laboratorio di sistemi operativi

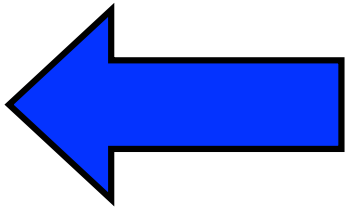
IPC: code di messaggi

Daniele Radicioni

- il materiale di questa lezione è tratto prevalentemente dai testi:
  - lucidi degli anni passati del Prof. Gunetti.
  - Michael Kerrisk, *The Linux Programming interface - a Linux and UNIX® System Programming Handbook*, No Starch Press, San Francisco, CA, 2010.
  - W. Richard Stevens (Author), Stephen A. Rago, *Advanced Programming in the UNIX® Environment* (2nd Edition), Addison-Wesley, 2005.

argomenti del laboratorio UNIX

1. introduzione a UNIX;
2. integrazione C: operatori bitwise, precedenze, preprocessore, pacchettizzazione del codice, compilazione condizionale e utility make;
3. controllo dei processi;
4. segnali;
5. pipe e fifo;
6. code di messaggi;
7. memoria condivisa;
8. semafori;
9. introduzione alla programmazione bash.



Introduction to System V IPC

System V IPC: message queues

- "System V IPC" è l'etichetta utilizzata per riferirsi a tre diversi meccanismi per l'interprocess communication:

1. Le code di messaggi, che possono essere utilizzate per scambiare messaggi fra processi. Le code di messaggi sono simili ai pipe, da cui differiscono per 2 aspetti.
   - primo, i confini dei messaggi sono delimitati, così che i lettori e gli scrittori comunicano in unità di messaggi, e non in stream di byte privi di delimitazioni interne.
   - secondo, ciascun messaggio contiene un membro *type* di tipo intero, ed è possibile selezionare i messaggi per tipo, piuttosto che leggerli nell'ordine in cui sono stati scritti.

System V IPC: semaphores

2. I semafori permettono a molteplici processi di sincronizzare le proprie azioni.
   - un semaforo è un valore intero mantenuto dal kernel visibile a tutti i processi che hanno i permessi necessari.
   - Un processo indica ai propri pari che sta eseguendo una qualche azione facendo una modifica al valore del semaforo.

System V IPC: shared memory

3. La memoria condivisa consente a molteplici processi di condividere lo stesso segmento di memoria.
   - Poiché l'accesso allo spazio della memoria utente è un'operazione veloce, la memoria condivisa è uno dei più veloci strumenti per l'IPC: una volta che un processo ha aggiornato la memoria condivisa, la modifica è immediatamente visibile agli altri processi che condividono lo stesso segmento.

creazione/apertura

- Ciascun meccanismo delle System V IPC ha associata una system call *get* (*msgget()*, *semget()*, o *shmget()*), che è l'analogo della system call *open()* utilizzata per i file.
- Data una *key* intera (analoga a un *filename*), la chiamata *get*:
  - crea un nuovo oggetto IPC con la *key* indicata e restituisce un identificatore unico per quell'oggetto; *oppure*
  - restituisce l'identificatore di un oggetto IPC esistente e avente quella *key*.

```
id = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);

if (id == -1)
   errExit("msgget");
```

- come per tutte le altre syscall *get*, *key* è il primo argomento, e l'identificatore è restituito come risultato della funzione.
- Specifichiamo i *permessi* di accesso al nuovo oggetto come ultimo argomento (flags), utilizzando le costanti elencate qui a fianco

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

Creating/opening a System V IPC object

```
id = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);

if (id == -1)
  ddErrExit("msgget");
```

- Ogni processo che desideri accedere allo stesso oggetto IPC esegue una chiamata *get*, specificando la stessa *key* per ottenere lo stesso identificatore per quell'oggetto.
- Se non esiste un oggetto IPC corrispondente alla *key*, ed è stata specificata la costante *IPC_CREAT* come parte dei flag, la *get* crea un nuovo oggetto IPC.
- Un processo può garantire di essere il creatore di un oggetto IPC specificando il flag *IPC_EXCL*:
  - Se il flag *IPC_EXCL* è specificato, e l'oggetto IPC corrispondente alla *key* esiste già, la *get* fallisce con l'errore *EEXIST*.

