria  
condivisa.
```

```
Ricordarsi di rimuovere la memoria condivisa al termine  
del programma lanciando shmctl.c oppure tramite il comando  
della shell ipcrm.
```

```
*****/
```

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

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

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

```
#include <stdio.h>
```

```
#define K 1
```

```
#define N 20
```

```
#define SHMKEY 75
```

```
int shmidx;
```

```
int main (int argc, char **argv) {
```

```
    int i, *pint;
```

```
char *addr;

/*mi attacco alla regione creata dal programma shm1.c*/
shmmid = shmget(SHMKEY, 128*K, 0777);
addr = shmat(shmmid,0,0);
printf("Address = 0x%x\n", addr);
pint = (int*) addr;
/* leggo dalla regione attaccata in precedenza */
for (i=0;i<N;i++) {
    printf("Reading: (Value = %4d)\n",*pint++);
}
}
```

```

/*****
MODULO: shm_server.c
SCOPO: server memoria condivisa
USO: lanciare il programma shm_server.c in una shell
     e il programma shm_client.c in un'altra shell
     Ricordarsi di rimuovere la memoria condivisa creata
     dal server al termine del programma lanciando shmctl.c
     oppure tramite il comando della shell ipcrm.
*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27

int main(int argc, char *argv[]) {
    char c;
    int shmid;

```



```

key_t key;
char *shm, *s;
key = 5678;

/* Creo il segmento */
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
    perror("shmget");
    exit(1);
}
/* Attacco il segmento all'area dati del processo */
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
}
s = shm;
for (c = 'a'; c <= 'z'; c++)
    *s++ = c;

*s = NULL;

```

```
while (*shm != '*')
    sleep(1);

printf("Received '*'. Exiting...\n");
exit(0);
}
```

```

/*****
MODULO: shm_client.c
SCOPO: client memoria condivisa
USO: lanciare il programma shm_server.c in una shell
     e il programma shm_client.c in un'altra shell
     Ricordarsi di rimuovere la memoria condivisa creata
     dal server al termine del programma lanciando shmctl.c
     oppure tramite il comando della shell ipcrm.
*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27

int main(int argc, char *argv[]) {
    int shmid;
    key_t key;

```

```

char *shm, *s;
key = 5678;

/* Recupero il segmento creato dal server */
if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
    perror("shmget");
    exit(1);
}

/* Attacco il segmento all'area dati del processo */
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
}

/* Stampo il contenuto della memoria */
printf("Contenuto del segmento condiviso con il server: ");
for (s = shm; *s != NULL; s++)
    putchar(*s);

```

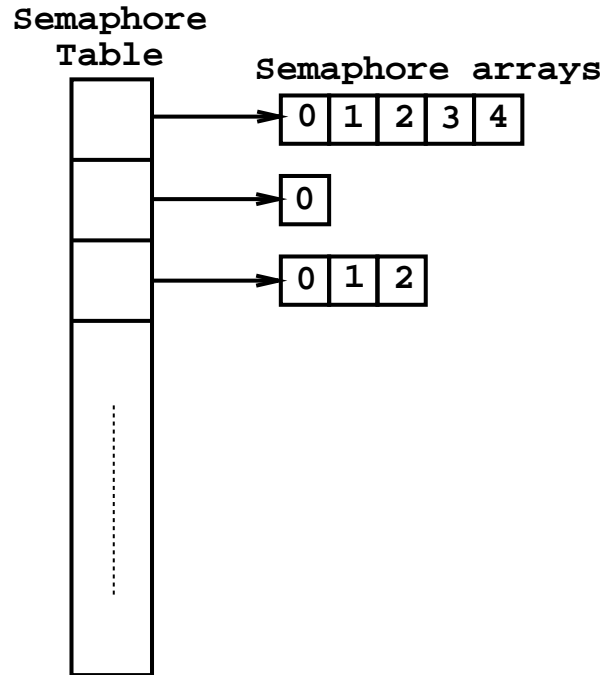
```
    putchar('\n');  
    sleep(3);  
    /* ritorno * al server affinche' possa terminare */  
    *shm = '*';  
    exit(0);  
}
```

Sincronizzazione tra Processi

- I semafori permettono la sincronizzazione dell'esecuzione di due o più processi
 - Sincronizzazione su un dato valore
 - Mutua esclusione
- Semafori SystemV:
 - piuttosto diversi da semafori classici
 - “pesanti” dal punto di vista della gestione
- Disponibili varie API (per es. POSIX semaphores)

Semafori (SystemV API)

- Non è possibile allocare un singolo semaforo, ma è necessario crearne un insieme (vettore di semafori)
- Struttura interna di un semaforo



Semafori (SystemV API)

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

