51

INTRODUCTION TO POSIX IPC

The POSIX.1b realtime extensions defined a set of IPC mechanisms that are analogous to the System V IPC mechanisms described in Chapters 45 to 48. (One of the POSIX.1b developers' aims was to devise a set of IPC mechanisms that did not suffer the deficiencies of the System V IPC facilities.) These IPC mechanisms are collectively referred to as POSIX IPC. The three POSIX IPC mechanisms are the following:

- *Message queues* can be used to pass messages between processes. As with System V message queues, message boundaries are preserved, so that readers and writers communicate in units of messages (as opposed to the undelimited byte stream provided by a pipe). POSIX message queues permit each message to be assigned a priority, which allows high-priority messages to be queued ahead of low-priority messages. This provides some of the same functionality that is available via the type field of System V messages.
- *Semaphores* permit multiple processes to synchronize their actions. As with System V semaphores, a POSIX semaphore is a kernel-maintained integer whose value is never permitted to go below 0. POSIX semaphores are simpler to use than System V semaphores: they are allocated individually (as opposed to System V semaphore *sets*), and they are operated on individually using two operations that increase and decrease a semaphore's value by one (as opposed to the ability of the *semop()* system call to atomically add or subtract arbitrary values from multiple semaphores in a System V semaphore set).

- *Shared memory* enables multiple processes to share the same region of memory. As with System V shared memory, POSIX shared memory provides fast IPC. Once one process has updated the shared memory, the change is immediately visible to other processes sharing the same region.

This chapter provides an overview of the POSIX IPC facilities, focusing on their common features.

51.1 API Overview

The three POSIX IPC mechanisms have a number of common features. Table 51-1 summarizes their APIs, and we go into the details of their common features in the next few pages.

> Except for a mention in Table 51-1, in the remainder of this chapter, we'll overlook the fact that POSIX semaphores come in two flavors: named semaphores and unnamed semaphores. Named semaphores are like the other POSIX IPC mechanisms that we describe in this chapter: they are identified by a name, and are accessible by any process that has suitable permissions on the object. An unnamed semaphore doesn't have an associated identifier; instead, it is placed in an area of memory that is shared by a group of processes or by the threads of a single process. We go into the details of both types of semaphores in Chapter 53.

Table 51-1: Summary of programming interfaces for POSIX IPC objects

Interface	Message queues	Semaphores	Shared memory
Header file	`<mqueue.h>`	`<semaphore.h>`	`<sys/mman.h>`
Object handle	*mqd_t*	*sem_t **	*int* (file descriptor)
Create/open	*mq_open()*	*sem_open()*	*shm_open()* + *mmap()*
Close	*mq_close()*	*sem_close()*	*munmap()*
Unlink	*mq_unlink()*	*sem_unlink()*	*shm_unlink()*
Perform IPC	*mq_send()*, *mq_receive()*	*sem_post()*, *sem_wait()*, *sem_getvalue()*	operate on locations in shared region
Miscellaneous operations	*mq_setattr()*—set attributes *mq_getattr()*—get attributes *mq_notify()*—request notification	*sem_init()*—initialize unnamed semaphore *sem_destroy()*—destroy unnamed semaphore	(none)

IPC object names

To access a POSIX IPC object, we must have some means of identifying it. The only portable means that SUSv3 specifies to identify a POSIX IPC object is via a name consisting of an initial slash, followed by one of more nonslash characters; for example, `/myobject`. Linux and some other implementations (e.g., Solaris) permit this type of portable naming for IPC objects.

On Linux, names for POSIX shared memory and message queue objects are limited to NAME_MAX (255) characters. For semaphores, the limit is 4 characters less, since the implementation prepends the string *sem.* to the semaphore name.