Cancellazione di oggetti IPC

```
if (shmctl(id, IPC_RMID, NULL) == -1)
  errExit("shmctl");
```

- La system call *ctl* (*msgctl()*, *semctl()*, *shmctl()*) per ciascun meccanismo di IPC esegue un gran numero di operazioni di controllo sull'oggetto.
- Mentre molte di queste operazioni sono specifiche dei vari meccanismi di IPC, alcune sono comuni a tutti.
  - Per esempio, una generica operazione di controllo è *IPC_RMID*, utilizzata per cancellare un oggetto.

Cancellazione e persistenza di oggetti IPC

- Per le code di messaggi e i semafori, la cancellazione degli oggetti IPC è immediata, e qualsiasi informazione contenuta all'interno dell'oggetto è distrutta, a prescindere dal fatto che qualche altro processo stia ancora utilizzando quell'oggetto.
- La cancellazione di oggetti legati alla memoria condivisa ha un diverso comportamento.
  - Seguendo la chiamata *shmctl(id, IPC_RMID, NULL)*, il segmento di memoria condivisa è rimosso solo dopo che tutti i processi che lo utilizzano lo staccano. Questa modalità è molto più simile alla situazione della cancellazione di un file.

Persistenza degli oggetti IPC

- Gli oggetti IPC hanno una *kernel persistence*: dopo essere stati creati, continuano ad esistere finché sono esplicitamente cancellati o il sistema viene spento.
  - Tale proprietà degli oggetti IPC fornisce alcuni vantaggi: è possibile per un processo creare un oggetto, modificarne lo stato e uscire, lasciando che l'oggetto resti accessibile da altri processi iniziati successivamente.
  - gli svantaggi: esistono limiti di sistema sul massimo numero di oggetti IPC di ogni tipo...
  - il problema è che *questi oggetti sono connectionless*—cioè il kernel non tiene traccia di quali processi hanno un oggetto aperto.
    - Quando si cancella una coda di messaggi, un'applicazione con molti processi può non essere agevolmente in grado di determinare quale sarà l'ultimo processo a richiedere l'accesso all'oggetto e quindi quando l'oggetto può essere cancellato senza problemi.

IPC *Keys*

- Le *keys* dei meccanismi IPC di System V sono valori interi rappresentati con il tipo *key_t*.
- La chiamata *get* mappa una *key* sul corrispondente *identificatore IPC intero*.
  - Queste chiamate garantiscono che se creiamo un nuovo oggetto, quell'oggetto abbia un identificatore unico, e che
  - se specifichiamo la *key* di un oggetto esistente, otteniamo sempre lo stesso identificatore per quell'oggetto.

IPC *Keys*

- Come si genera una *key* unica, tale da garantirci di non ottenere accidentalmente l'identificatore di un oggetto IPC esistente, utilizzato da qualche altra applicazione?
  - Scelta casuale di una valore intero, che è tipicamente memorizzato in un header file incluso da tutti i programmi che usano l'oggetto IPC.
  - Utilizzo della costante *IPC_PRIVATE* come valore della *key* nella invocazione alla *get* al momento della creazione dell'oggetto, che produce sempre un oggetto con una chiave unica.
  - Utilizzo della funzione *ftok()* per generare una *key* molto probabilmente unica.

Generazione di *key* con *IPC_PRIVATE*

```
id = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR);
```

- In questo caso *non* è necessario specificare i flag *IPC_CREAT* o *IPC_EXCL*.
  - Questa tecnica è particolarmente utile in applicazioni con molti processi in cui il processo padre crea l'oggetto IPC prima di eseguire la *fork()*, con il risultato che il figlio eredita l'identificatore dell'oggetto IPC.
  - È possibile utilizzare questa tecnica anche in applicazioni client-server (che coinvolgono processi *non collegati*), ma i client devono avere un mezzo per ottenere gli identificatori degli oggetti IPC creati dal server (e viceversa).
    - Per esempio, dopo avere creato un oggetto IPC, il server potrebbe scrivere il proprio identificatore su un file, che potrebbe essere letto dai client.

