# IPC

System V ha introdotto per primo in Unix alcuni meccanismi di *interprocess communication*:

**message queues** (code di brevi messaggi)

**semafori**

**shared memory** (memoria condivisa)

N.B.: i processi comunicanti sono esclusivamente **locali**.

# Comandi di sistema e chiavi

`ipcs`: stato delle risorse condivise (per cui il processo ha read access):

```
ipcs [ -asmq ] [ -tclup ]
ipcs [ -smq ] -i resource-id
ipcs -h
```

`asmq` è il tipo (All/Semaphore/shared Memory/message Queue).
`tclup` dà il formato: (Time/Creator/Limits/sUmmary/Pid).

`ipcrm`: eliminazione delle risorse condivise:

```
ipcrm [ shm | msg | sem ] id
```

Le risorse IPC System V sono individuate da una chiave, generata con la library function `ftok()`.

```
# include <sys/types.h>
# include <sys/ipc.h>
key_t ftok ( char *pathname, char proj-id );
```

Il valore `key_t` è calcolato da:

- i-node number di `pathname` (16 bit bassi)
- minor device number del file system di `pathname` (8 bit bassi)
- `proj-id` (1 char)

Le chiavi non sono garantite uniche.

# Esempio: `ftok`

```
/* tryftok.c */

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

/* key_t value ftok(pathname,proj-id) returns computed from:
 *   i-node number of pathname (lower 16 bit)
 *   minor device number of pathname's filesystem (lower 8 bit)
 *   proj-id (1 char)
 *  The algorithm does not guarantee a unique key  value.
 */

int main(int argc, char * argv[])
{
    char project_id, pathname[128];
    key_t key;

    if(argc != 3) {
        fprintf(stderr,
                "Usage: %s pathname project-id (1 char)\n",
                argv[0]);
        exit(-1);
    }

    strcpy(pathname, argv[1]);
    project_id = argv[2][0];

    if((key = ftok(pathname,project_id)) == -1) {
        perror("generating key");
        exit(-1);
    }
    printf("key from ftok(%s, %c) is %x\n",
           pathname, project_id, key);
    exit(0);
}
```

# Code di messaggi

Si tratta di:

- code di messaggi *typed*
- con politica FIFO, ma
- *retrieve* selettivo (per tipo)

# Code di messaggi - creazione/apertura

Creazione/accesso a coda:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget ( key_t key, int msgflg );
```

dove

**msgflg** - or logico di:

- **IPC_CREAT** - se la coda non esiste, viene creata
- **IPC_EXCL** - restituisce errore se la coda `key` esiste e si è usato `IPC_CREAT`
- **0xyz** - dove le cifre ottali `x`, `y`, `z` codificano i diritti d'accesso per *user*, *group*, *other*.

  N.B.: i diritti sono i 9 bit bassi (`rw-rw-rw-`) di `msgflg`; hanno senso solo se la coda viene creata;

**key** - chiave o costante `IPC_PRIVATE`: i flag vengono ignorati (eccetto i diritti) e viene creata una nuova coda.

Il valore restituito è l'identificatore della coda creata, o `-1`, che segnala un errore tra, p.es.:

**EACCES** - queue exists, but process has no access permissions to it;

**EEXIST** - queue exists and `msgflg` asserts `IPC_CREAT` and `IPC_EXCL`

**ENOENT** - no message queue exists and `msgflg` didn't assert `IPC_CREAT`

**ENOMEM** - not enough memory

**ENOSPC** - maximum number of queues (`MSGMNI`) would be exceeded

# Code di messaggi - controllo

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl ( int msqid, int cmd, struct msqid_ds *buf );
```

dove:

**buf** punta a una `struct msqid_ds` che descrive lo stato della coda

**msqid** è l'identificatore della message queue

**cmd** è un comando tra:

- **IPC_STAT** - copia lo stato di `msqid` su `*buf`
- **IPC_SET** - cambia lo stato della coda come specificato da `*buf` (limitatamente a *uid*, *gid*, permessi e dimensione)
- **IPC_RMID** - rimuove la coda

Errori (se `msgctl()` restituisce `-1`):

**EACCES** `cmd` is `IPC_STAT` but calling process has no read access permissions

**EFAULT** address pointed to by `buf` isn't accessible

**EIDRM** message queue was removed

**EINVAL** invalid value for `cmd` or `msqid`

**EPERM** `cmd` is `IPC_SET` or `IPC_RMID` but calling process' effective user-ID has insufficient privileges

# La struct msqid_ds

Dal file <bits/msq.h>:

```
struct msqid_ds {
  struct ipc_perm msg_perm;        // describe operation permission
  struct msg *__msg_first;         // pointer to first message on queue
  struct msg *__msg_last;          // pointer to last message on queue
  __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
  struct wait_queue *__wwait;      // ???
  struct wait_queue *__rwait;      // ???
  unsigned short int __msg_cbytes;// current number of bytes on queue
  unsigned short int msg_qnum;     // number of messages currently on queue
  unsigned short int msg_qbytes;   // max number of bytes allowed on queue
  __ipc_pid_t msg_lspid;           // pid of last msgsnd()
  __ipc_pid_t msg_lrpid;           // pid of last msgrcv()
};
```