```
int semid = semget(int key, int count, short flags);
```

- I valori di **key** e di **flags** sono identici al caso delle code di messaggi e della shared memory.
- **count** è il numero di semafori identificati dal **semid** (quanti semafori sono contenuti nel vettore).
 - Il primo vettore del semaforo ha indice 0
- NOTA: I semafori hanno sempre valore iniziale = 0 \Rightarrow potenziali deadlock durante la creazione!

Operazioni su Semafori

```
int semctl(int semid, int semnum, int command, union semun *args);
union semun {
    int val;
    struct semid_ds* buffer;
    unsigned short *array;
};
```

- Operazioni (**command**):

IPC_RMID* Rimuove il set di semafori

IPC_SET@ Modifica il set di semafori

IPC_STAT@ Statistiche sul set di semafori

GETVAL* legge il valore del semaforo **semnum** in **args.val**

GETALL# legge tutti i valori in **args.array**

SETVAL* assegna il valore del semaforo **semnum** in **args.val**

SETALL# assegna tutti i semafori con i valori in **args.array**

GETPID* Valore di PID dell'ultimo processo che ha fatto operazioni

GETNCNT* numero di processi in attesa che un semaforo aumenti

GETZCNT* numero di processi in attesa che un semaforo diventi 0

Operazioni su Semafori

- Operazioni con "*" : `arg = arg.val` (intero)
- Operazioni con "#": `arg = arg.array` (puntatore a array di `short`)
- Operazioni con "@": `arg = arg.buf` (puntatore a buffer di tipo `semid_ds`)
- `buffer` è un puntatore ad una struttura `semid_ds` definita in `sys/sem.h`:

```
struct semid_ds {
    struct ipc_perm sem_perm;           /* operation permission struct */
    time_t sem_otime;                  /* Last semop time */
    time_t sem_ctime;                  /* Last change time */
    unsigned short sem_nsems;          /* No. of semaphores */
};
```

- La union `semun` deve essere definita nel codice del processo che chiama la `semctl()`

Operazioni su Semafori

La `semctl()` ritorna valori diversi a seconda del comando eseguito :

- **GETNCNT**: il numero di processi in attesa che il semaforo **semnum** venga incrementato
- **GETPID**: il PID dell'ultimo processo che ha effettuato `semop()` sul semaforo **semnum** dell'insieme
- **GETVAL**: il valore del semaforo **semnum** dell'insieme
- **GETZCNT**: il numero di processi in attesa che il semaforo **semnum** diventi 0

```

/*****
MODULO: semctl.c
SCOPO: Illustrare il funz. di semctl()
USO:   prima di lanciare semctl.c() creare un semaforo usando un altro
        programma (ad esempio sem.x)
*****/
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <time.h>

struct semid_ds semid_ds;

/* explicit declaration required */
union semun {
    int val;
    struct semid_ds* buf;
    unsigned short int *array;
}

```

```

    struct seminfo *__buf;
} arg;

void do_semctl(int semid, int semnum, int cmd, union semun arg);
void do_stat();
char warning_message[] = "If you remove read permission for yourself,
                          this program will fail frequently!";

int main(int argc, char *argv[]) {
    union semun arg; /* union to pass to semctl() */
    int cmd;         /* command to give to semctl() */
    int i;
    int semid;       /* semid to pass to semctl() */
    int semnum;      /* semnum to pass to semctl() */

    fprintf(stderr, "Enter semid value: ");
    scanf("%i", &semid);

    fprintf(stderr, "Valid semctl cmd values are:\n");
    fprintf(stderr, "\tGETALL = %d\n", GETALL);

```

```

fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
fprintf(stderr, "\tGETPID = %d\n", GETPID);
fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
fprintf(stderr, "\tSETALL = %d\n", SETALL);
fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
fprintf(stderr, "\nEnter cmd: ");
scanf("%i", &cmd);

/* Do some setup operations needed by multiple commands. */
switch (cmd) {
    case GETVAL:
    case SETVAL:
    case GETNCNT:
    case GETZCNT:
        /* Get the semaphore number for these commands. */
        fprintf(stderr, "\nEnter semnum value: ");

```

