XSI IPC associates an `ipc_perm` structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
struct ipc_perm {
  uid_t  uid;  /* owner's effective user id */
  gid_t  gid;  /* owner's effective group id */
  uid_t  cuid; /* creator's effective user id */
  gid_t  cgid; /* creator's effective group id */
  mode_t mode; /* access modes */
  .
  .
  .
};
```

Each implementation includes additional members. See `<sys/ipc.h>` on your system for the complete definition.

All the fields are initialized when the IPC structure is created. At a later time, we can modify the `uid`, `gid`, and `mode` fields by calling `msgctl`, `semctl`, or `shmctl`. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling `chown` or `chmod` for a file.

The values in the `mode` field are similar to the values we saw in Figure 4.6, but there is nothing corresponding to execute permission for any of the IPC structures. Also, message queues and shared memory use the terms *read* and *write*, but semaphores use the terms *read* and *alter*. Figure 15.24 shows the six permissions for each form of IPC.

Figure 15.24. XSI IPC permissions

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

15.8. Semaphores

A semaphore isn't a form of IPC similar to the others that we've described (pipes, FIFOs, and message queues). A semaphore is a counter used to provide access to a shared data object for multiple processes.

> The Single UNIX Specification includes an alternate set of semaphore interfaces in the semaphore option of its real-time extensions. We do not discuss these interfaces in this text.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.

A common form of semaphore is called a *binary semaphore*. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (`semget`) is independent of its initialization (`semctl`). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated.

The *undo* feature that we describe later is supposed to handle this.

The kernel maintains a `semid_ds` structure for each semaphore set:

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
  unsigned short   sem_nsems; /* # of semaphores in set */
  time_t           sem_otime; /* last-semop() time */
  time_t           sem_ctime; /* last-change time */
  .
  .
  .
};
```

The Single UNIX Specification defines the fields shown, but implementations can define additional members in the `semid_ds` structure.

Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct {
  unsigned short  semval;   /* semaphore value, always >= 0 */
  pid_t           sempid;   /* pid for last operation */
  unsigned short  semncnt;  /* # processes awaiting semval>curval */
  unsigned short  semzcnt;  /* # processes awaiting semval==0 */
  .
  .
  .
};
```

Figure 15.28 lists the system limits (Section 15.6.3) that affect semaphore sets.

Figure 15.28. System limits that affect semaphores

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum value of any semaphore	32,767	32,767	32,767	32,767
The maximum value of any semaphore's adjust-on-exit value	16,384	32,767	16,384	16,384

Description	Typical values			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
The maximum number of semaphore sets, systemwide	10	128	87,381	10
The maximum number of semaphores, systemwide	60	32,000	87,381	60
The maximum number of semaphores per semaphore set	60	250	87,381	25
The maximum number of undo structures, systemwide	30	32,000	87,381	30
The maximum number of undo entries per undo structures	10	32	10	10
The maximum number of operations per `semop` call	100	32	100	10

The first function to call is `semget` to obtain a semaphore ID.

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

int semget(key_t key, int nsems, int flag);
```

Returns: semaphore ID if OK, 1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and discussed whether a new set is created or an existing set is referenced. When a new set is created, the following members of the `semid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `sem_otime` is set to 0.
- `sem_ctime` is set to the current time.
- `sem_nsems` is set to *nsems*.

The number of semaphores in the set is *nsems*. If a new set is being created (typically in the server), we must specify *nsems*. If we are referencing an existing set (a client), we can specify *nsems* as 0.

The `semctl` function is the catchall for various semaphore operations.

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

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

Returns: (see following)

The fourth argument is optional, depending on the command requested, and if present, is of type `semun`, a union of various command-specific arguments:

```
union semun {
  int               val;    /* for SETVAL */
  struct semid_ds *buf;     /* for IPC_STAT and IPC_SET */
  unsigned short  *array;   /* for GETALL and SETALL */
};
```

Note that the optional argument is the actual union, not a pointer to the union.

The *cmd* argument specifies one of the following ten commands to be performed on the set specified by *semid*. The five commands that refer to one particular semaphore value use *semnum* to specify one member of the set. The value of *semnum* is between 0 and *nsems-1*, inclusive.

`IPC_STAT`	Fetch the `semid_ds` structure for this set, storing it in the structure pointed to by *arg.buf*.
`IPC_SET`	Set the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` fields from the structure pointed to by *arg.buf* in the `semid_ds` structure associated with this set. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.
`IPC_RMID`	Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of `EIDRM` on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.
`GETVAL`	Return the value of `semval` for the member *semnum*.
`SETVAL`	Set the value of `semval` for the member *semnum*. The value is specified by *arg.val*.
`GETPID`	Return the value of `sempid` for the member *semnum*.
`GETNCNT`	Return the value of `semncnt` for the member *semnum*.
`GETZCNT`	Return the value of `semzcnt` for the member *semnum*.

GETALL	Fetch all the semaphore values in the set. These values are stored in the array pointed to by *arg.array*.
SETALL	Set all the semaphore values in the set to the values pointed to by *arg.array*.