Da <bits/ipc.h> (NB in ipc_perm, owner≠creator):

```
/* Mode bits for 'msgget', 'semget', and 'shmget'.  */
#define IPC_CREAT   01000       // Create key if key does not exist.
#define IPC_EXCL    02000       // Fail if key exists.
#define IPC_NOWAIT  04000       // Return error on wait.

/* Control commands for 'msgctl', 'semctl', and 'shmctl'.  */
#define IPC_RMID    0       // Remove identifier.
#define IPC_SET     1       // Set 'ipc_perm' options.
#define IPC_STAT    2       // Get 'ipc_perm' options.
#define IPC_INFO    3       // See ipcs.

/* Special key values.  */
#define IPC_PRIVATE ((__key_t) 0)   // Private key.

/* structure passing permission info to IPC operations  */
struct ipc_perm {
    key_t key;                  // Key.
    unsigned short int uid;     // Owner's user ID.
    unsigned short int gid;     // Owner's group ID.
    unsigned short int cuid;    // Creator's user ID.
    unsigned short int cgid;    // Creator's group ID.
    unsigned short int mode;    // Read/write permission.
    unsigned short int seq;     // Sequence number.
};
```

# Invio di messaggi

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msqid, struct msgbuf *msgp,
            int msgsz, int msgflg);
```

**msqid** è la coda oggetto dell'operazione

**msgsz** è la dimensione del messaggio in byte

**msgp** punta al messaggio spedito, che rispecchia il template:

```
struct msgbuf {
  long mtype;       //message type, must be positive
  char mtext[...]; //body, often user-defined struct
};
```

**msgflg** specifica la semantica se inserire il messaggio richiede più di `msg_qbytes` (v. p. 7):

- se asserisce `IPC_NOWAIT`, `msgsnd()` fallisce con `EAGAIN`
- se no `msgsnd()` si blocca finché:
  - o lo spazio diviene disponibile (la chiamata ha successo)
  - o la coda è rimossa (fallisce con `errno==EIDRM`)
  - o il processo riceve un segnale (e `errno==EINTR`)

`msgsnd()` restituisce `0` in caso di successo, e aggiorna i campi della `struct msqid_ds` associata alla coda (v. p. 7).

Restituisce `-1` in caso di errori, come i precedenti e:

**EACCES** calling process has no write access permissions

**EFAULT** address pointed to by `msgp` isn't accessible

**EINVAL** invalid `msqid` or `mtype<=0` or `msgsz<0` or `>MSGMAX`

**ENOMEM** system hasn't enough memory to make a copy of `msgbuf * msgp`

# Ricezione di messaggi con tipo

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int msqid, struct msgbuf *msgp,
            int msgsz, long msgtyp, int msgflg);
```

**msgsz** dim del member `mtext` (v. p. 8) del messaggio `*msgp`

**msgtyp**

- `==0` riceve il 1o messaggio in coda
- `>0` riceve il 1o messaggio di tipo `==msgtyp`, (o `!=msgtyp` se il flag `MSG_EXCEPT` è asserito)
- `<0` riceve il 1o messaggio di tipo `<=-msgtype`

**msgflg** può asserìre, oltre a `MSG_EXCEPT`, i flag:

**MSG_NOERROR** tronca messaggi ricevuti con body `>msgsz`; diversamente questi causano l'errore `E2BIG`

**IPC_NOWAIT** ritorna con errore `ENOMSG` se nessun messaggio del tipo richiesto è in coda

In mancanza di `IPC_NOWAIT`, `msgrcv()` blocca finché:

- o un messaggio del tipo richiesto è inserito in coda
- o la coda è rimossa (fallisce con `errno==EIDRM`)
- o il processo riceve un segnale (fallisce con `errno==EINTR`)

`msgrcv()` restituisce:

- n. byte ricevuti e aggiorna `struct msqid_ds` associata (p. 7)
- `-1` in caso di errori come i suddetti o:

**EACCES** calling process has no read access permissions
**EFAULT** address pointed to by `msgp` isn't accessible
**EINVAL** illegal `msgqid` value, or `msgsz` less than 0

# Esempi: code di messaggi

[Pagina intenzionalmente vuota]

# Send su coda di messaggi

```
/* qsend.c */
/* spedisce messaggio (argv[3] ...) di tipo argv[2]
 *          su coda con projectid argv[1]            */

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

#include "perr.h"

#define MAXC 128
typedef struct {
    long type;
    char body[MAXC];
} mesg_t;

#define PATH "/etc/passwd"


int main(int argc, char * argv[])
{
    mesg_t msg;
    key_t key;
    int msgq_id, retcode, i;
    struct msqid_ds stats;

    if (argc < 4) {
        printf("Usage: %s char msgtype message\n", argv[0]);
        exit(-1);
    }

    key = ftok(PATH, argv[1][0]);
    PERROR(key, "generating key");
    msgq_id = msgget(key, IPC_CREAT | 0600);
    PERROR(msgq_id,"opening msgqueue");
    retcode = msgctl(msgq_id, IPC_STAT, &stats);
    PERROR(retcode,"getting stats");
    printf("Before send, queue %d holds %d messages ",
           msgq_id, stats.msg_qnum);
    printf("=%d bytes (max %d)\n",
           stats.__msg_cbytes, stats.msg_qbytes);

/* qsend.c … */
```