```

    scanf("%i", &semnum);
    break;
case GETALL:
case SETALL:
    /* Allocate a buffer for the semaphore values. */
    fprintf(stderr, "Get number of semaphores in the set.\n");
    arg.buf = &semid_ds;
    /* when IPC_STAT is called the second argument of semctl() is ignored.
       IPC_STAT is called to retrieve info semid_ds on the semaphore set */
    do_semctl(semid, 0, IPC_STAT, arg);
    if (arg.array =
        (u_short *)malloc((unsigned) (semid_ds.sem_nsems * sizeof(u_short)))) {
        /* Break out if you got what you needed */
        break;
    }
    fprintf(stderr, "semctl: unable to allocate space for %d values\n",
            semid_ds.sem_nsems);
    exit(2);
}

```

```

/* Get the rest of the arguments needed for the specified command. */
switch (cmd) {
case SETVAL:
    /* Set value of one semaphore. */
    fprintf(stderr, "\nEnter semaphore value: ");
    scanf("%i", &arg.val);
    do_semctl(semid, semnum, SETVAL, arg);
    /* Fall through to verify the result. */
    fprintf(stderr, "Executing semctl GETVAL command to verify results...\n");
case GETVAL:
    /* Get value of one semaphore. */
    arg.val = 0;
    do_semctl(semid, semnum, GETVAL, arg);
    break;
case GETPID:
    /* Get PID of the last process that successfully completes a
    semctl(SETVAL), semctl(SETALL), or semop() on the semaphore. */
    arg.val = 0;
    do_semctl(semid, 0, GETPID, arg);
    break;

```



```

case GETNCNT:
    /* Get number of processes waiting for semaphore value to increase. */
    arg.val = 0;
    do_semctl(semid, semnum, GETNCNT, arg);
    break;
case GETZCNT:
    /* Get number of processes waiting for semaphore value to become zero. */
    arg.val = 0;
    do_semctl(semid, semnum, GETZCNT, arg);
    break;
case SETALL:
    /* Set the values of all semaphores in the set. */
    fprintf(stderr, "There are %d semaphores in the set.\n", semid_ds.sem_nsems);
    fprintf(stderr, "Enter semaphore values:\n");
    for (i = 0; i < semid_ds.sem_nsems; i++) {
        fprintf(stderr, "Semaphore %d: ", i);
        scanf("%hi", &arg.array[i]);
    }
    do_semctl(semid, 0, SETALL, arg);
    /* Fall through to verify the results. */

```

```

    fprintf(stderr, "Do semctl GETALL command to verify results.\n");
case GETALL:
    /* Get and print the values of all semaphores in the set.*/
    do_semctl(semid, 0, GETALL, arg);
    fprintf(stderr, "The values of the %d semaphores are:\n", semid_ds.sem_nsems);
    for (i = 0; i < semid_ds.sem_nsems; i++)
        fprintf(stderr, "%d ", arg.array[i]);
    fprintf(stderr, "\n");
    break;
case IPC_SET:
    /* Modify mode and/or ownership. */
    arg.buf = &semid_ds;
    do_semctl(semid, 0, IPC_STAT, arg);
    fprintf(stderr, "Status before IPC_SET:\n");
    do_stat();
    fprintf(stderr, "Enter sem_perm.uid value: ");
    scanf("%hi", &semid_ds.sem_perm.uid);
    fprintf(stderr, "Enter sem_perm.gid value: ");
    scanf("%hi", &semid_ds.sem_perm.gid);
    fprintf(stderr, "%s\n", warning_message);

```

```

    fprintf(stderr, "Enter sem_perm.mode value: ");
    scanf("%hi", &semid_ds.sem_perm.mode);
    do_semctl(semid, 0, IPC_SET, arg);
    /* Fall through to verify changes. */
    fprintf(stderr, "Status after IPC_SET:\n");
case IPC_STAT:
    /* Get and print current status. */
    arg.buf = &semid_ds;
    do_semctl(semid, 0, IPC_STAT, arg);
    do_stat();
    break;
case IPC_RMID:
    /* Remove the semaphore set. */
    arg.val = 0;
    do_semctl(semid, 0, IPC_RMID, arg);
    break;
default:
    /* Pass unknown command to semctl. */
    arg.val = 0;
    do_semctl(semid, 0, cmd, arg);

```

