System V IPC

Unix Systems Programming

CSPP 51081
System V IPC

Overview
System V InterProcess Communication (IPC)

- Provides three mechanisms for sharing data between processes
  - message queues (similar to a bi-directional pipe)
  - semaphore sets (shared counter variables for synchronization)
  - shared memory segment
- System V IPC structures can be shared among any processes on the same system.
- Each IPC mechanism has a common interface for creating, controlling, and removing.
- Each IPC structure is implemented by the Kernel and referred to by a unique non-negative integer, an identifier.
<table>
<thead>
<tr>
<th>mechanism</th>
<th>function</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>message queues</td>
<td>msgget</td>
<td>create or access</td>
</tr>
<tr>
<td></td>
<td>msgctl</td>
<td>control</td>
</tr>
<tr>
<td></td>
<td>msgsnd</td>
<td>send message</td>
</tr>
<tr>
<td></td>
<td>msgrcv</td>
<td>receive message</td>
</tr>
<tr>
<td>semaphores</td>
<td>semget</td>
<td>create or access</td>
</tr>
<tr>
<td></td>
<td>semctl</td>
<td>control</td>
</tr>
<tr>
<td></td>
<td>semop</td>
<td>execute operation (wait or signal)</td>
</tr>
<tr>
<td>shared memory</td>
<td>shmget</td>
<td>create or access</td>
</tr>
<tr>
<td></td>
<td>shmctl</td>
<td>control</td>
</tr>
<tr>
<td></td>
<td>shmat</td>
<td>attach memory to process</td>
</tr>
<tr>
<td></td>
<td>shmdt</td>
<td>detach memory to process</td>
</tr>
</tbody>
</table>
**IPC identifiers and keys**

- Each IPC *structure* in the kernel is referred to by a nonnegative integer *identifier* which is unique among all structures of that IPC *IPC*.
- The *identifier* is obtained when the structure is created by a call to `XXXget` (where `XXX` is one of `msg`, `sem`, or `shm`).
- The *identifier* is used by all other IPC functions to reference this structure.
- Because processes need to share IPC structures and they cannot know the identifier ahead of time, whenever a structure is created or accessed by `XXXget` a *key* is specified as an argument.
- The type of *key* is `key_t` (probably a long integer), and is converted by the kernel into an *identifier*. 
Picking the key

There are three ways of picking a key

1. Let the kernel pick the key by using the key `IPC_PRIVATE` and pass the identifier directly to other processes. Using the key `IPC_PRIVATE` guarantees a new IPC structure is created. (This method is most useful when the children of the process will use the structure, since the identifier can be stored in a variable which is copied to the children.)

2. Choose the key value directly. (The problem is that the key value may already be in use.)

3. Ask the kernel to generate a key from a specified `path` and one byte character project ID, using the function `ftok(3)`. The file corresponding to `path` must exist and be accessible to all processes that want to access the IPC structure. (The project ID allows several structures to be keyed to the same path.)
SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(char *path, char id);

Return:

0 on success       -1 on error

Possible errors are due to problems with path. See man 3 ftok.
Accessing IPC resources from the shell

- A strong disadvantage of IPC structures is that they remain in the system, unless explicitly removed by the process (using \texttt{XXXcntl}) or at the shell prompt (using the utility \texttt{ipcrm}).

- To obtain information about the system limits, use the utility \texttt{ipcs \textendash l}.

- To find-out about the IPC structures allocated by the system on which you have read permission, use \texttt{ipcs}. (see \texttt{man ipcs}).

- To remove an IPC structure use \texttt{ipcrm [msg | shm | sem] id} (the identifier can be obtained by calling \texttt{ipcs}.) Since IPC structures of different types can have the same identifier, you must specify the structure type. (see \texttt{man iprm}).