/* **qsend.c** (cont.) */

```
    // construct message to send
    strcpy(msg.body, argv[3]);
    for(i = 4; i < argc; i++) {
        strcat(msg.body, " ");        // concatenate blank
        strcat(msg.body, argv[i]);    // and next arg
    }
    msg.type = atoi(argv[2]);

    retcode =
    msgsnd( msgq_id,                  // identificatore della coda
            (struct msgbuf *) &msg,   // indirizzo del messaggio
            sizeof(msg),              // dimensione del messaggio
            0                         // aspetta se la coda e' piena
          );
    PERROR(retcode, "sending message");

    printf("Try \"ipcs -q\"\n");
    exit(0);
}
```

# Receive da coda di messaggi

```
/* qrecv.c */

/* riceve un msg di msgtyp dato da coda con dato projectid */

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

#include "perr.h"

#define MAXC 128
typedef struct {
    long type;
    char body[MAXC];
} mesg_t;

#define PATH "/etc/passwd"


int main(int argc, char * argv[])
{
    mesg_t msg;
    key_t key;
    int msgq_id, retcode, msgtyp;
    struct msqid_ds stats;

    if (argc != 3) {
        printf("Usage: %s proj-id msgtype\n", argv[0]);
        exit(-1);
    }

    key = ftok(PATH,argv[1][0]);
    PERROR(key,"generating key");
    msgq_id = msgget(key, IPC_CREAT | 0600);
    PERROR(msgq_id, "opening message queue");

    retcode = msgctl(msgq_id, IPC_STAT, &stats);
    PERROR(retcode,"getting stats");


/* qrecv.c … */
```

/* **qrecv.c** (cont.) */

```
    // ricezione di messaggi con tipo

    msgtyp = atoi(argv[2]);
    retcode =
    msgrcv( msgq_id,                // identificatore della coda
            (struct msgbuf *) &msg, // indirizzo del messaggio
            sizeof(msg),            // dimensione del messaggio
            msgtyp,                 // tipo del messaggio da ricevere
            0                       // non troncare, aspetta
          );
    PERROR(retcode, "receiving message");
    printf("from queue %d (of %d messages)\n",
           msgq_id, stats.msg_qnum);
    printf("got message (of type %ld):\n<%s>\n",
           msg.type, msg.body);

    exit(0);
}
```

# Comunicazione client-server via message queue: definizioni

```
/* rotserv.h */

#define SERVER_MSGKEY   0xabcde
#define RESPBASEKEY     172683L
#define PERMALL         0666

typedef struct {
  long  id;
  key_t resp_key;   // carries msg queue for response
  char  data[1024];
} request_t;

typedef struct {
  long  id;
  char  data[1024];
} response_t;
```

# Client-server via message queue: server

```
/* rotserv.c */

/* Server parallelo (ruota modulo 1 i caratteri ricevuti):
 *     riceve richieste su msg queue fissata;
 *     la richiesta contiene la key della msg queue
 *     per la risposta;
 *     fa spawn di una instance di server che calcola la
 *     risposta e la invia.
 */

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

#include "rotserv.h"

int msgid;

void cleanup(int sig)
{
    msgctl(msgid, IPC_RMID, 0);
    fprintf(stderr, "cleaning up ...\n");
    return;
}

int main(int argc,char *argv[])
{
    int i, nbytes, data_len, respid;
    char c;
    request_t req;
    response_t resp;

    msgid = msgget(SERVER_MSGKEY, IPC_CREAT | PERMALL);
    if (msgid < 0) {
        perror("creating request queue"); exit(1);
    }
    signal(SIGINT,cleanup);   // see man 7 signal
    signal(SIGQUIT,cleanup);  // if not installed, SIGQUITquits and
                              // dumps core (default)
    signal(SIGHUP,cleanup);
    signal(SIGCLD,SIG_IGN);   // Ignore child's death (no POSIX!)

/* rotserv.c ... */
```