Generazione di *key* con *ftok()*

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

        Returns integer key on success, or –1 on error
```

- Questo valore di *key* è generato dal *pathname* fornito e dal valore *proj* utilizzando un algoritmo definito a livello di implementazione.
- Nella generazione della key, *ftok()* utilizza il numero *i-node* piuttosto che il nome del file.
  - Poiché l'algoritmo della *ftok()* dipende dal numero *i-node*, il file in questione non dovrebbe essere rimosso e ricreato mentre l'applicazione è in esecuzione, poiché è probabile che il file sia ricreato con un diverso numero *i-node*.

Generazione di key con ftok()

- Il fine del valore *proj* è consentire di generare diverse *key* a partire dallo stesso file; è utile quando un'applicazione deve creare vari oggetti IPC dello stesso tipo.
  - Storicamente, l'argomento *proj* era di tipo *char*, ed è spesso specificato come tale nelle chiamate a *ftok()*.

```
key_t key;
int id;
key = ftok("/mydir/myfile", 'x');
...
id = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);
...
```

Associated Object Permissions

```
struct ipc_perm {
  key_t  key;
  uid_t uid; /* ushort: Owner's user ID */
  gid_t gid; /* ushort: Owner's group ID */
  uid_t cuid; /* Creator's user ID */
  gid_t cgid; /* Creator's group ID */
  unsigned short mode; /* permissions */
  unsigned short __seq; /*Sequence number*/
};
```

- Il kernel mantiene una struttura dati per ogni istanza di un oggetto IPC.
  - La forma di questa struttura dati varia a seconda del meccanismo (code di messaggi, semafori o memoria condivisa) ed è definito nell'header file corrispondente a ciascun meccanismo IPC.

- La struttura dati associata a un oggetto IPC è inizializzata quando l'oggetto è creato per mezzo dell'appropriata system call *get*.
  - Una volta che l'oggetto è stato creato, un programma può ottenere una copia di questa struttura dati utilizzando l'apposita syscall *ctl*, e specificando un'operazione di tipo *IPC_STAT*.
  - Alcuni elementi della struttura dati possono essere modificati per mezzo della operazione *IPC_SET*.

```
struct shmid_ds shmds;

if (shmctl(id, IPC_STAT, &shmds) == -1)// prelevo dal kernel
  // ... gestione errore ...

shmds.shm_perm.uid = newuid;              // modifico l'owner ID
                                          // (nella copia locale)

if (shmctl(id, IPC_SET, &shmds) == -1) // modifico la copia
                                       // mantenuta dal kernel
 // ... gestione errore ...
```

- Modifica del campo *uid* per un *segmento* di memoria condivisa. La struttura dati associata è di tipo *shmid_ds*.

Associated Object Permissions

- Il campo *mode* della sottostruttura *ipc_perm* contiene i permessi per l'oggetto IPC. I permessi sono inizializzati utilizzando i 9 bit più bassi specificati nei flag della syscall *get*, ma possono essere modificati successivamente utilizzando l'operazione *IPC_SET*.
- Come con i file, i permessi sono divisi in tre categorie: *owner* (o *user*), *group*, e *other*, ed è possibile specificare diversi permessi per ogni categoria.

comandi *ipcs* and *ipcrm*

- I comandi *ipcs* e *ipcrm* sono analoghi ai comandi *ls* e *rm* per i file.

```
$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x6d0731db 262147     mtk        600        8192        2

