# 3

# System V IPC

## 3.1 Introduction

The three types of IPC,

- System V message queues (Chapter 6),
- System V semaphores (Chapter 11), and
- System V shared memory (Chapter 14)

are collectively referred to as "System V IPC." This term is commonly used for these three IPC facilities, acknowledging their heritage from System V Unix. They share many similarities in the functions that access them, and in the information that the kernel maintains on them. This chapter describes all these common properties.

A summary of their functions is shown in Figure 3.1.

| | Message queues | Semaphores | Shared memory |
|---|---|---|---|
| Header | `<sys/msg.h>` | `<sys/sem.h>` | `<sys/shm.h>` |
| Function to create or open | `msgget` | `semget` | `shmget` |
| Function for control operations | `msgctl` | `semctl` | `shmctl` |
| Functions for IPC operations | `msgsnd` `msgrcv` | `semop` | `shmat` `shmdt` |

Figure 3.1 Summary of System V IPC functions.

Information on the design and development of the System V IPC functions is hard to find. [Rochkind 1985] provides the following information: System V message queues, semaphores, and shared memory were developed in the late 1970s at a branch laboratory of Bell

Laboratories in Columbus, Ohio, for an internal version of Unix called (not surprisingly) "Columbus Unix" or just "CB Unix." This version of Unix was used for "Operation Support Systems," transaction processing systems that automated telephone company administration and recordkeeping. System V IPC was added to the commercial Unix system with System V around 1983.

## 3.2 key_t Keys and ftok Function

In Figure 1.4, the three types of System V IPC are noted as using `key_t` values for their names. The header `<sys/types.h>` defines the `key_t` datatype, as an integer, normally at least a 32-bit integer. These integer values are normally assigned by the `ftok` function.

The function `ftok` converts an existing pathname and an integer identifier into a `key_t` value (called an *IPC key*).

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

key_t ftok(const char *pathname, int id);
```

Returns: IPC key if OK, –1 on error

This function takes information derived from the *pathname* and the low-order 8 bits of *id*, and combines them into an integer IPC key.

This function assumes that for a given application using System V IPC, the server and clients all agree on a single *pathname* that has some meaning to the application. It could be the pathname of the server daemon, the pathname of a common data file used by the server, or some other pathname on the system. If the client and server need only a single IPC channel between them, an *id* of one, say, can be used. If multiple IPC channels are needed, say one from the client to the server and another from the server to the client, then one channel can use an *id* of one, and the other an *id* of two, for example. Once the *pathname* and *id* are agreed on by the client and server, then both can call the `ftok` function to convert these into the same IPC key.

Typical implementations of `ftok` call the `stat` function and then combine

1. information about the filesystem on which *pathname* resides (the `st_dev` member of the `stat` structure),
2. the file's i-node number within the filesystem (the `st_ino` member of the `stat` structure), and
3. the low-order 8 bits of the *id*.

The combination of these three values normally produces a 32-bit key. No guarantee exists that two different pathnames combined with the same *id* generate different keys, because the number of bits of information in the three items just listed (filesystem identifier, i-node, and *id*) can be greater than the number of bits in an integer. (See Exercise 3.5.)

The i-node number is never 0, so most implementations define `IPC_PRIVATE` (which we describe in Section 3.4) to be 0.

If the *pathname* does not exist, or is not accessible to the calling process, `ftok` returns −1. Be aware that the file whose *pathname* is used to generate the key must not be a file that is created and deleted by the server during its existence, since each time it is created, it can assume a new i-node number that can change the key returned by `ftok` to the next caller.

## Example

The program in Figure 3.2 takes a pathname as a command-line argument, calls `stat`, calls `ftok`, and then prints the `st_dev` and `st_ino` members of the `stat` structure, and the resulting IPC key. These three values are printed in hexadecimal, so we can easily see how the IPC key is constructed from these two values and our *id* of `0x57`.

*svipc/ftok.c*

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    struct stat stat;

    if (argc != 2)
        err_quit("usage: ftok <pathname>");

    Stat(argv[1], &stat);
    printf("st_dev: %lx, st_ino: %lx, key: %x\n",
           (u_long) stat.st_dev, (u_long) stat.st_ino,
           Ftok(argv[1], 0x57));

    exit(0);
}
```

*svipc/ftok.c*

Figure 3.2 Obtain and print filesystem information and resulting IPC key.