/* **rotserv.c** (cont.) */

```
    // Ciclo server

    for (;;) {
        // riceve la request
        nbytes =
        msgrcv(msgid, (struct msgbuf *) &req,
               sizeof(request_t)
               - sizeof(long),   // size of type within request
               0L,               // any type
               MSG_NOERROR       // truncate excess data received
              );
        if (nbytes < 0) {        // msgrcv() returns error!
            if (errno != EINTR) {    // under Linux, SIGQUIT/SIGINT seem
                perror("receive");   // to take here with EINVAL, not EINTR
                msgctl(msgid, IPC_RMID, 0);
                exit(1);
            }
            else             // EINTR: got signal while awaiting request
                continue;    // resume receiving requests
        }
        // here main loop forks: father resumes loop immediately
        if (fork() == 0) {
        // child: do service
            data_len = strlen(req.data);
            // make response: rotate ASCII
            for (i = 0; i < data_len; i++) {
                c = req.data[i];
                resp.data[i] = (c == 127) ? 32 : c+1;
            }
            resp.data[i] = 0;    // terminate string
            resp.id = req.id;    // response carries same msgtype
            // open (other) msgqueue for response
            if ((respid = msgget(req.resp_key, 0)) < 0) {
                perror("opening response queue");
                exit(1);
            }
            // send response
            msgsnd(respid, (struct msgbuf *) &resp,
                   data_len+1, 0);
            exit(0);     // child exits here
        }
        // father skips here
    } // server main loop
}
```

# Client-server via message queue: client

```
/* rotclnt.c */

/* adapted from C. Santoro */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#include "rotserv.h"

int reqqid, respqid;     // message queue identifiers

void cleanup(int sig)
{
    msgctl(respqid, IPC_RMID, 0);     // this is why respqid is global
    exit(1);
}

int main(int argc,char *argv[])
{
    int nbytes, reqlen;
    request_t request;
    response_t response;
    key_t respqkey;

    reqqid = msgget(SERVER_MSGKEY, 0);
    if (reqqid < 0) {
        perror("opening request queue");
        exit(1);
    }
    respqkey = RESPBASEKEY+ (key_t) getpid();
    respqid = msgget(respqkey,                // Each client has
                     IPC_CREAT | PERMALL);   // its own key
    if (respqid < 0) {
        perror("creating response queue"); exit(1);
    }

    signal(SIGINT,cleanup);
    signal(SIGQUIT,cleanup);
    signal(SIGHUP,cleanup);

/* rotclnt.c … */
```

/* **rotclnt.c** (cont.) */

```
    // prepare request
    printf("Request:  ");
    scanf("%s", request.data);
    request.id = 1L;
    request.resp_key = respqkey;
    reqlen = sizeof(key_t) + strlen(request.data)
             + 1;        // +1: null byte terminating string

    // send request
    if ( msgsnd(reqqid, (struct msgbuf *) &request,
                reqlen,
                0) < 0 ) {
        perror("sending");
        msgctl(respqid, IPC_RMID, 0); // cleanup
        exit(1);
    }
    nbytes =
    msgrcv(respqid, (struct msgbuf *) &response,
           sizeof(response_t)-sizeof(long),
           0L,  // receive any message type
           MSG_NOERROR);
    if (nbytes < 0) {
        perror("receiving");
        msgctl(respqid,IPC_RMID,0);      // cleanup
        exit(1);
    }
    printf("Response: %s\n",response.data);
    msgctl(respqid, IPC_RMID, 0);    // remove response queue
    exit(0);
}
```

# Semafori

Si tratta di *set* di semafori (*sem set*), condivisibili tra tutti o alcuni tra i processi.

Le operazioni disponibili sono incrementi e decrementi (non necessariamente unitari) atomici.

# Semafori - creazione/apertura

Creazione/accesso a coda:

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int semget ( key_t key, int nsems, int semflg );
```

dove

**nsems** cardinalità del sem set creato (`0` se il set esiste)

**semflg** - or logico di:

- **IPC_CREAT** - se il set non esiste, viene creato
- **IPC_EXCL** - restituisce errore se il set `key` esiste e si è usato `IPC_CREAT`
- **0xyz** - dove le cifre ottali `x`, `y`, `z` codificano i diritti d'accesso per *user*, *group*, *other*.

  N.B.: i diritti sono i 9 bit bassi di `msgflg` e hanno senso solo per una creazione

**key** - chiave o costante `IPC_PRIVATE`: i flag vengono ignorati (eccetto i diritti) e viene creato un nuovo set

# Semafori - creazione/apertura (+)

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int semget ( key_t key, int nsems, int semflg );
```

Il valore iniziale del semaforo è indefinito (occorre una successiva `semctl()` con `SETVAL` o `SETALL`).

Il valore restituito è l'identificatore del set creato, o `-1`, che segnala un errore tra, p.es.:

**EACCES** - sem set exists, but process has no access permissions to it;

**EEXIST** - sem set exists and msgflg asserts `IPC_CREAT` and `IPC_EXCL`

**ENOENT** - no sem set exists and msgflg didn't assert `IPC_CREAT`

**ENOMEM** - not enough memory

**ENOSPC** - creation would exceed maximum number of semaphore identifiers (`SEMMNI`) or maximum total number of semaphores (`SEMMNS`)

NB: `SEMMSL` is maximum cardinality of a semaphore set

**NB** SVr4, SVID. SVr4 documents additional error conditions EINVAL, EFBIG, E2BIG, EAGAIN, ERANGE, EFAULT.

# Semafori - controllo

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl ( int semid, int semnum, int cmd, union semun arg );
```

dove:

**`semid`** è l'identificatore del sem set

**`semnum`** è il semaforo su cui si applica il comando `cmd` (il 1o semaforo è `0`)

ignorato per operazioni riferite a tutto il sem set

**`semun`** qualifica l'operazione o ne contiene il risultato:

```
#if defined(__GNU_LIBRARY__) &&
    !defined(_SEM_SEMUN_UNDEFINED)
/* union semun defined by including sys/sem.h */
#else
/* X/OPEN says to define it ourselves */
union semun {
  int val;                    // value for SETVAL
  struct semid_ds *buf;       // buffer for IPC_STAT, IPC_SET
  unsigned short int *array;      // array for GETALL, SETALL
  struct seminfo *__buf;          // buffer for IPC_INFO (internal)
};
#endif

struct semid_ds
{
  struct ipc_perm sem_perm;               // operation permission struct
  __time_t sem_otime;                     // last semop() time
  __time_t sem_ctime;                     // last time changed by semctl()
  struct sem *__sembase;                  // ptr to 1st semaphore in array
  struct sem_queue *__sem_pending;        // pending operations
  struct sem_queue *__sem_pending_last;   // last pending operation
  struct sem_undo *__undo;                // undo requests on this array
  unsigned short int sem_nsems;           // number of semaphores in set
};
```

# semctl(): comandi

```
int semctl ( int semid, int semnum, int cmd, union semun arg );
```

`cmd` è un comando tra:

**IPC_STAT** copy info from the semaphore set data structure into `struct semid_ds` pointed to by `arg.buf`

**IPC_SET** write permissions stored in `struct semid_ds` pointed to by `arg.buf` to the semaphore set data structure.

**IPC_RMID** remove immediately semaphore set, awakening all waiting processes (returns with error `EIDRM`); `semnum` ignored;

**GETALL** Return value for all semaphores of the set into `arg.array`

**GETNCNT** returns the no. of processes waiting for an increase on the `semnum`th member of `semid`

**GETPID** returns the `pid` of last process that called `semop()` on the `semnum`th member of `semid`

**GETVAL** `semctl()` returns the value of the `semnum`th member of `semid`

**GETZCNT** returns the no. of processes waiting for the `semnum`th member of `semid` to become zero

**SETALL** set value for all semaphores using `arg.array`; undo entries are cleared for altered semaphores in all processes; processes sleeping on the wait queue are awakened if the relevant semaphore becomes 0 or increases

**SETVAL** the `semnum`th member of `semid` is set to `arg.val`; undo entry is cleared for this altered semaphore in all processes; processes sleeping on the wait queue for this semaphore are awakened if it becomes `0` or increases

# semctl(): errori

Errori (se `semctl()` restituisce `-1`):

**EACCES** calling process hasn't access permissions required for `cmd`

**EFAULT** address pointed to by `arg.buf` or `arg.array` isn't accessible

**EIDRM** semaphore set was removed

**EINVAL** invalid value for `cmd` or `semid`

**EPERM** `cmd` is `IPC_SET` or `IPC_RMID` but calling process' effective user-ID has insufficient privileges

**ERANGE** values for `SETVAL` or `SETALL` are `<0` or `>SEMVMX` (`SEMVMX==2**15-1` for this kernel)

# Operazioni sui semafori

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int semop ( int semid, struct sembuf *sops, unsigned nsops );
```

dove `nsops` è il n. di operazioni descritte dall'array (puntato da) `sops`.

Ogni operazione è descritta da una

```
struct sembuf {
  short int sem_num;        // semaphore number within sem set
  short int sem_op;         // semaphore operation
  short int sem_flg;        // operation flag
};
```

La semantica è tutto-o-niente rispetto alle operazioni richieste.

Il comportamento dipende dal segno di `sem_op` e da `sem_flg`, costruito da:

**`IPC_NOWAIT`**

**`SEM_UNDO`** - quando un processo termina, vengono annullate le operazioni che ha effettuato con questo flag;

per questo, viene mantenuto un contatore di *undo*

# Operazioni sui semafori (+)

```
int semop ( int semid, struct sembuf *sops, unsigned nsops );
```

**`sem_op > 0`** viene sommato al valore del semaforo;

il processo non si blocca mai;

il suo contatore di *undo* è aggiornato se `SEM_UNDO` è asserito

**`sem_op == 0`**

se il semaforo è a `0`, l'operazione si completa; se no:

- se `IPC_NOWAIT` è asserito, fallisce con `EAGAIN`
- se no, il processo va a dormire finché:
  - o il semaforo va a `0`
  - o viene cancellato (l'operazione termina con `EIDRM`)
  - o il processo riceve un segnale (l'operazione termina con `EINTR`)

**`sem_op < 0`**

se il semaforo è `>=-sem_op`, questo gli viene sottratto e si procede;

inoltre il contatore di *undo* del processo è aggiornato se `SEM_UNDO` è asserito

se il semaforo è `<-sem_op`:

- se `IPC_NOWAIT` è asserito, `semop()` fallisce con `EAGAIN`
- se no, il processo va a dormire finché:
  - o il semaforo raggiunge/supera `-sem_op` (che gli viene sottratto)
    inoltre il contatore di *undo* del processo è incrementato se `SEM_UNDO` è asserito
  - o viene cancellato (l'operazione termina con `EIDRM`)
  - o il processo riceve un segnale (l'operazione termina con `EINTR`)

# Operazioni sui semafori (+)

```
int semop ( int semid, struct sembuf *sops, unsigned nsops );
```

Se `semop()` ha successo, restituisce `0`, se no `-1`, con gli errori suddetti e:

**E2BIG** argument `nsops>SEMOPM` (max no. of operations allowed per system call)

**EACCES** calling process has no access permissions on the semaphore set as required by one of the specified operations

**EFAULT** address pointed to by `sops` isn't accessible

**EFBIG** for some operation, `sem_num<0` or `sem_num>=` no. of semaphores in the set (`sem_num` is a member within struct `sembuf`)

**EINVAL** semaphore set doesn't exist, or `semid<0` or `nsops<=0`

**ENOMEM** system has not enough memory to allocate an *undo* structure

**ERANGE** for some operation, `semop` plus the current semaphore value exceeds `SEMVMX`, the implementation-dependent maximum for semaphore values

# Semafori e processi: note

- I semafori dei figli di P, in quanto dati dinamici (anziché variabili statiche), coincidono con quelli di P anche dopo il `fork()`.
- Le strutture dati che supportano *undo* di un processo P non vengono ereditate dai suoi figli;
- vengono invece ereditate dal processo che, via `exec()`, rimpiazza P

# Esempi: semafori

[Pagina intenzionalmente vuota]

# Controllo dei semafori: GETALL, SETALL

```
/* getset.c */

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

#include "perr.h"

#define PATH "/etc/passwd"
#define CHAR 'a'

int main(int argc, char * argv[])
{
    key_t chiave;
    int semid,retcode,i;
    short v[16];

    chiave = ftok(PATH,CHAR);
    PERROR(chiave,"generando la chiave");
    semid = semget(chiave, 16, IPC_CREAT | 0600); // crea 16 semafori
    PERROR(semid,"ottenendo l'identificatore del sem set");
    retcode = semctl(semid, 0, GETALL, v);
    PERROR(retcode, "cercando di fare GETALL");
    printf("I semafori (semid %d) valgono: \n", semid);
    for (i = 0; i < 16; i++)
        printf("%d ", v[i]);
    printf("\n");

    for (i = 0; i < 16; i++)
        v[i] += i;
    retcode = semctl(semid, 0, SETALL, v);
    PERROR(retcode, "cercando di fare GETALL");
    retcode = semctl(semid, 0, GETALL, v);
    printf("I semafori (semid %d) valgono: \n", semid);
    for (i = 0; i < 16; i++)
        printf("%d ", v[i]);
    printf("\n");

    exit(0);
}
```

# Operazioni su semaforo

```
/* semop.c */

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

#include "perr.h"

#define PATH "/etc/passwd"


void main(int argc, char * argv[])
{
    key_t key;
    int semid, whichsem, nsem, delta, retcode;
    struct sembuf operation[1];

    if (argc != 4) {
        printf("Usage: %s proj-id which-sem incr\n", argv[0]);
        exit(-1);
    }
    key = ftok(PATH, argv[1][0]);
    PERROR(key, "generating key");

    // Open semaphore argv[2] or create 3*argv[2] semaphores

    whichsem = atoi(argv[2]);
    semid = semget(key, 0, 0);
    if (semid == -1 && errno == ENOENT) {
        perror("opening semaphore set");
        fprintf(stderr, "trying to create %d semaphores\n",
                nsem = 3*whichsem);
        semid = semget(key, nsem, IPC_CREAT | 0600);
        PERROR(semid, "creating semaphore set");
    }
    PERROR(semid, "opening semaphore set");

/* semop.c … */
```

/* **semop.c** (cont.) */

```
    operation[0].sem_num = whichsem;
    operation[0].sem_op = delta = atoi(argv[3]);
    operation[0].sem_flg = 0;
    printf("Attempting semop %d\n", delta);
    retcode = semop(semid, operation, 1);
    PERROR(retcode,"semaphore op");

    printf("Semop %d terminated\n", delta);
    exit(0);
}
```

# Lock/unlock con semafori

```
/* lock.c */

/* I commenti che seguono sono incompleti ed inesatti. */

/* Implementa (singolo) lock, con stato
- coda processi bloccati su lock() non completata
- intero $>$=0 PENDING = n. processi in coda
- val.lock = 0 (open), 1 (locked)
- NB: LOCK open =$>$ PENDING = 0

              LOCK open = 0      LOCK closed     LOCK closed
                                 PENDING = 0     PENDING $>$ 0

    LOCK()    LOCK = locked      process blocks  process blocks
                                 PENDING++       PENDING++

    UNLOCK()  fails              LOCK = open     rendez-vous
                                 succeeds        PENDING--

e con semantica:

 *              lock aperto (0)      |  lock chiuso (1)   *
 *            :-------------------------------------.     *
 *     lock() | chiude               |  blocca      |     *
 *   unlock() | fallisce/non blocca  |  apre        |     *
 *             -------------------------------------      *
 *  NB: unlock() in sequenza non vengono "memorizzati",   *
 *  come accadrebbe se lock()=down(), unlock()=up()       *
 */

/* Si tratta di una semantica simile alle condition
 * variables; vedi ../shmem/condvar.c
 */

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

#define SEMKEY  123456L
#define PERMS   0666


/* lock.c … */
```