SUSv3 doesn't prohibit names of a form other than /myobject, but says that the semantics of such names are implementation-defined. The rules for creating IPC object names on some systems are different. For example, on Tru64 5.1, IPC object names are created as names within the standard file system, and the name is interpreted as an absolute or relative pathname. If the caller doesn't have permission to create a file in that directory, then the IPC *open* call fails. This means that unprivileged programs can't create names of the form /myobject on Tru64, since unprivileged users normally can't create files in the root directory (/). Some other implementations have similar implementation-specific rules for the construction of the names given to IPC *open* calls. Therefore, in portable applications, we should isolate the generation of IPC object names into a separate function or header file that can be tailored to the target implementation.

Creating or opening an IPC object

Each IPC mechanism has an associated *open* call (*mq_open()*, *sem_open()*, or *shm_open()*), which is analogous to the traditional UNIX *open()* system call used for files. Given an IPC object name, the IPC *open* call either:

- creates a new object with the given name, opens that object, and returns a handle for it; or
- opens an existing object and returns a handle for that object.

The handle returned by the IPC *open* call is analogous to the file descriptor returned by the traditional *open()* system call—it is used in subsequent calls to refer to the object.

The type of handle returned by the IPC *open* call depends on the type of object. For message queues, it is a message queue descriptor, a value of type *mqd_t*. For semaphores, it is a pointer of type *sem_t* *. For shared memory, it is a file descriptor.

All of the IPC *open* calls permit at least three arguments—*name*, *oflag*, and *mode*—as exemplified by the following *shm_open()* call:

```
fd = shm_open("/mymem", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
```

These arguments are analogous to the arguments of the traditional UNIX *open()* system call. The *name* argument identifies the object to be created or opened. The *oflag* argument is a bit mask that can include at least the following flags:

O_CREAT
: Create the object if it doesn't already exist. If this flag is not specified and the object doesn't exist, an error results (ENOENT).

O_EXCL
: If O_CREAT is also specified and the object already exists, an error results (EEXIST). The two steps—check for existence and creation—are performed atomically (Section 5.1). This flag has no effect if O_CREAT is not specified.

Depending on the type of object, *oflag* may also include one of the values O_RDONLY, O_WRONLY, or O_RDWR, with meanings similar to *open()*. Additional flags are allowed for some IPC mechanisms.

The remaining argument, *mode*, is a bit mask specifying the permissions to be placed on a new object, if one is created by the call (i.e., O_CREAT was specified and the object did not already exist). The values that may be specified for *mode* are the same as for files (Table 15-4, on page 295). As with the *open()* system call, the permissions mask in *mode* is masked against the process umask (Section 15.4.6). The ownership and group ownership of a new IPC object are taken from the effective user and group IDs of the process making the IPC *open* call. (To be strictly accurate, on Linux, the ownership of a new POSIX IPC object is determined by the process's file-system IDs, which normally have the same value as the corresponding effective IDs. Refer to Section 9.5.)

> On systems where IPC objects appear in the standard file system, SUSv3 permits an implementation to set the group ID of a new IPC object to the group ID of the parent directory.

Closing an IPC object

For POSIX message queues and semaphores, there is an IPC *close* call that indicates that the calling process has finished using the object and the system may deallocate any resources that were associated with the object for this process. A POSIX shared memory object is closed by unmapping it with *munmap()*.

IPC objects are automatically closed if the process terminates or performs an *exec()*.

IPC object permissions

IPC objects have a permissions mask that is the same as for files. Permissions for accessing an IPC object are similar to those for accessing files (Section 15.4.3), except that execute permission has no meaning for POSIX IPC objects.

Since kernel 2.6.19, Linux supports the use of access control lists (ACLs) for setting the permissions on POSIX shared memory objects and named semaphores. Currently, ACLs are not supported for POSIX message queues.

IPC object deletion and object persistence

As with open files, POSIX IPC objects are *reference counted*—the kernel maintains a count of the number of open references to the object. By comparison with System V IPC objects, this makes it easier for applications to determine when the object can be safely deleted.