```
        break;
    }
    exit(0);
}
```

```
void do_semctl(int semid, int semnum, int cmd, union semun arg) {
    int i;

    fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d, ", semid, semnum, cmd);
    switch (cmd) {
        case GETALL:
            fprintf(stderr, "arg.array = %#x)\n", arg.array);
            break;
        case IPC_STAT:
        case IPC_SET:
            fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
            break;
        case SETALL:
            fprintf(stderr, "arg.array = [");
            for (i = 0; i < semid_ds.sem_nsems; i++) {
```

```

        fprintf(stderr, "%d", arg.array[i]);
        if (i < semid_ds.sem_nsems)
            fprintf(stderr, ", ");
    }
    fprintf(stderr, "])\n");
    break;
case SETVAL:
default:
    fprintf(stderr, "arg.val = %d)\n", arg.val);
    break;
}
/* call to semctl() */
i = semctl(semid, semnum, cmd, arg);
if (i == -1) {
    perror("semctl: semctl failed");
    exit(1);
}
fprintf(stderr, "semctl: semctl returned %d\n", i);
return;
}

```

```

void do_stat() {
    fprintf(stderr, "sem_perm.uid = %d\n", semid_ds.sem_perm.uid);
    fprintf(stderr, "sem_perm.gid = %d\n", semid_ds.sem_perm.gid);
    fprintf(stderr, "sem_perm.cuid = %d\n", semid_ds.sem_perm.cuid);
    fprintf(stderr, "sem_perm.cgid = %d\n", semid_ds.sem_perm.cgid);
    fprintf(stderr, "sem_perm.mode = %#o, ", semid_ds.sem_perm.mode);
    fprintf(stderr, "access permissions = %#o\n", semid_ds.sem_perm.mode & 0777);
    fprintf(stderr, "sem_nsems = %d\n", semid_ds.sem_nsems);
    fprintf(stderr, "sem_otime = %s",
            semid_ds.sem_otime ? ctime(&semid_ds.sem_otime) : "Not Set\n");
    fprintf(stderr, "sem_ctime = %s", ctime(&semid_ds.sem_ctime));
}

```

Operazioni su Semafori

```
int oldval = semop(int id, struct sembuf* ops, int count);
```

- Applica l'insieme **ops** di operazioni (in numero pari a **count**) al semaforo **id**.
- Le operazioni, contenute in un vettore opportunamente allocato, sono descritte dalla **struct sembuf**:

```
struct sembuf {  
    short sem_num;  
    short sem_op;  
    short sem_flg;  
};
```

- **sem_num**: semaforo su cui l'operazione (i-esima) viene applicata
- **sem_op**: l'operazione da applicare
- **sem_flg**: le modalità con cui l'operazione viene applicata

Operazioni su Semafori

- Valori di `sem_op`:

<code>< 0</code>	equivale a P (si blocca se <code>sem_val</code> \leq 0) Decrementa il semaforo della quantità <code>ops.sem_op</code>
<code>= 0</code>	In attesa che il valore del semaforo diventi 0
<code>> 0</code>	equivalente a V Incrementa il semaforo della quantità <code>ops.sem_op</code>

- Valori di `sem_flg`:

`IPC_NOWAIT` Per realizzare **P** e **V** non bloccanti
(comodo per realizzare *polling*)

`SEM_UNDO` Ripristina il vecchio valore quando termina
(serve nel caso di terminazioni precoci)

Operazioni su Semafori

NOTA BENE: L'insieme di operazioni inserite in una chiamata alla system call **semop()** viene eseguito in modo atomico. Se una delle operazioni non può essere eseguita, il comportamento della system call **semop()** dipende dalla flag **IPC_NOWAIT**:

- se **IPC_NOWAIT** è settato, **semop()** fallisce e ritorna -1;
- se **IPC_NOWAIT** non è settato, il processo viene bloccato;

```

/*****
MODULO: sem.c
SCOPO: Illustrare il funz. di semop() e semctl()
USO:   Lanciare il programma sem.x e quindi
       semop.x o semctl.x in una shell separata
*****/

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

#define KEY 14

int main(int argc, char *argv[]) {

    int semid; /* identificatore dei semafori */
    int i;

```

```

/* struttura per le operazioni sui semafori */
struct sembuf * sops = (struct sembuf *) malloc (sizeof(struct sembuf));