/* **lock.c** (cont.) */

```
struct sembuf op_lock[2] = {
    { 0, 0, 0 },          // aspetta che il semaforo 0 diventi 0 (lock aperto)
    { 0, 1, 0 }           // poi incrementa di 1 il semaforo 0 (lock chiuso)
} ;
struct sembuf op_unlock[1] = {// NB: non blocca (IPC_NOWAIT)
    {0, -1, IPC_NOWAIT}        // decrementa il semaforo 0 di 1 (lock aperto)
} ;
extern int sem_id;

int myLock()
{
    if (sem_id < 0) {
        sem_id = semget(SEMKEY, 1, IPC_CREAT | PERMS);
        if (sem_id < 0)
            {perror("creazione del semaforo"); return 1; }
    }
    if (semop(sem_id, op_lock, 2) < 0)      // 2 operazioni in op_lock
        {perror("lock del semaforo"); return 1; }
    return 0;
}

int myUnlock()
{
    if (semop(sem_id, op_unlock, 1) < 0)    // 1 operazione in op_unlock
        {perror("unlock del semaforo"); return 1; }
    return 0;
}
```

# Produttore/consumatore con semafori

```
/* prodcons.c */

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

#define SEMKEY  123456L
#define PERMS   0666

#define PRODUCED    0        // sort of condition variable
#define CONSUMED    1        // sort of condition variable

char buffer[10];
FILE *f;

int sem_id = -1;

void semcreate(void)
{
    sem_id = semget(SEMKEY, 2 , IPC_CREAT | PERMS);
    if (sem_id < 0) {
        perror("creazione del semaforo"); exit(-1);
    }
}

void cleanup(int sig)        // signal handler
{
    semctl(sem_id, 0, IPC_RMID, 0); exit(-1);
}

struct sembuf op_SIGNAL[1] = {
    { 0, 1, SEM_UNDO }       // incrementalo di 1
};
int SIGNAL(int sem_num)      // not signal() syscall!
{
    op_SIGNAL[0].sem_num = sem_num;
    if (semop(sem_id, op_SIGNAL, 1) < 0) {
        perror("SIGNAL"); exit(-1);
    }
    return 0;
}

/* prodcons.c … */
```

/* **prodcons.c** (cont.) */

```
struct sembuf op_WAIT[1] = {
    { 0, -1, SEM_UNDO }     // decrementa il semaforo 0 di 1
};
int WAIT(int sem_num)       // not wait() syscall!
{
    op_WAIT[0].sem_num = sem_num;
    if (semop(sem_id, op_WAIT, 1) < 0) {
        perror("WAIT"); exit(-1);
    }
    return 0;
}

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

    semcreate();
    signal(SIGINT, cleanup);        // installa
// signal(SIGQUIT, cleanup);     //  signal handler
    signal(SIGHUP, cleanup);

    if (fork() != 0) {  // parent process, producer
        for (c = 'A'; ;) {
            memset(buffer,c,10);
            if (++c > 127) c = 32;      // increment c circularly
            printf("Trying to write:\t");
            for (i = 0; i < 10; i++)
                putchar(buffer[i]);
            printf("\t\t\tto common file\n");
                                        // here consumer (other guy) is
            f = fopen("CommonFile","w");//    known to be blocked
            fwrite(buffer,10,1,f);      // write n.1 item of 10 bytes
            fclose(f);
            SIGNAL(PRODUCED);           // signal data were produced
            WAIT(CONSUMED);             // wait for data to be consumed
        }                               //    and consumer to block
    } else {
```

/* **prodcons.c** ... */

/* **prodcons.c** (cont.) */

```
    // child process, consumer
        for (;;) {
            WAIT(PRODUCED);                 // wait for data items ready
            f = fopen("CommonFile","r");//    and producer to block
            fread(buffer,10,1,f);           // read n.1 item of 10 bytes
            fclose(f);
            printf("Read:\t\t\t\t\t");
            for (i = 0; i < 10; i++)
                putchar(buffer[i]);
            printf("\tfrom common file\n");
            sleep(1);
            SIGNAL(CONSUMED);               // signal data were read
        }
    }
}
```

# Segmenti di memoria condivisi

Shared memory segments (SHMS).

# Shared memory segments - creazione/apertura

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

dove

**size** è arrotondato a un multiplo della pagina

**shmflg** - or logico di:

- **IPC_CREAT** - se lo shms non esiste, viene creata
- **IPC_EXCL** - restituisce errore se lo shms `key` esiste e si è usato `IPC_CREAT`
- **0xyz** - dove le cifre ottali `x`, `y`, `z` codificano i diritti d'accesso per *user*, *group*, *other*.