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x6107c0b8 0          cecilia    660        6
0x6107c0b6 32769      britta     660        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x71075958 229376     cecilia    620        12           2
```

comandi *ipcs* and *ipcrm*

- Di default, per ciascun oggetto, *ipcs* visualizza la *key*, l'identificatore, l'owner e i permessi (espressi in notazione ottale), seguiti da informazioni specifiche per l'oggetto:
  - per la memoria condivisa, *ipcs* visualizza la *dimensione* della regione di memoria condivisa, il numero di processi che attualmente hanno la regione attaccata ai propri spazi di indirizzi e dei flag di stato.
  - per i semafori, *ipcs* visualizza la *dimensione del set* di semafori.
  - per le code di messaggi, *ipcs* visualizza il *numero totale di byte* di dati e il numero di messaggi presenti nella coda.

comandi ipcs and ipcrm

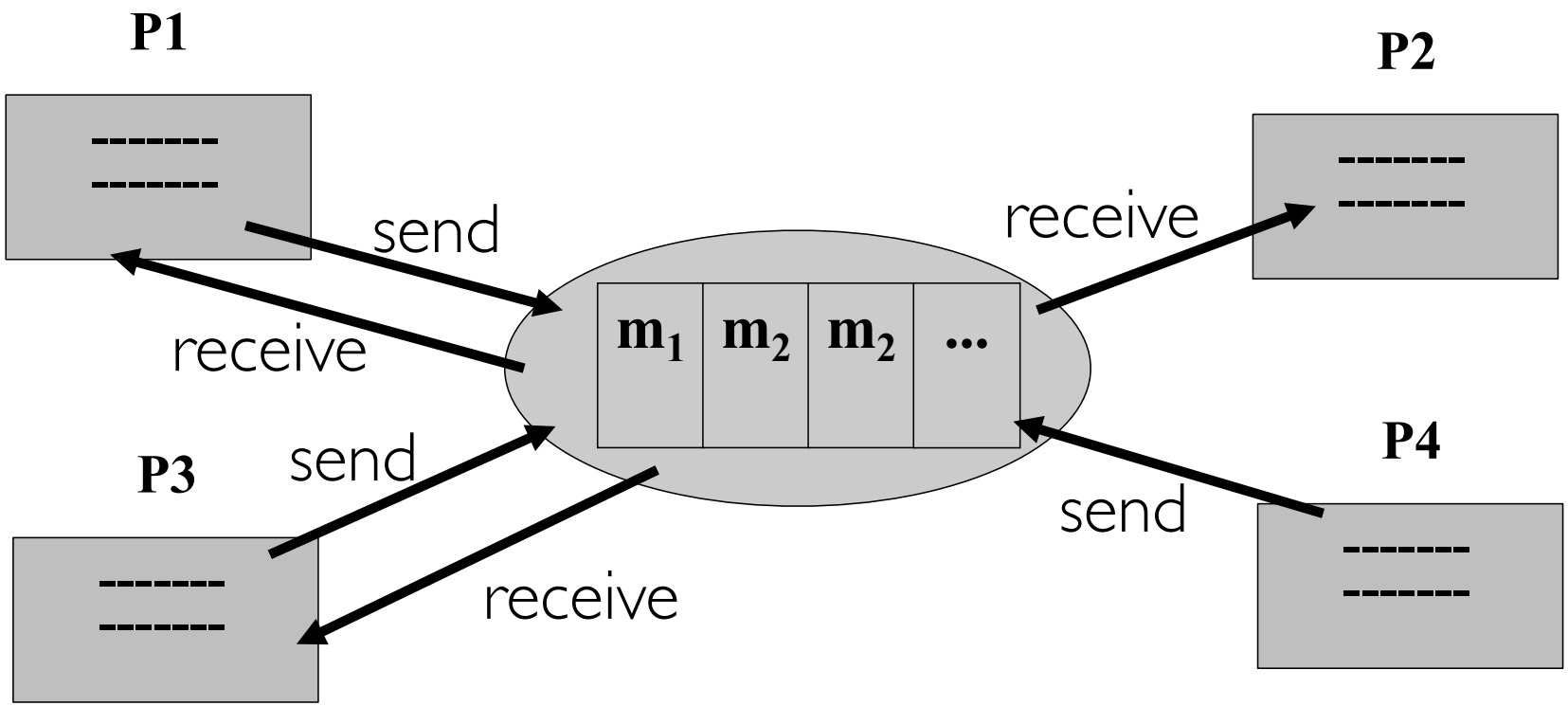
- Il comando *ipcrm* cancella un oggetto IPC object. La forma generale di questo comando è la seguente:

```
$ ipcrm -X key
$ ipcrm -x id
```

- specifichiamo *key* (oppure l'identificatore *id*), e la lettera *X* (oppure *x*) è una ***q*** per le *message queues*, da una ***s*** per i *semaphores*, o da una ***m*** per la *shared memory*.

code di messaggi

Canale di comunicazione su cui possono affacciarsi più processi che inviano e ricevono messaggi



Message Queues

- Le code di messaggi differiscono da pipe e FIFO per queste caratteristiche:
  1. l'identificativo utilizzato per riferirsi a una coda di messaggi è l'identificatore restituito da una chiamata a *msgget()*.
  2. La comunicazione per mezzo di code di messaggi è 'message-oriented'; cioè, il lettore riceve messaggi interi, scritti dallo scrittore.
  3. Non è possibile leggere porzioni di messaggi, lasciandone altre porzioni in coda, o leggere più messaggi alla volta.
  4. Oltre a contenere dati, ogni messaggio contiene un membro di tipo intero che permette di prelevare i messaggi dalla coda in ordine first-in, first-out oppure per tipo.

Creating/Opening a Message Queue

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

 Returns message queue identifier on success, or
                                        –1 on error
```