/*
    Creazione di due semafori con permessi di lettura e scrittura per tutti.
    Le flag ICP_CREAT e IPC_EXCL fanno si che la funzione semget ritorni
    errore se esiste gi un vettore di semafori con chiave KEY.
    Vedere man semget.
*/
if((semid = semget(KEY, 2, IPC_CREAT | IPC_EXCL | 0666)) == -1) {
    perror("semget");
    exit(1);
}

/* Inizializzo i due semafori a 1 */

/*
    Per eseguire operazioni ATOMICHE sui semafori si usa la funzione
    semop(). Vedere man semop.
    Alla funzione semop() vengono passati 3 argomenti:

```

- l'identificatore dell'array di semafori su cui eseguire l'operazione
- il puntatore alla struttura sembuf necessaria per eseguire le operazioni
- il numero di operazioni da eseguire

Per ogni operazione da eseguire necessario creare una struttura di tipo sembuf. La struttura contiene 3 campi:

- il numero del semaforo da utilizzare. Ricordare che la semget ritorna array di semafori.
- un intero N che rappresenta l'operazione da eseguire.
Se l'intero N e' > 0 il valore del semaforo viene incrementato di tale quantit. Se $N = 0$ la semop blocca il processo in attesa che il valore del semaforo diventi 0. Se $N < 0$ la semop blocca il processo in attesa che il valore del semaforo meno N sia maggiore o uguale a 0.
- una eventuale flag (IPC_NOWAIT o SEM_UNDO)
IPC_NOWAIT serve per avere semafori non bloccanti
SEM_UNDO serve per ripristinare il vecchio valore del semaforo in caso di terminazioni precoci.

*/

```
sops[0].sem_num = 0; /* semaforo 0 */
```

```
sops[0].sem_op = 1; /* incrementa di uno il valore del semaforo 0 */
sops[0].sem_flg = 0; /* nessuna flag settata */
/* esecuzione dell'operazione sul semaforo 0 */
semop(semid, sops, 1);
```

```
sops[0].sem_num = 1; /* semaforo 1 */
sops[0].sem_op = 1; /* incrementa di uno il valore del semaforo 1 */
sops[0].sem_flg = 0;
/* esecuzione dell'operazione sul semaforo 1 */
semop(semid, sops, 1);
```

```
printf("I semafori sono stati inizializzati a 1.\n");
printf("Lanciare il programma semctl.x o semop.x su un'altra shell
      e fornire semid=%d\n", semid);
```

```
while(1) {

    sops->sem_num = 0; /* semaforo 0 */
    sops->sem_op = 0; /* attende che il semaforo valga zero */
    sops->sem_flg = 0;
```

```

/* quando viene eseguita questa operazione il codice
   si blocca in attesa che il valore del semaforo 0 diventi 0 */
semop(semid, sops, 1);
/* Quando il semaforo diventa 0 stampo che il processo
   e' stato sbloccato */
printf("Sbloccato 1\n");

sops->sem_num = 1; /* semaforo 1 */
sops->sem_op = 0; /* attende che il semaforo valga zero */
sops->sem_flg = 0; /* notare il comportamento diverso del semaforo 1*/

/* quando viene eseguita questa operazione il codice
   si blocca in attesa che il valore del semaforo 1 diventi 0 */
semop(semid, sops, 1);
/* Quando il semaforo diventa 0 stampo che il processo
   e' stato sbloccato */
printf("          Sbloccato 2\n");

}
}

```

```

/*****
MODULO: semop.c
SCOPO: Illustrare il funz. di semop()
*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
    int          val;      /* Valore per SETVAL */
    struct semid_ds *buf;  /* Buffer per IPC_STAT, IPC_SET */
    unsigned short *array; /* Array per GETALL, SETALL */
    struct seminfo *_buf;  /* Buffer per IPC_INFO (specifico di Linux) */
};

int ask(int* semidp, struct sembuf **sopsp);

```

```

static struct semid_ds  semid_ds; /* stato del set di semafori */
static char error_mesg1[] = "semop: Non posso allocare spazio per un vettore
                             di %d valori.\n";
static char error_mesg2[] = "semop: Non posso allocare spazio per %d strutture
                             sembuf. \n";

int main(int argc, char* argv[]) {
    int    i;
    int    nsops;          /* numero di operazioni da fare */
    int    semid;         /* semid per il set di semafori */
    struct sembuf *sops; /* puntatore alle operazioni da eseguire */