Executing this under Solaris 2.6 gives us the following:

```
solaris % ftok /etc/system
st_dev: 800018, st_ino: 4a1b, key: 57018a1b
solaris % ftok /usr/tmp
st_dev: 800015, st_ino: 10b78, key: 57015b78
solaris % ftok /home/rstevens/Mail.out
st_dev: 80001f, st_ino: 3b03, key: 5701fb03
```

Apparently, the *id* is in the upper 8 bits, the low-order 12 bits of `st_dev` in the next 12 bits, and the low-order 12 bits of `st_ino` in the low-order 12 bits.

Our purpose in showing this example is not to let us count on this combination of information to form the IPC key, but to let us see how one implementation combines the *pathname* and *id*. Other implementations may do this differently.

FreeBSD uses the lower 8 bits of the *id*, the lower 8 bits of `st_dev`, and the lower 16 bits of `st_ino`.

Note that the mapping done by `ftok` is one-way, since some bits from `st_dev` and `st_ino` are not used. That is, given a key, we cannot determine the pathname that was used to create the key.

## 3.3 `ipc_perm` Structure

The kernel maintains a structure of information for each IPC object, similar to the information it maintains for files.

```
struct ipc_perm {
  uid_t    uid;      /* owner's user id */
  gid_t    gid;      /* owner's group id */
  uid_t    cuid;     /* creator's user id */
  gid_t    cgid;     /* creator's group id */
  mode_t   mode;     /* read-write permissions */
  ulong_t  seq;      /* slot usage sequence number */
  key_t    key;      /* IPC key */
};
```

This structure, and other manifest constants for the System V IPC functions, are defined in the `<sys/ipc.h>` header. We talk about all the members of this structure in this chapter.

## 3.4 Creating and Opening IPC Channels

The three `get`*XXX* functions that create or open an IPC object (Figure 3.1) all take an IPC *key* value, whose type is `key_t`, and return an integer *identifier*. This identifier is *not* the same as the *id* argument to the `ftok` function, as we see shortly. An application has two choices for the *key* value that is the first argument to the three `get`*XXX* functions:

1. call `ftok`, passing it a *pathname* and *id*, or
2. specify a *key* of `IPC_PRIVATE`, which guarantees that a new, unique IPC object is created.

The sequence of steps is shown in Figure 3.3.

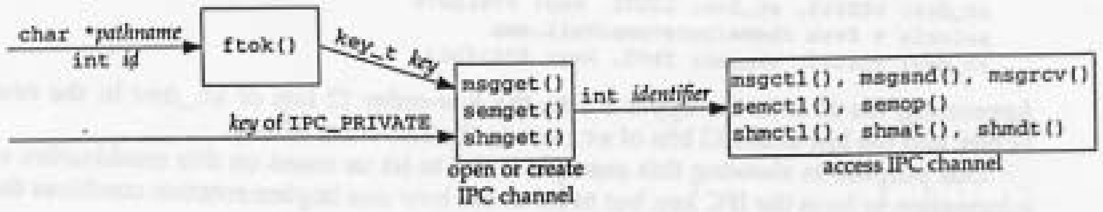

Figure 3.3 Generating IPC identifiers from IPC keys.

All three `getXXX` functions (Figure 3.1) also take an *oflag* argument that specifies the read–write permission bits (the `mode` member of the `ipc_perm` structure) for the IPC object, and whether a new IPC object is being created or an existing one is being referenced. The rules for whether a new IPC object is created or whether an existing one is referenced are as follows:

- Specifying a *key* of `IPC_PRIVATE` guarantees that a unique IPC object is created. No combinations of *pathname* and *id* exist that cause `ftok` to generate a *key* value of `IPC_PRIVATE`.
- Setting the `IPC_CREAT` bit of the *oflag* argument creates a new entry for the specified *key*, if it does not already exist. If an existing entry is found, that entry is returned.
- Setting both the `IPC_CREAT` and `IPC_EXCL` bits of the *oflag* argument creates a new entry for the specified *key*, only if the entry does not already exist. If an existing entry is found, an error of `EEXIST` is returned, since the IPC object already exists.