- l'argomento *key* è una chiave generata utilizzando un numero casuale, *IPC_PRIVATE* o *ftok()*;
- L'argomento *msgflg* è una maschera di bit che specifica i permessi da associare a una nuova coda di messaggi. Se la coda esiste già, permette di verificarne i permessi.

msgflg argument

- zero o più fra i seguenti flag possono essere concatenati in OR (|) nel *msgflg* per controllare la *msgget()*:
  - *IPC_CREAT*: se non esiste una coda con la *key* specificata, crea una nuova coda;
  - *IPC_EXCL*: se è presente anche *IPC_CREAT*, e una coda con la *key* specificata esiste già, restituisci un fallimento con errore *EEXIST*.

msgget() system call

- la system call *msgget()* inizia cercando all'interno delle code di messaggi esistenti quella con la key specificata.
  - Se tale *key* corrisponde a una coda, la *msgget()* restituisce l'identificatore di quella coda (a meno che siano stati specificati sia *IPC_CREAT* sia *IPC_EXCL*, nel qual caso viene restituito un errore).
  - Se la coda non esiste e *IPC_CREAT* è specificato, la *msgget()* crea una nuova coda e ne restituisce l'identificatore al chiamante.

esempio di invocazione *msgget()*

```
// creo una coda di messaggi tramite msgget

if((m_id = msgget(ftok("f_name.c",1), IPC_CREAT))<0)
  errExit("msgget error");

if((m_id = msgget( IPC_PRIVATE, 0644 ))<0)
  errExit("msgget error");
```

condivisione della *key*

- La condivisione della *key* può avvenire in diversi modi:
  - In un file di definizioni *f_header.h*, incluso da tutti i processi che devono usare la stessa coda:

    ```
    #define MYKEY 1234
    ```

  - Il processo responsabile per l'allocazione della coda eseguirà:

    ```
    int q_id = msgget(MYKEY, IPC_CREAT | 0644);
    ```

  - Un processo che deve usare la coda associata a *MYKEY* eseguirà una chiamata come la seguente:

    ```
    int q_id = msgget(MYKEY, 0);
    ```

    - Se la coda associata a *MYKEY* esiste, viene restituito il suo identificatore, altrimenti viene restituito -1

condivisione della *key*

- Se la coda viene usata da un gruppo di processi fratelli, ossia creati tutti dallo stesso padre, è possibile sfruttare questa caratteristica così:

```
int q_id = msgget(getppid(), ...);
```

- Se la coda viene usata da processi in relazione padre-figlio, si può sfruttare il fatto che un figlio eredita copia delle variabili da padre:

```
qid = msgget ( IPC_PRIVATE, ...);
p = fork();
if (p) { ... usa qid ...}
else { ... usa qid …} // --figlio --
```

Exchanging Messages

- Le system call *msgsnd()* e *msgrcv()* eseguono le operazioni di I/O sulle code di messaggi.
  - Il primo argomento in entrambe le chiamate (*msqid*) è un identificatore di coda di messaggi.
  - Il secondo argomento, *msgp*, è un puntatore a una struttura definita dal programmatore e utilizzata per contenere il messaggio inviato o ricevuto. Questa struttura ha la seguente forma:

```
struct mymsg {
   long mtype;    /* Message type */
   char mtext[]; /* Message body */
}
```

```
#include <sys/types.h> /* For portability */
#include <sys/msg.h>
int msgsnd( int msqid, const void *msgp,
            size_t msgsz, int msgflg );

            Returns 0 on success, or -1 on error
```

- La system call *msgsnd()* scrive un messaggio su una coda di messaggi.
- Per inviare un messaggio con la *msgsnd()*, è necessario assegnare il membro *mtype* della struttura a un valore maggiore di 0 e copiare i dati da trasmettere nei membri della struttura.
- L'argomento *msgsz* specifica il numero di bytes contenuti nel membro *mtext* della struttura.

```
#include <sys/types.h> /* For portability */
#include <sys/msg.h>
int msgsnd( int msqid, const void *msgp,
            size_t msgsz, int msgflg );

            Returns 0 on success, or -1 on error
```