Each IPC object has a corresponding *unlink* call whose operation is analogous to the traditional *unlink()* system call for files. The *unlink* call immediately removes the object's name, and then destroys the object once all processes cease using it (i.e., when the reference count falls to zero). For message queues and semaphores, this means that the object is destroyed after all processes have closed the object; for shared memory, destruction occurs after all processes have unmapped the object using *munmap()*.

After an object is unlinked, IPC *open* calls specifying the same object name will refer to a new object (or fail, if O_CREAT was not specified).

As with System V IPC, POSIX IPC objects have kernel persistence. Once created, an object continues to exist until it is unlinked or the system is shut down. This allows a process to create an object, modify its state, and then exit, leaving the object to be accessed by some process that is started at a later time.

Listing and removing POSIX IPC objects via the command line

System V IPC provides two commands, *ipcs* and *ipcrm*, for listing and deleting IPC objects. No standard commands are provided to perform the analogous tasks for POSIX IPC objects. However, on many systems, including Linux, IPC objects are implemented within a real or virtual file system, mounted somewhere under the root directory (/), and the standard *ls* and *rm* commands can be used to list and remove IPC objects. (SUSv3 doesn't specify the use of *ls* and *rm* for these tasks.) The main problem with using these commands is the nonstandard nature of POSIX IPC object names and their location in the file system.

On Linux, POSIX IPC objects are contained in virtual file systems mounted under directories that have the sticky bit set. This bit is the restricted deletion flag (Section 15.4.5); setting it means that an unprivileged process can unlink only the POSIX IPC objects that it owns.

Compiling programs that use POSIX IPC on Linux

On Linux, programs employing the POSIX IPC mechanisms must be linked with the *realtime* library, *librt*, by specifying the *–lrt* option to the *cc* command.

51.2 Comparison of System V IPC and POSIX IPC

As we look at the POSIX IPC mechanisms in the following chapters, we'll compare each mechanism against its System V counterpart. Here, we consider a few general comparisons for these two types of IPC.

POSIX IPC has the following general advantages when compared to System V IPC:

- The POSIX IPC interface is simpler than the System V IPC interface.
- The POSIX IPC model—the use of names instead of keys, and the *open*, *close*, and *unlink* functions—is more consistent with the traditional UNIX file model.
- POSIX IPC objects are reference counted. This simplifies object deletion, because we can unlink a POSIX IPC object, knowing that it will be destroyed only when all processes have closed it.

However, there is one notable advantage in favor of System V IPC: portability. POSIX IPC is less portable than System V IPC in the following respects:

- System V IPC is specified in SUSv3 and supported on nearly every UNIX implementation. By contrast, each of the POSIX IPC mechanisms is an optional component in SUSv3. Some UNIX implementations don't support (all of) the POSIX IPC mechanisms. This situation is reflected in microcosm on Linux: POSIX shared memory is supported only since kernel 2.4; a full implementation of POSIX semaphores is available only since kernel 2.6; and POSIX message queues are supported only since kernel 2.6.6.

- Despite the SUSv3 specification for POSIX IPC object names, the various implementations follow different conventions for naming IPC objects. These differences require us to do (a little) extra work to write portable applications.
- Various details of POSIX IPC are not specified in SUSv3. In particular, no commands are specified for displaying and deleting the IPC objects that exist on a system. (In many implementations, standard file-system commands are used, but the details of the pathnames used to identify IPC objects vary.)

51.3 Summary

POSIX IPC is the general name given to three IPC mechanisms—message queues, semaphores, and shared memory—that were devised by POSIX.1b as alternatives to the analogous System V IPC mechanisms.

The POSIX IPC interface is more consistent with the traditional UNIX file model. IPC objects are identified by names, and managed using *open*, *close*, and *unlink* calls that operate in a manner similar to the analogous file-related system calls.

POSIX IPC provides an interface that is superior in many respects to the System V IPC interface. However, POSIX IPC is somewhat less portable than System V IPC.