  The combination of `IPC_CREAT` and `IPC_EXCL` with regard to IPC objects is similar to the combination of `O_CREAT` and `O_EXCL` with regard to the `open` function.

  Setting the `IPC_EXCL` bit, without setting the `IPC_CREAT` bit, has no meaning.

The actual logic flow for opening an IPC object is shown in Figure 3.4. Figure 3.5 shows another way of looking at Figure 3.4.

Note that in the middle line of Figure 3.5, the `IPC_CREAT` flag without `IPC_EXCL`, we do not get an indication whether a new entry has been created or whether we are referencing an existing entry. In most applications, the server creates the IPC object and specifies either `IPC_CREAT` (if it does not care whether the object already exists) or `IPC_CREAT | IPC_EXCL` (if it needs to check whether the object already exists). The clients specify neither flag (assuming that the server has already created the object).

The System V IPC functions define their own `IPC_xxx` constants, instead of using the `O_CREAT` and `O_EXCL` constants that are used by the standard `open` function along with the Posix IPC functions (Figure 2.3).

Also note that the System V IPC functions combine their `IPC_xxx` constants with the permission bits (which we describe in the next section) into a single *oflag* argument. The `open` function along with the Posix IPC functions have one argument named *oflag* that specifies the various `O_xxx` flags, and another argument named *mode* that specifies the permission bits.

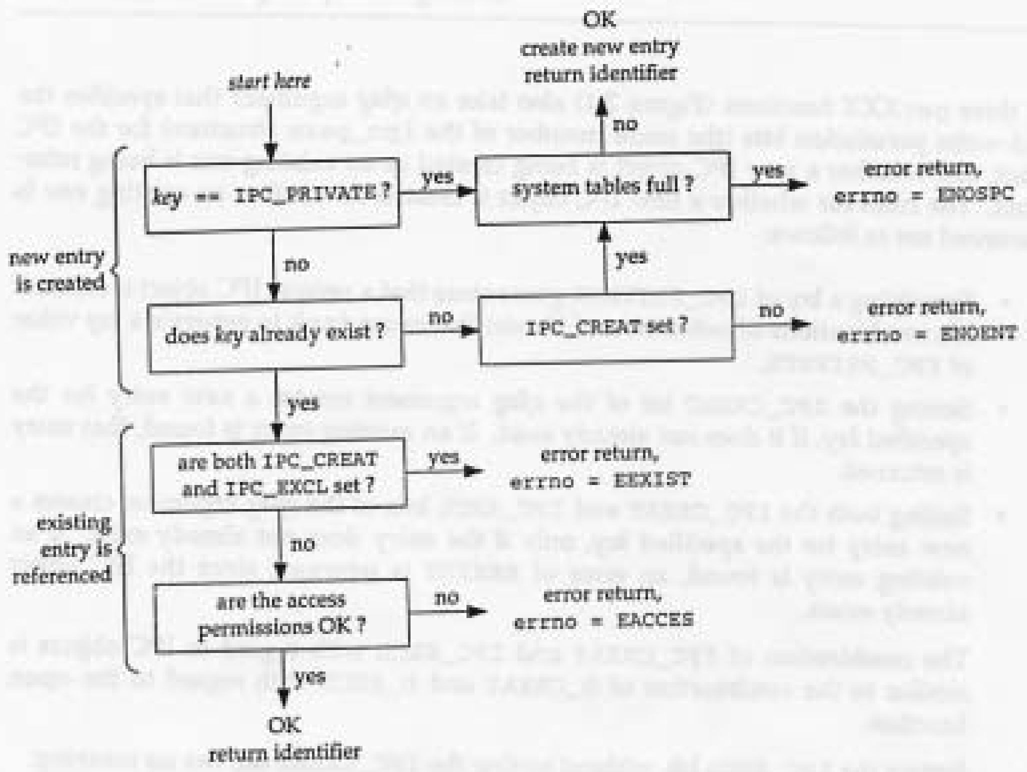

Figure 3.4 Logic for creating or opening an IPC object.

| *oflag* argument | *key* does not exist | *key* already exists |
|---|---|---|
| no special flags | error, `errno = ENOENT` | OK, references existing object |
| `IPC_CREAT` | OK, creates new entry | OK, references existing object |
| `IPC_CREAT \| IPC_EXCL` | OK, creates new entry | error, `errno = EEXIST` |

Figure 3.5 Logic for creating or opening an IPC channel.