- L'ultimo argomento, *msgflg*, è una maschera di bit dei flag che controllano l'operazione di *msgsnd()*. È definito solo un flag:
  - *IPC_NOWAIT*. Consente di eseguire una 'nonblocking send'. Di norma, se una coda è piena, *msgsnd()* si blocca finché non si libera abbastanza spazio per il messaggio che si desidera aggiungere. Se è specificato questo flag, la *msgsnd()* restituisce immediatamente con l'errore *EAGAIN*.

esempio di utilizzo della *msgsnd()*

```
struct queue q;
int m_id;

// ... inizializzazione di m_id e q ...

if(msgsnd(m_id,
      &q, (sizeof(q)-sizeof(long)), IPC_NOWAIT) < 0){
    printf("Message send Error\n");
    exit(1);
}   printf("Message send Error\n");
    exit(1);
}
```

alla dimensione della struttura si sottrae la dimensione del membro type

```
#include <sys/types.h>  /* For portability */
#include <sys/msg.h>
ssize_t msgrcv( int msqid, void *msgp,
    size_t maxmsgsz, long msgtyp, int msgflg );

   Returns number of bytes copied into mtext field, or
                                          -1 on error
```

- La capienza del membro *mtext* del buffer *msgp* è espressa dall'argomento *maxmsgsz*.
  - Se il corpo del messaggio da rimuovere dalla coda supera *maxmsgsz* bytes, nessun messaggio viene rimosso dalla coda, e la *msgrcv()* fallisce con errore *E2BIG*.

Utilizzo di *msgtyp*

- Non necessariamente i messaggi vengono letti nell'ordine con cui sono stati scritti e inviati alla coda. È possibile selezionarli utilizzando il valore contenuto nel membro *mtype*. Questa selezione è controllata dall'argomento *msgtyp*:
  - se *msgtyp* è uguale a *0*, viene prelevato il primo messaggio dalla coda e restituito al processo chiamante;
  - se *msgtyp* è maggiore di *0*, viene prelevato il primo messaggio il cui *mtype* è uguale a *msgtyp* e restituito al chiamante.
    - specificando diversi valori per *msgtyp*, vari processi possono leggere da una coda di messaggi senza competere (*racing*) per leggere gli stessi messaggi. Una tecnica utile è quella in cui ciascun processo seleziona messaggi contenenti il proprio process ID.

Utilizzo di *msgtyp*

- Se *msgtyp* è minore di *0*, la coda è trattata come una coda con priorità. Viene prelevato e restituito per primo il messaggio con il minimo *mtype* minore o uguale al valore assoluto di *msgtyp*.

```
msgrcv(id, &msg, maxmsgsz, -300, 0);
```

queue position	Message type (*mtype*)	Message body (*mtext*)
1	300	...
2	100	...
3	200	...
4	400	...
5	100	...

- queste chiamate *msgrcv()* preleverebbero i messaggi nell'ordine: 2 (type 100), 5 (type 100), 3 (type 200), e 1 (type 300). Un'ulteriore chiamata si bloccherebbe poiché il type dell'ultimo messaggio (400) supera 300.

```
ssize_t msgrcv(int msqid, void *msgp,
    size_t maxmsgsz, long msgtyp, int msgflg);
```

- L'argomento *msgflg* è una maschera di bit formata mettendo in OR zero o più flag:
  - *IPC_NOWAIT*. Esegui una ricezione 'nonblocking'. Normalmente, se sulla coda non sono presenti messaggi con il *msgtyp* specificato, *msgrcv()* si blocca fino a quando tale messaggio non diviene disponibile. Specificando il flag *IPC_NOWAIT* comporta che in questo caso la *msgrcv()* ritorni immediatamente con errore *ENOMSG*.
  - *MSG_NOERROR*. Di default, se la dimensione dell'*mtext* eccede lo spazio disponibile (definito dall'argomento *maxmsgsz*), *msgrcv()* fallisce. Se viene specificato il flag *MSG_NOERROR*, la *msgrcv()* rimuove il messaggio dalla coda, ne tronca l'*mtext* a *maxmsgsz* bytes, e lo restituisce al chiamante. I dati troncati sono persi.

esempio di invocazione di *msgrcv()*

```
// la struct queue è stata definita altrove...
struct queue q;
int m_id;

if((msgrcv(m_id,
   &q, (sizeof(q)-sizeof(long)), getpid(),0))== -1)
errExit("msgrcv error");
```

Message Queue Control Operations

```
#include <sys/types.h>  /* For portability */
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

                        Returns 0 on success, or -1 on error
```

- L'argomento *cmd* specifica l'operazione da eseguire sulla coda.
  - *IPC_RMID*. Rimuove immediatamente la coda di messaggi e la sua associata struttura dati *msqid_ds*.
  - Tutti i messaggi presenti sulla coda vanno persi e qualsiasi processo lettore o scrittore in attesa sulla coda è immediatamente svegliato, con la *msgsnd()* o la *msgrcv()* che falliscono con errore *EIDRM*. Il terzo argomento *msgctl()* è ignorato per questa operazione.