For all the GET commands other than GETALL, the function returns the corresponding value. For the remaining commands, the return value is 0.

The function semop atomically performs an array of operations on a semaphore set.

[View full width]

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

int semop(int semid, struct sembuf semoparray[],
size_t nops);
```

Returns: 0 if OK, 1 on error

The *semoparray* argument is a pointer to an array of semaphore operations, represented by sembuf structures:

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

The *nops* argument specifies the number of operations (elements) in the array.

The operation on each member of the set is specified by the corresponding sem_op value. This value can be negative, 0, or positive. (In the following discussion, we refer to the "undo" flag for a semaphore. This flag corresponds to the SEM_UNDO bit in the corresponding sem_flg member.)

1. The easiest case is when sem_op is positive. This case corresponds to the returning of resources by the process. The value of sem_op is added to the semaphore's value. If the undo flag is specified, sem_op is also subtracted from the semaphore's adjustment value for this process.
2. If sem_op is negative, we want to obtain resources that the semaphore controls.

   If the semaphore's value is greater than or equal to the absolute value of sem op (the resources

are available), the absolute value of `sem_op` is subtracted from the semaphore's value. This guarantees that the resulting value for the semaphore is greater than or equal to 0. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.

If the semaphore's value is less than the absolute value of `sem_op` (the resources are not available), the following conditions apply.

a. If `IPC_NOWAIT` is specified, `semop` returns with an error of `EAGAIN`.

b. If `IPC_NOWAIT` is not specified, the `semncnt` value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.

i. The semaphore's value becomes greater than or equal to the absolute value of `sem_op` (i.e., some other process has released some resources). The value of `semncnt` for this semaphore is decremented (since the calling process is done waiting), and the absolute value of `sem_op` is subtracted from the semaphore's value. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.

ii. The semaphore is removed from the system. In this case, the function returns an error of `EIDRM`.

iii. A signal is caught by the process, and the signal handler returns. In this case, the value of `semncnt` for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of `EINTR`.

3. If `sem_op` is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.

If the semaphore's value is currently 0, the function returns immediately.

If the semaphore's value is nonzero, the following conditions apply.

a. If `IPC_NOWAIT` is specified, return is made with an error of `EAGAIN`.

b. If `IPC_NOWAIT` is not specified, the `semzcnt` value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.

i. The semaphore's value becomes 0. The value of `semzcnt` for this semaphore is decremented (since the calling process is done waiting).

ii. The semaphore is removed from the system. In this case, the function returns an error of `EIDRM`.

iii. A signal is caught by the process, and the signal handler returns. In this case, the value of `semzcnt` for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of `EINTR`.

The `semop` function operates atomically; it does either all the operations in the array or none of them.

Semaphore Adjustment on `exit`

As we mentioned earlier, it is a problem if a process terminates while it has resources allocated through a semaphore. Whenever we specify the `SEM_UNDO` flag for a semaphore operation and we allocate resources (a `sem_op` value less than 0), the kernel remembers how many resources we allocated from that particular semaphore (the absolute value of `sem_op`). When the process terminates, either voluntarily or involuntarily, the kernel checks whether the process has any outstanding semaphore adjustments and, if so, applies the adjustment to the corresponding semaphore.

If we set the value of a semaphore using `semctl`, with either the `SETVAL` or `SETALL` commands, the adjustment value for that semaphore in all processes is set to 0.

ExampleTiming Comparison of Semaphores versus Record Locking

If we are sharing a single resource among multiple processes, we can use either a semaphore or record locking. It's interesting to compare the timing differences between the two techniques.

With a semaphore, we create a semaphore set consisting of a single member and initialize the semaphore's value to 1. To allocate the resource, we call `semop` with a `sem_op` of -1; to release the resource, we perform a `sem_op` of +1. We also specify `SEM_UNDO` with each operation, to handle the case of a process that terminates without releasing its resource.

With record locking, we create an empty file and use the first byte of the file (which need not exist) as the lock byte. To allocate the resource, we obtain a write lock on the byte; to release it, we unlock the byte. The properties of record locking guarantee that if a process terminates while holding a lock, then the lock is automatically released by the kernel.

Figure 15.29 shows the time required to perform these two locking techniques on Linux. In each case, the resource was allocated and then released 100,000 times. This was done simultaneously by three different processes. The times in Figure 15.29 are the totals in seconds for all three processes.

On Linux, there is about a 60 percent penalty in the elapsed time for record locking compared to semaphore locking.

Even though record locking is slower than semaphore locking, if we're locking a single resource (such as a shared memory segment) and don't need all the fancy features of XSI semaphores, record locking is preferred. The reasons are that it is much simpler to use, and the system takes care of any lingering locks when a process terminates.

Figure 15.29. Timing comparison

of locking alternatives on Linux

Operation	User	System	Clock
semaphores with undo	0.38	0.48	0.86
advisory record locking	0.41	0.95	1.36