## 3.5 IPC Permissions

Whenever a new IPC object is created using one of the `get`*XXX* functions with the `IPC_CREAT` flag, the following information is saved in the `ipc_perm` structure (Section 3.3):

1. Some of the bits in the *oflag* argument initialize the `mode` member of the `ipc_perm` structure. Figure 3.6 shows the permission bits for the three different IPC mechanisms. (The notation >> 3 means the value is right shifted 3 bits.)

| Numeric (octal) | Symbolic values | | | Description |
|---|---|---|---|---|
| | Message queue | Semaphore | Shared memory | |
| `0400` | `MSG_R` | `SEM_R` | `SHM_R` | read by user |
| `0200` | `MSG_W` | `SEM_A` | `SHM_W` | write by user |
| `0040` | `MSG_R >> 3` | `SEM_R >> 3` | `SHM_R >> 3` | read by group |
| `0020` | `MSG_W >> 3` | `SEM_A >> 3` | `SHM_W >> 3` | write by group |
| `0004` | `MSG_R >> 6` | `SEM_R >> 6` | `SHM_R >> 6` | read by others |
| `0002` | `MSG_W >> 6` | `SEM_A >> 6` | `SHM_W >> 6` | write by others |

Figure 3.6 *mode* values for IPC read-write permissions.

2. The two members `cuid` and `cgid` are set to the effective user ID and effective group ID of the calling process, respectively. These two members are called the *creator IDs*.
3. The two members `uid` and `gid` in the `ipc_perm` structure are also set to the effective user ID and effective group ID of the calling process. These two members are called the *owner IDs*.

The creator IDs never change, although a process can change the owner IDs by calling the `ctl`*XXX* function for the IPC mechanism with a command of `IPC_SET`. The three `ctl`*XXX* functions also allow a process to change the permission bits of the `mode` member for the IPC object.

> Most implementations define the six constants `MSG_R`, `MSG_W`, `SEM_R`, `SEM_A`, `SHM_R`, and `SHM_W` shown in Figure 3.6 in the `<sys/msg.h>`, `<sys/sem.h>`, and `<sys/shm.h>` headers. But these are not required by Unix 98. The suffix `A` in `SEM_A` stands for "alter."
>
> The three `get`*XXX* functions do not use the normal Unix *file mode creation mask*. The permissions of the message queue, semaphore, or shared memory segment are set to exactly what the function specifies.
>
> Posix IPC does not let the creator of an IPC object change the owner. Nothing is like the `IPC_SET` command with Posix IPC. But if the Posix IPC name is stored in the filesystem, then the superuser can change the owner using the chown command.

Two levels of checking are done whenever an IPC object is accessed by any process, once when the IPC object is opened (the `get`*XXX* function) and then each time the IPC object is used:

1. Whenever a process establishes access to an existing IPC object with one of the `get`*XXX* functions, an initial check is made that the caller's *oflag* argument does not specify any access bits that are not in the `mode` member of the `ipc_perm` structure. This is the bottom box in Figure 3.4. For example, a server process can set the `mode` member for its input message queue so that the group-read and other-read permission bits are off. Any process that tries to specify an *oflag* argument that includes these bits gets an error return from the `msgget` function. But this test that is done by the `get`*XXX* functions is of little use. It implies that

the caller knows which permission category it falls into—user, group, or other. If the creator specifically turns off certain permission bits, and if the caller specifies these bits, the error is detected by the get.XXX function. Any process, however, can totally bypass this check by just specifying an *oflag* argument of 0 if it knows that the IPC object already exists.