- IT IS IMPERATIVE THAT YOU REMOVE IPC STRUCTURES.
Permission Structure

- All IPC structures have an owner, the user ID of the process which created the structure, and group, the group ID of the process which created the structure.
- All IPC structures have a 9-bit permission access mode for owner, group and everyone else. The permission flags are just as in `open`, except only read and write permissions make sense. For semaphores, permission to write to the semaphore set means permission to alter the semaphore.
- Permissions are set in the `mode` field of `xxxget`.
- The flags `IPC_CREAT` and `IPC_EXCL` have the same meaning as with `open` and are bit-wise or’d to the permissions. `IPC_CREAT` creates the IPC structure if it does no exist; the combination `IPC_CREAT` and `IPC_EXCL` ensure the structure is new, or `xxxget` returns an error.
Semaphores

- A semaphore is a shared counting variable (of type integer) which restricts the number of processes which access a shared data object. If access must be exclusive to a single process, the semaphore is called a mutex, and otherwise is called a counting semaphore.

- The steps for obtaining access to a shared resource using a semaphore is as follows
  1. Test the semaphore to see if it is positive.
  2. If the value of the semaphore is positive, the process can access the resource, and decrements the value of the semaphore.
  3. If the value of the semaphore is zero, the process waits on a queue until the semaphore value is greater than zero and the process is given permission to decrement the value of the semaphore.
  4. When the process finishes with the shared resource, it increments the semaphore value (allowing another process to pass the semaphore.) This step never blocks.
Semaphore Sets

- The operation of testing and modifying a semaphore is *atomic*—it always takes place as a single step. For this reason semaphores are implemented inside the kernel.

- System V semaphores are a *set* of semaphores which allow the testing and setting of any number of the semaphores in the set in a single operation.

- The number of semaphores in the set are determined when the set is created.

- On *admiral* there is a maximum of 128 semaphore sets allowed in the system. Each set can have at most 250 semaphores. Up 32 of the semaphores in the set can be tested and modified atomically.
semget : create or obtain a semaphore set

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

int semget( key_t key, int nsems, int mode);

Return:

semid, identifier on success -1 on error

see man 2 semget for possible errors
Obtaining a Semaphore Set

Let $nsems$ be the number of semaphores in the set and $perm$ be the permissions and $key$ be a key value. Then there are 4 ways to obtain a semaphore set:

- Let the kernel choose the key and guarantee the semaphore set is new
  
  ```c
  semid = semget(IPC_PRIVATE, nsems, perm);
  ```

- Obtain the semaphore set associated with $key$, creating if it does not exist
  
  ```c
  semid = semget(key, nsems, IPC_CREAT | perm);
  ```

- Obtain the semaphore set associated with $key$ guaranteeing it is new or return with error EEXIST
  
  ```c
  semid = semget(key, nsems, IPC_CREAT | IPC_EXCL | perm);
  ```

- Obtain the existing semaphore set associated with $key$
  
  ```c
  semid = semget(key, 0, 0);
  ```
**semctl: The catchall of semaphore operations**

**SYNOPSIS**

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

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

**Return:**

0 on success -1 on error

`semctl` has many different uses, all return non-negative on success. We will only be using the system call to initialize semaphore sets and remove semaphore sets. See `man 2 semctl`
union semun: the union that isn’t

There are ten possible values to *cmd* argument of *semctl*. We will only be interested in three here. Some of these arguments require the fourth argument and others ignore it. The union type, `union semun`, is probably not defined so you will need to define it in your program:

```c
union semun {
    int val;    /* used for SETVAL */
    struct semid_ds *buf; /* We won’t use */
    ushort *array; /* used for SETALL */
}
```
Initializing a Semaphore Set

Initially, the semaphores in the set have the value 0. If you are using the semaphore to synchronize access to a shared resource, you need to set this value to the maximum number of processes that can access the resource at a single time. There are two ways to do this with semctl (a third way using semop will be described below):

- Set a single semaphore in the set. The semaphores are numbered from 0 to \(semnum - 1\). To set semaphore \(n\) to value \(v\):
  
  ```c
  union semun arg;
  arg.val = v;
  semctl(semid, n, IPC_SETVAL, arg);
  ```

- Set values to all the semaphores in the set.
  