    /* Cicla finche' l'utente vuole eseguire operazioni chiamando la funzione ask */
    while (nsops = ask(&semid, &sops)) {
        /* Inizializza il vettore di operazioni da eseguire */
        for (i = 0; i < nsops; i++) {
            fprintf(stderr, "\nInserire il valore per l'operazione %d di %d.\n", i+1, nsops);
            fprintf(stderr, "sem_num(i valori validi sono 0 <= sem_num < %d): ",
                    semid_ds.sem_nsems);
            scanf("%d", &sops[i].sem_num);
        }
    }
}

```



```

    fprintf(stderr, "sem_op: ");
    scanf("%d", &sops[i].sem_op);
    fprintf(stderr, "Possibili flag per sem_flg:\n");
    fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n", IPC_NOWAIT);
    fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n", SEM_UNDO);
    fprintf(stderr, "\tNESSUNO =\t%6d\n", 0);
    fprintf(stderr, "sem_flg: ");
    /* controllare cosa fa %i su man scanf */
    scanf("%i", &sops[i].sem_flg);
}

/* Ricapitola la chiamata da fare a semop() */
fprintf(stderr, "\nsemop: Chiamo semop(%d, &sops, %d) with:", semid, nsops);
for (i = 0; i < nsops; i++) {
    fprintf(stderr, "\nsops[%d].sem_num = %d, ", i, sops[i].sem_num);
    fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
    fprintf(stderr, "sem_flg = %o\n", sops[i].sem_flg);
}

/* Chiama la semop() e riporta il risultato */

```

```

    if ((i = semop(semid, sops, nsops)) == -1) {
        perror("semop: semop failed");
    } else {
        fprintf(stderr, "semop: valore di ritorno = %d\n", i);
    }
}
}

```

```

int ask(int *semidp, struct sembuf **sopsp) {
    static union semun arg;      /* argomento per semctl() */
    static int nsops = 0;       /* dimensione del vettore di sembuf */
    static int semid = -1;      /* semid del set di semafori */
    static struct sembuf *sops; /* puntatore al vettore di sembuf */
    int i;

    if (semid < 0) {
        /* Prima chiamata alla funzione ask()
           Recuperiamo semid dall'utente e lo stato corrente del set di semafori */
        fprintf(stderr, "Inserire semid del set di semafori su cui operare: ");
    }
}

```

```

scanf("%d", &semid);
*semidp = semid;

arg.buf = &semid_ds;

/* chiamata a semctl() */
if (semctl(semid, 0, IPC_STAT, arg) == -1) {
    perror("semop: semctl(IPC_STAT) fallita.");
    /* Notare che se semctl() fallisce, semid_ds viene riempita con 0,
       e successivi test per controllare il numero di semafori
       ritorneranno 0.
       Se invece semctl() va a buon fine, arg.buf verra'
       riempito con le informazioni relative al set di semafori */

    /* allocazione della memoria per un vettore di interi la cui
       dimensione dipende dal numero di semafori inclusi nel set */
} else if ((arg.array =
            (ushort *)malloc(sizeof(ushort) * semid_ds.sem_nsems)) == NULL) {
    fprintf(stderr, error_mesg1, semid_ds.sem_nsems);
    exit(1);
}

```

```

    }
}

/* Stampa i valori correnti dei semafori */
if (semid_ds.sem_nsems != 0) {
    fprintf(stderr, "Ci sono %d semaphores.\n", semid_ds.sem_nsems);
    /* Chiama la funzione semctl per recuperare i valori
       di tutti i semafori del set. Nel caso di GETALL il secondo
       argomento della semctl() viene ignorato e si utilizza il
       campo array della union semun arg */
    if (semctl(semid, 0, GETALL, arg) == -1) {
        perror("semop: semctl(GETALL) fallita");
    } else {
        fprintf(stderr, "I valori correnti dei semafori sono:");
        for (i = 0; i < semid_ds.sem_nsems; i++)
            fprintf(stderr, " %d", arg.array[i]);
        fprintf(stderr, "\n");
    }
}
}

```

```

/* Allocazione dello spazio per le operazioni che l'utente desidera eseguire */
fprintf(stderr, "Quante operazioni vuoi eseguire con la prossima semop()?\n");
fprintf(stderr, "Inserire 0 or control-D per uscire: ");
i = 0;
if (scanf("%d", &i) == EOF || i == 0)
    exit(0);
if (i > nsops) {
    if (nsops != 0) /* libero la memoria precedentemente allocata */
        free((char *)sops);
    nsops = i;
    /* Allocazione della memoria per le operazioni da eseguire*/
    if ((sops =
        (struct sembuf *)malloc((nsops * sizeof(struct sembuf)))) == NULL) {
        fprintf(stderr, error_mesg2, nsops);
        exit(2);
    }
}
*sopsp = sops;
return (i);
}

```