Message Queue Control Operations

```
#include <sys/types.h>  /* For portability */
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

                          Returns 0 on success, or -1 on error
```

- L'argomento *cmd* specifica l'operazione da eseguire sulla coda.
  - *IPC_STAT*. Copia la struttura *msqid_ds* nel buffer puntato da *buf*.
  - *IPC_SET*. Aggiorna i membri della struttura *msqid_ds* associata con questa coda di messaggi, utilizzando i valori presenti nel buffer puntato da *buf*.

Associated Data Structure

- Ogni coda di messaggi ha associata una struttura *msqid_ds*:

```
struct msqid_ds {
  struct ipc_perm msg_perm; /*Ownership e permissions */
  time_t msg_stime; /* Time of last msgsnd() */
  time_t msg_rtime; /* Time of last msgrcv() */
  time_t msg_ctime; /* Time of last change */
  unsigned long __msg_cbytes; /* Number of bytes in
                                 queue*/
  msgqnum_t msg_qnum; /* Number of messages in queue */
  msglen_t msg_qbytes; /* Maximum bytes in queue */
  pid_t msg_lspid; /* PID of last msgsnd() */
  pid_t msg_lrpid; /* PID of last msgrcv() */
}
```

```
struct msqid_ds {
  struct ipc_perm msg_perm; /*Ownership e permissions */
  time_t msg_stime; /* Time of last msgsnd() */
  time_t msg_rtime; /* Time of last msgrcv() */
  time_t msg_ctime; /* Time of last change */
  unsigned long __msg_cbytes; /* Number of bytes in
                                 queue*/
  msgqnum_t msg_qnum; /* Number of messages in queue */
  msglen_t msg_qbytes; /* Maximum bytes in queue */
  pid_t msg_lspid; /* PID of last msgsnd() */
  pid_t msg_lrpid; /* PID of last msgrcv() */
}
```

```
struct ipc_perm {
  key_t  key;
  uid_t uid; /* ushort: Owner's user ID */
  gid_t gid; /* ushort: Owner's group ID */
  uid_t cuid; /* Creator's user ID */
  gid_t cgid; /* Creator's group ID */
  unsigned short mode; /* permissions */
  unsigned short __seq; /*Sequence number*/
};
```