  ```c
  union semun arg;
  /* Assign arg.array[n] to the value of the (n+1)st semaphore */
  semctl(semid, 0, IPC_SETALL, arg);
  ```
Removing a Semaphore Set

Semaphore sets remain in the system until removed. Since there is a small system-wide limit to the number of semaphore sets, it is important to remove them when not using them. There are two ways of removing a semaphore set

- On the shell command line, to remove the semaphore set with the identifier \textit{id},
  \begin{verbatim}
  ipcrm sem id
  \end{verbatim}

- In a running process, to remove a semaphore set with identifier \textit{id}.
  \begin{verbatim}
  semctl(id, 0, IPC_RMID);
  \end{verbatim}
  Note that it is not necessary to specify the final argument.
  If a semaphore is removed by a process, then all processes which were waiting on a semaphore will return with the error \texttt{ERMI}}
semop: semaphore operations

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

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

Return:
0 on success       -1 on error

where struct sembuf include the following members

short sem_num;    /* semaphore number: 0 = first */
short sem_op;     /* semaphore operation */
short sem_flg;    /* IPC_NOWAIT, SEM_UNDO */
Semaphore Operations

- *sops* is an array of *nops* operations to perform on the semaphores of the set. The operations are done atomically (up to some maximum number, 32 on *admiral*) and are either all performed or none are performed.
- The *sem_num* field of **struct buf** specifies the semaphore to perform the operation upon and ranges from 0 to *semnum*–1, where *semnum* is the size of the set.
- The *sem_op* field of **struct buf** is an integer value that will be added to the current value of the semaphore, or the process will be forced to wait if this value would be negative.
- The flag is usually left at 0, although IPC_NOWAIT will force a return with error EAGAIN if the process would have waited for a change in the semaphore value. (There is a flag SEM_UNDO which I will not describe.)
Semaphore Operations

- **sem_op** is positive. This corresponds to returning resources by the process. The value of **sem_op** is added to the semaphore’s value. This operation never blocks.

- **sem_op** is negative. This corresponds to obtaining resources by the process. If adding **sem_op** to the semaphore’s current value is non-negative, then the operation is performed; otherwise, the operation blocks until the semaphore’s value is at least as great as the absolute value of **sem_op**.

- **sem_op** is zero then the process will wait until the value of the semaphore is 0 before returning.
Examples of Semaphore Operations

- The *down* or *wait* operation on semaphore \( n \) of semaphore set \( semid \)
  
  ```c
  struct sembuf sops[1];
  semopr.sem_num = semnum;
  semopr.sem_op = -1;
  semopr.sem_flg = 0;
  semop(semid, sops, 1);
  ```

- The *up* or *signal* operation on semaphore \( n \) of semaphore set \( semid \)
  
  ```c
  struct sembuf sops[1];
  semopr.sem_num = semnum;
  semopr.sem_op = 1;
  semopr.sem_flg = 0;
  semop(semid, sops, 1);
  ```
Example of Multiple Semaphore Operations

The power of System V semaphores is that they can be used to atomically check and set multiple semaphores in one operation. The following waits on one semaphore while simultaneously signaling another semaphore:

```c
struct sembuf sops[2];
semopr.sem_num = semnum1;
semopr.sem_op = -1;
semopr.sem_flg = 0;

semopr.sem_num = semnum2;
semopr.sem_op = 1;
semopr.sem_flg = 0;

semop(semid, sops, 2);
```
Using semop to Initialize Semaphore Sets

One of the serious problems with System V IPC semaphore sets is that creation of the semaphore is distinct from initialization. Semaphores are initially assigned 0. Taking advantage of this, we can use semop to initialize semaphores. If semaphore semnum in semaphore set semid is to be initialized to \( n \):

```c
struct sembuf sops[1];
semopr.sem_num = semnum;
semopr.sem_op = n;
semopr.sem_flg = 0;
semop(semid, sops, 1);
```

It is preferable to use semctl though because (i) semctl allows all semaphores to be set in one operation and (ii) semop can be interrupted by signals, so requires much greater care (using a restart version, for example) then the above code indicates.
Conditions when `semop` Returns

A process may return from waiting on performing an array of semaphore operations under the following conditions:

- The operations were successfully performed, so `semop` returned 0.
- The flag `IPC_NOWAIT` set and the process would have waited to perform the semaphore. In this case, `semop` returns -1 and sets `errno` to `EAGAIN`.
- `semop` was interrupted by a signal, in which case it returns with the value -1 and sets `errno` to `EINTR`. You may want to restart the operation:
  ```c
  while(semop(semid, sops, nsops)<0 && errno==EINTR);
  ```
- Some process removed the semaphore set, in which case `semop` returns -1 and sets `errno` to `ERMID`. 