  N.B.: i diritti sono i 9 bit bassi di `shmflg` e hanno senso solo per una creazione

**key** - chiave o costante `IPC_PRIVATE`: i flag vengono ignorati (eccetto i diritti) e viene creata un nuovo shms.

Il valore restituito è l'identificatore del shms creata, o `-1`, che segnala un errore tra, p.es.:

**EINVAL** - shms requested size too big or too small

**EACCES** - shms exists, but process has no access permissions to it;

**EEXIST** - shms exists and `shmflg` asserts `IPC_CREAT` and `IPC_EXCL`

**EIDRM** - shms marked as destroyed or already removed

**ENOENT** - no shms exists for `key` and `shmflg` didn't assert `IPC_CREAT`

**ENOMEM** - no memory could be allocated

**ENOSPC** - maximum number of shms ids (`SHMMNI`) or max total shared memory limit would be exceeded

# Shms - controllo

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

dove:

**buf** punta a una `struct shmid_ds` che descrive lo stato del shms

**shmid** è l'identificatore del shms

**cmd** è un comando tra:

- **IPC_STAT** - copia lo stato di `shmid` su `*buf`
- **IPC_SET** - cambia lo stato del shms come specificato da `*buf` (limitatamente a *uid*, *gid*, permessi e dimensione)
- **IPC_RMID** - rimuove lo shms (un shms deve prima o poi essere rimossa, per liberarne le pagine)

Il super-user può anche usare `SHM_LOCK` e `SHM_UNLOCK`.

Semantica rispetto alle system call:

**fork()** child inherits attached shms

**exec()** all attached shms are detached (not destroyed)

**exit()** all attached shms are detached (not destroyed)

# shmctl(): errori

Errori (se `shmctl()` restituisce `-1`):

**EACCES** `cmd` is `IPC_STAT` but calling process has no read access permissions

**EFAULT** address pointed to by `buf` isn't accessible

**EIDRM** shms was removed

**EINVAL** invalid value for `cmd` or `shmid`

**EPERM** `cmd` is `IPC_SET` or `IPC_RMID` but calling process' effective user-ID has insufficient privileges

# La struct shmid_ds

Struct dei permessi, dal file <bits/shm.h>:

```
struct shmid_ds {
    struct  ipc_perm shm_perm;  // operation perms
    int shm_segsz;              // size of segment (bytes)
    time_t  shm_atime;          // last attach time
    time_t  shm_dtime;          // last detach time
    time_t  shm_ctime;          // last change time
    unsigned short shm_cpid;    // pid of creator
    unsigned short shm_lpid;    // pid of last operator
    short   shm_nattch;         // no. of current attaches
/* the following are private */
    unsigned short   shm_npages;// size of segment (pages)
    unsigned long   *shm_pages;
    struct shm_desc *attaches;  // descriptors for attaches
};
```

Da <bits/ipc.h> (NB owner≠creator):

```
/* Mode bits for 'msgget', 'semget', and 'shmget'.  */
#define IPC_CREAT   01000       // Create key if key does not exist.
#define IPC_EXCL    02000       // Fail if key exists.
#define IPC_NOWAIT  04000       // Return error on wait.

/* Control commands for 'msgctl', 'semctl', and 'shmctl'.  */
#define IPC_RMID    0       // Remove identifier.
#define IPC_SET     1       // Set 'ipc_perm' options.
#define IPC_STAT    2       // Get 'ipc_perm' options.
#define IPC_INFO    3       // See ipcs.

/* Special key values.  */
#define IPC_PRIVATE ((__key_t) 0)   // Private key.

/* structure passing permission info to IPC operations  */
struct ipc_perm {
    key_t key;                  // Key.
    unsigned short int uid;     // Owner's user ID.
    unsigned short int gid;     // Owner's group ID.
    unsigned short int cuid;    // Creator's user ID.
    unsigned short int cgid;    // Creator's group ID.
    unsigned short int mode;    // Read/write permission.
    unsigned short int seq;     // Sequence number.
};
```

# Shms attach e detach

```
# include <sys/types.h>
# include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

`shmat` *attacca* lo shms `shmid` al segmento dati del chiamante, all'indirizzo cosìdeterminato:

**`shmaddr==0`**

Unix cerca una regione libera a partire da `1.5G` verso `1G`

**`shmaddr!=0`**

l'indirizzo mappato è `shmaddr` con arrotondamento alla pagina o a `SHMLBA`, se `shmflg` ha `SHM_RND` asserito;

come scegliere `shmaddr`? dev'essere oltre l'ultimo indirizzo di memoria allocato;

l'indirizzo mappato viene restituito e (in presenza di `SHM_RND`) può differire di `shmaddr`; va quindi usato al suo posto, p.es. come argomento del detach `shmdt()`.

Il processo può usare il flag `SHM_RDONLY` e attaccare lo shms in più indirizzi.

NB: `shmdt()` non cancella lo shms.