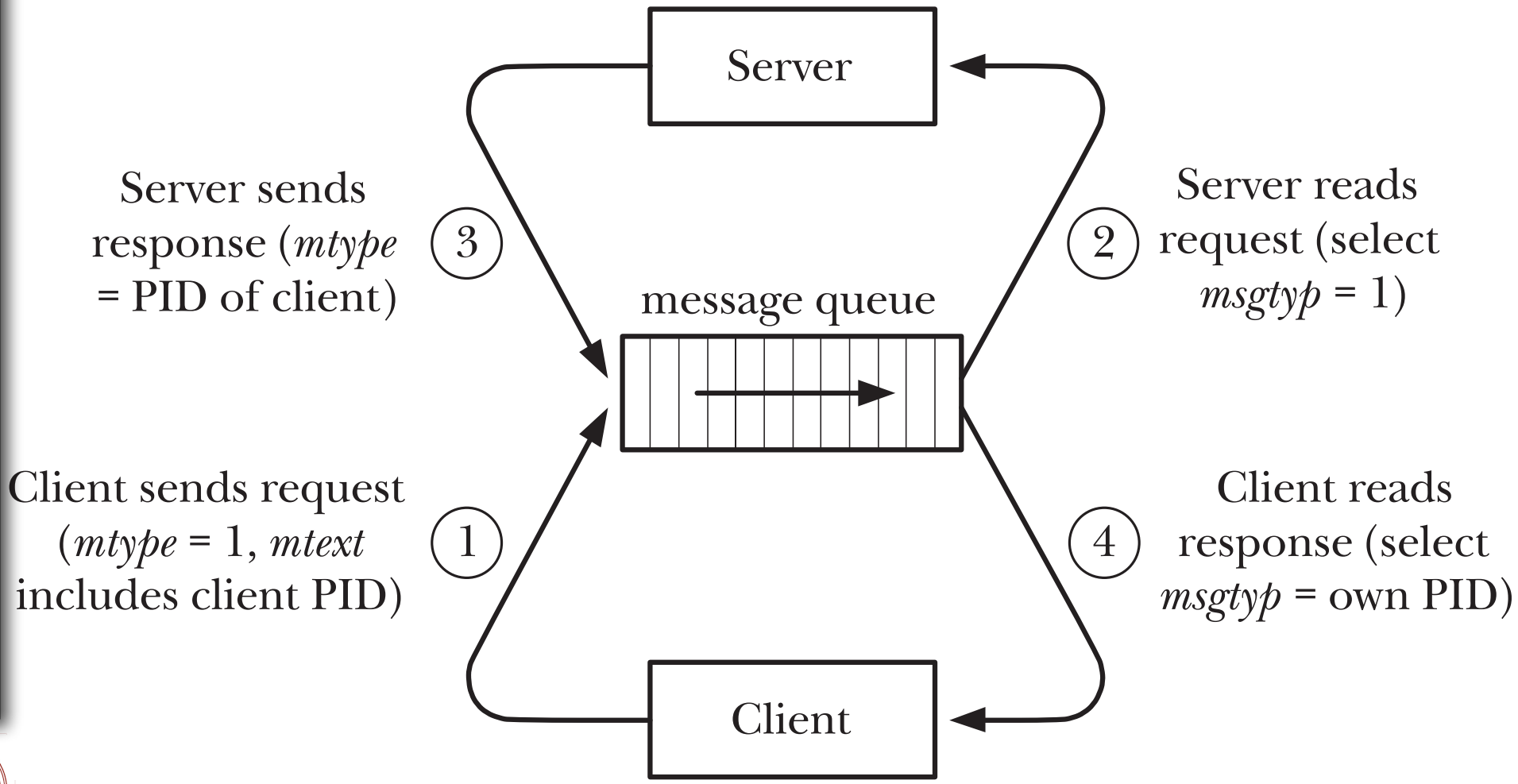
Client-Server with Message Queues

- analizziamo due fra le possibili strategie per implementare un'applicazione client-server utilizzando le code di messaggi:
  - L'utilizzo di una singola coda di messaggi per scambiarsi messaggi in entrambe le direzioni fra server e client.
  - L'utilizzo di code di messaggi separate per il server e per ciascun client. La coda del server è utilizzata per ricevere le richieste provenienti dai client, e le risposte sono inviate ai client per mezzo delle code dei client stessi.

Utilizzo di coda singola

- Appropriato quando i messaggi scambiati fra server e client sono piccoli.
  - Poiché vari processi possono tentare di leggere i messaggi allo stesso tempo, è necessario utilizzare il membro message type (*mtype*) per consentire a ciascun processo di prelevare solo i messaggi destinati a lui.
  - Un modo di realizzare questa soluzione è utilizzare il PID del client come message *type* per i messaggi inviati dal server per il client. Il client può inviare il proprio PID all'interno del messaggio al server.
  - Inoltre, i messaggi al server devono essere distinti da un message type unico. A tale fine possiamo utilizzare il numero 1, che inteso come PID del processo *init*, non rischia mai di essere il PID di un processo client.

Utilizzo di coda singola



Utilizzo di una coda per client

- L'utilizzo di una coda di messaggi per ciascun client (e per il server, ovviamente) è preferibile quando si devono scambiare messaggi di dimensione maggiore.
  - Ciascun client crea la propria coda (tipicamente usando la *key* *IPC_PRIVATE*) e informare il server dell'identificatore della coda, di norma trasmettendo l'identificatore della coda come parte del proprio messaggio al server.
  - Esiste un limite system-wide (*MSGMNI*) al numero di code di messaggi, e il valore di default è basso su alcuni sistemi. Se ci aspettiamo di avere molti client simultaneamente, possiamo avere bisogno di aumentare questo limite.
  - Il server dovrebbe gestire l'eventualità che la coda di messaggi del client non esista più (per esempio, perché il client l'ha cancellata).

Client-server IPC using one message queue per client