```

/*****
MODULO: semaph.c
SCOPO: Utilizzo di semafori
*****/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>

union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};

int main(int argc, char *argv[]) {
    int i,j;
    int pid;

```

```

int semid; /* semid of semaphore set */
key_t key = 1234; /* key to pass to semget() */
int semflg = IPC_CREAT | 0666; /* semflg to pass to semget() */
int nsems = 1; /* nsems to pass to semget() */
int nsops; /* number of operations to do */
struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf));
/* ptr to operations to perform */

/* set up semaphore */

fprintf(stderr, "\nsemget: Setting up semaphore:
           semget(%#lx, %d, %#o)\n",key, nsems, semflg);
if ((semid = semget(key, nsems, semflg)) == -1) {
    perror("semget: semget failed");
    exit(1);
} else
    fprintf(stderr, "semget: semget succeeded: semid = %d\n", semid);

/* get child process */
if ((pid = fork()) < 0) {

```

```

    perror("fork");
    exit(1);
}

if (pid == 0) { /* child */
    i = 0;
    while (i < 3) { /* allow for 3 semaphore sets */

        nsops = 2;

        /* wait for semaphore to reach zero */

        sops[0].sem_num = 0; /* We only use one track */
        sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
        sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous */

        sops[1].sem_num = 0;
        sops[1].sem_op = 1; /* increment semaphore -- take control of track */
        sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */
    }
}

```



```

/* Recap the call to be made. */
fprintf(stderr, "\nsemop:Child Calling semop(%d,&sops,%d) \
        with:",semid,nsops);
for (j = 0; j < nsops; j++) {
    fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
    fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
    fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
}

/* Make the semop() call and report the results. */
if ((j = semop(semid, sops, nsops)) == -1) {
    perror("semop: semop failed");
} else {
    fprintf(stderr, "\tsemop: semop returned %d\n", j);
    fprintf(stderr, "\n\nChild Process Taking Control of Track:
        %d/3 times\n", i+1);
    sleep(5); /* DO Nothing for 5 seconds */
}
nsops = 1;
/* wait for semaphore to reach zero */

```

```

sops[0].sem_num = 0;
sops[0].sem_op = -1; /* Give UP COntrol of track */
sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, async */

if ((j = semop(semid, sops, nsops)) == -1) {
    perror("semop: semop failed");
} else {
    fprintf(stderr, "Child Process Giving up Control of Track:
                %d/3 times\n", i+1);
    sleep(5);
    /* halt process to allow parent to catch semaphor change first */
}
++i;
}
} else /* parent */ {
    i = 0;

while (i < 3) { /* allow for 3 semaphore sets */
    nsops = 2;
    /* wait for semaphore to reach zero */

```

```

sops[0].sem_num = 0;
sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous */

sops[1].sem_num = 0;
sops[1].sem_op = 1; /* increment semaphore -- take control of track */
sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

/* Recap the call to be made. */
fprintf(stderr, "\nsemop:Parent Calling semop(%d, &sops, %d) \
        with:", semid, nsops);
for (j = 0; j < nsops; j++) {
    fprintf(stderr, "\n\t%sops[%d].sem_num = %d, ", j, sops[j].sem_num);
    fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
    fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
}
/* Make the semop() call and report the results. */
if ((j = semop(semid, sops, nsops)) == -1) {
    perror("semop: semop failed");
} else {

```

```

fprintf(stderr, "semop: semop returned %d\n", j);
fprintf(stderr, "Parent Process Taking Control of Track:
           %d/3 times\n", i+1);
sleep(5); /* Do nothing for 5 seconds */
nsops = 1;

/* wait for semaphore to reach zero */
sops[0].sem_num = 0;
sops[0].sem_op = -1; /* Give UP Control of track */
sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT;
/* take off semaphore, asynchronous */
if ((j = semop(semid, sops, nsops)) == -1) {
    perror("semop: semop failed");
} else {
    fprintf(stderr, "Parent Process Giving up Control of Track:
           %d/3 times\n", i+1);
    sleep(5);
    /* halt process to allow child to catch semaphor change first */
}
++i;

```

}
 }
 }
 }