2. Every IPC operation does a permission test for the process using the operation. For example, every time a process tries to put a message onto a message queue with the `msgsnd` function, the following tests are performed in the order listed. As soon as a test grants access, no further tests are performed.

   a. The superuser is always granted access.

   b. If the effective user ID equals either the `uid` value or the `cuid` value for the IPC object, and if the appropriate access bit is on in the `mode` member for the IPC object, permission is granted. By "appropriate access bit," we mean the read-bit must be set if the caller wants to do a read operation on the IPC object (receiving a message from a message queue, for example), or the write-bit must be set for a write operation.

   c. If the effective group ID equals either the `gid` value or the `cgid` value for the IPC object, and if the appropriate access bit is on in the `mode` member for the IPC object, permission is granted.

   d. If none of the above tests are true, the appropriate "other" access bit must be on in the `mode` member for the IPC object, for permission to be allowed.

## 3.6 Identifier Reuse

The `ipc_perm` structure (Section 3.3) also contains a variable named `seq`, which is a slot usage sequence number. This is a counter that is maintained by the kernel for every potential IPC object in the system. Every time an IPC object is removed, the kernel increments the slot number, cycling it back to zero when it overflows.

What we are describing in this section is the common SVR4 implementation. This implementation technique is not mandated by Unix 98.

This counter is needed for two reasons. First, consider the file descriptors maintained by the kernel for open files. They are small integers, but have meaning only within a single process—they are process-specific values. If we try to read from file descriptor 4, say, in a process, this approach works only if that process has a file open on this descriptor. It has no meaning whatsoever for a file that might be open on file descriptor 4 in some other unrelated process. System V IPC identifiers, however, are *systemwide* and not process-specific.

We obtain an IPC identifier (similar to a file descriptor) from one of the `get` functions: `msgget`, `semget`, and `shmget`. These identifiers are also integers, but their meaning applies to *all* processes. If two unrelated processes, a client and server, for example, use a single message queue, the message queue identifier returned by the

```
msqid = 100
msqid = 150
msqid = 200
msqid = 250
msqid = 300
msqid = 350
msqid = 400
msqid = 450
```

If we run the program again, we see that this slot usage sequence number is a kernel variable that persists between processes.

```
solaris % slot
msqid = 500
msqid = 550
msqid = 600
msqid = 650
msqid = 700
msqid = 750
msqid = 800
msqid = 850
msqid = 900
msqid = 950
```

## 3.7 ipcs and ipcrm Programs

Since the three types of System V IPC are not identified by pathnames in the filesystem, we cannot look at them or remove them using the standard `ls` and `rm` programs. Instead, two special programs are provided by any system that implements these types of IPC: `ipcs`, which prints various pieces of information about the System V IPC features, and `ipcrm`, which removes a System V message queue, semaphore set, or shared memory segment. The former supports about a dozen command-line options, which affect which of the three types of IPC is reported and what information is output, and the latter supports six command-line options. Consult your manual pages for the details of all these options.

Since System V IPC is not part of Posix, these two commands are not standardized by Posix.2. But these two commands are part of Unix 98.

## 3.8 Kernel Limits

Most implementations of System V IPC have inherent kernel limits, such as the maximum number of message queues and the maximum number of semaphores per semaphore set. We show some typical values for these limits in Figures 6.25, 11.9, and 14.5. These limits are often derived from the original System V implementation.

Section 11.2 of [Bach 1986] and Chapter 8 of [Goodheart and Cox 1994] both describe the System V implementation of messages, semaphores, and shared memory. Some of these limits are described therein.

Unfortunately, these kernel limits are often too small, because many are derived from their original implementation on a small address system (the 16-bit PDP-11). Fortunately, most systems allow the administrator to change some or all of these default limits, but the required steps are different for each flavor of Unix. Most require rebooting the running kernel after changing the values. Unfortunately, some implementations still use 16-bit integers for some of the limits, providing a hard limit that cannot be exceeded.

Solaris 2.6, for example, has 20 of these limits. Their current values are printed by the `sysdef` command, although the values are printed as 0 if the corresponding kernel module has not been loaded (i.e., the facility has not yet been used). These may be changed by placing any of the following statements in the `/etc/system` file, which is read when the kernel bootstraps.

```
set msgsys:msginfo_msgseg = value
set msgsys:msginfo_msgssz = value
set msgsys:msginfo_msgtql = value
set msgsys:msginfo_msgmap = value
set msgsys:msginfo_msgmax = value
set msgsys:msginfo_msgmnb = value
set msgsys:msginfo_msgmni = value

set semsys:seminfo_semopm = value
set semsys:seminfo_semume = value
set semsys:seminfo_semaem = value
set semsys:seminfo_semmap = value
set semsys:seminfo_semvmx = value
set semsys:seminfo_semmsl = value
set semsys:seminfo_semmni = value
set semsys:seminfo_semmns = value
set semsys:seminfo_semmnu = value

set shmsys:shminfo_shmmin = value
set shmsys:shminfo_shmseg = value
set shmsys:shminfo_shmmax = value
set shmsys:shminfo_shmmni = value
```

The last six characters of the name on the left-hand side of the equals sign are the variables listed in Figures 6.25, 11.9, and 14.5.

With Digital Unix 4.0B, the `sysconfig` program can query or modify many kernel parameters and limits. Here is the output of this program with the `-q` option, which queries the kernel for the current limits, for the `ipc` subsystem. We have omitted some lines unrelated to the System V IPC facility.

```
alpha % /sbin/sysconfig -q ipc
ipc:
msg-max = 8192
msg-mnb = 16384
msg-mni = 64
msg-tql = 40

shm-max = 4194304
shm-min = 1
shm-mni = 128
shm-seg = 32
```

```
sem-mni = 16
sem-msl = 25
sem-opm = 10
sem-ume = 10
sem-vmx = 32767
sem-aem = 16384
num-of-sems = 60
```

Different defaults for these parameters can be specified in the /etc/sysconfigtab file, which should be maintained using the sysconfigdb program. This file is read when the system bootstraps.

## 3.9 Summary

The first argument to the three functions, msgget, semget, and shmget, is a System V IPC key. These keys are normally created from a pathname using the system's ftok function. The key can also be the special value of IPC_PRIVATE. These three functions create a new IPC object or open an existing IPC object and return a System V IPC identifier: an integer that is then used to identify the object to the remaining IPC functions. These integers are not per-process identifiers (like descriptors) but are systemwide identifiers. These identifiers are also reused by the kernel after some time.

Associated with every System V IPC object is an ipc_perm structure that contains information such as the owner's user ID, group ID, read-write permissions, and so on. One difference between Posix IPC and System V IPC is that this information is always available for a System V IPC object (by calling one of the three XXXctl functions with an argument of IPC_STAT), but access to this information for a Posix IPC object depends on the implementation. If the Posix IPC object is stored in the filesystem, and if we know its name in the filesystem, then we can access this same information using the existing filesystem tools.

When a new System V IPC object is created or an existing object is opened, two flags are specified to the getXXX function (IPC_CREAT and IPC_EXCL), combined with nine permission bits.

Undoubtedly, the biggest problem in using System V IPC is that most implementations have artificial kernel limits on the sizes of these objects, and these limits date back to their original implementation. These mean that most applications that make heavy use of System V IPC require that the system administrator modify these kernel limits, and accomplishing this change differs for each flavor of Unix.

## Exercises

3.1 Read about the msgctl function in Section 6.5 and modify the program in Figure 3.7 to print the seq member of the ipc_perm structure in addition to the assigned identifier.

3.2 Immediately after running the program in Figure 3.7, we run a program that creates two message queues. Assuming no other message queues have been used by any other applications since the kernel was booted, what two values are returned by the kernel as the message queue identifiers?

3.3 We noted in Section 3.5 that the System V IPC getXXX functions do not use the file mode creation mask. Write a test program that creates a FIFO (using the mkfifo function described in Section 4.6) and a System V message queue, specifying a permission of (octal) 666 for both. Compare the permissions of the resulting FIFO and message queue. Make certain your shell umask value is nonzero before running this program.

3.4 A server wants to create a unique message queue for its clients. Which is preferable—using some constant pathname (say the server's executable file) as an argument to ftok, or using IPC_PRIVATE?

3.5 Modify Figure 3.2 to print just the IPC key and pathname. Run the find program to print all the pathnames on your system and run the output through the program just modified. How many pathnames map to the same key?

3.6 If your system supports the sar program ("system activity reporter"), run the command

```
sar -m 5 6
```

This prints the number of message queue operations per second and the number of semaphore operations per second, sampled every 5 seconds, 6 times.