Basics of System V Semaphore

A semaphore is an inter-process synchronisation mechanism used to inform processes of the occurrence of a particular event. This article aims to explain semaphore related system calls and relevant data structures with the help of simple example programs.

My last article in *LINUX For You*, 'Sharing Data with Shared Memory,' dealt with the concept of shared memory. Actually, efficient usage of shared memory is not possible without the help of a semaphore. A semaphore is a synchronisation mechanism that implements mutual exclusion among processes to avoid race conditions while accessing any shared resource. For example, the synchronised modification of a shared memory, which is being concurrently accessed by many processes, can only be done by using a semaphore.

System V Semaphore creates a semaphore set, where many semaphores can be created and each semaphore may be either binary or counting, depending on the number of critical resources. To implement mutual exclusion, the binary semaphore is used. If many shared resources are to be synchronised—for instance, if three printers are shared by fifty desktop systems—then a counting semaphore is an appropriate choice, where the initial value of the semaphore is to be set as three.

Semaphore implementation and working procedure is relatively complex compared to message queues and shared memory. So the topic has been split into two parts—namely, "Basics of System V Semaphore" and "Intricacies of System V Semaphore". This article explains semaphore related system calls, relevant data structures and simple example programs to implement mutual

exclusion. In the next part, which will appear in a forthcoming issue of *LFY,* more complex practical programs and the internals of the system calls with kernel source code will be discussed.

System calls for a semaphore

To create a semaphore set, the *semget* system call is used. To do semaphore operations such as p ( ) or v ( ), the *semop* function is used, and to set or get the value of semaphore(s) and to perform control operations, the *semctl* system call is used.

semget

The syntax for the system call is:

```
int semget (key_t key, int
NumberOfSemaphore, int
SemaphoreFlag);
```

The first argument, *key,* has already been discussed in previous articles of this series. The second argument represents the number of semaphores in a set, and the third argument is a semaphore flag. IPC_CREAT is used to create a new semaphore set. When IPC_CREAT | IPC_EXCL is used, the system call returns -1 if the semaphore set already exists; otherwise, the call creates a new semaphore set and returns zero.

semctl

The *semctl* system call accepts either three or four arguments. The syntax for the call is as follows:

```
int semctl (int SemaphoreID, int
SemaphoreNumber, int Command, union
semun);
```

…where the structure of *union semun* is:

```
union semun {
        int val;
        struct semid_ds *buffer;
        unsigned short  *array;
        strcut seminfo  *__buffer;
};
```

The first member of the *union semun, val* sets the value of a semaphore. Here is where programmers decide whether they need the binary or counting semaphore. If the *val* is assigned to 1, then it becomes a binary semaphore and is used to perform mutual exclusion; if the value is >1 then the semaphore is a counting semaphore and the initial value is to be equivalent to the number of shared resources.

The second member, *buffer,* is a data structure describing a set of semaphores. It is used to get the created semaphore's status information. If users want to change attributes of the created semaphore, they can fill the *semid_ds* structure and pass it on to the system for modification.. The structure of *semid_ds* is described below:

```
struct semid_ds {
        struct   ipc_perm sem_perm;
        __time_t sem_otime;
        __time_t sem_ctime;
        unsigned long int sem_nsems;
};
```

The first member of the *semid_ds* structure has members like *user id, group id, access permission, number of semaphores in a set, creation time*, etc. The time of the last semaphore's operations (*semop* call), the last time it was changed by the *semctl* function and the total number of semaphores in the set are assigned to the second, third and fourth members of the *semid_ds* structure, respectively.

The third member of the *union semun, array* is used to set or get the value for all the created semaphores by passing the *SETALL* or *GETALL* command correspondingly to the *semctl* function.

The fourth member of the *union semun* is a Linux specific part and is used to fill the *seminfo* structure, when an IPC_STAT command is passed into the *semctl* function. The *seminfo* structure is given below:

```
struct  seminfo {
    int semmap;
    int semmni;
    int semmns;
    int semmnu
    int semmsl;
    int semopm;
    int semume;
    int semusz;
    int semvmx;
    int semaem;
};
```

The brief description of each member of the *seminfo* structure is as follows:

semmap—number of semaphore map entries used to manage semaphore sets.

semmni—maximum number of active semaphore sets.

semmns—maximum number of semaphores in the system.

semmnu—number of undo structures in the system.

semmsl—maximum number of semaphores per semaphore set, SEMMNI and SEMMSL, are usually chosen so that their product equals SEMMNS.

semopm—maximum number of operations allowed for one *semop* call.

semume—maximum number of undo entries per process.

semusz—size in bytes of the undo structure.

semvmx—maximum semaphore value.

semaem—adjust on exit maximum value.

The third argument of the *semctl* system call is a command. Some of the possible commands are *IPC_STAT*, *IPC_SET* and *IPC_RMID*. These commands have been discussed in the articles on shared memory and message queues; hither also the functionality of these commands are almost same. So the remaining semaphore's specific commands are taken here for discussion.

SETVAL—sets the given value to the semaphore of the set.

GETNCNT—returns the value of

the number of processes waiting for an increase value of the semaphore in the set.

GETPID—the ID of the process that executed the last semaphore operation (*semop* system call).

GETVAL—returns the current value of the semaphore.

GETZCNT—returns the number of threads waiting for the value of the semaphore *semaphorenumber* to reach zero. The GETZCNT command requires read permission to the semaphore set. The remaining commands, *SETALL* and *GETALL,* have been explained earlier.

semop

The syntax of the *semop* system call is as follows:

```
int semop ( int SemaphoreID, struct
sembuf *buffer, unsigned Nbuffer );
```

Locking or unlocking a critical section by means of decrementing or incrementing the semaphore's count value is done by the *semop*. This function is executed atomically and performs operations on selected members of the semaphore set indicated by SemaphoreID (the return value of *semget* call). The second argument is a pointer to the array of semaphore operations structure *sembuf*, which has the following three members:

```
struct sembuf {
short sem_num;
short sem_op;
short sem_flg;
};
```

The first member, *sem_num,* is used to specify the number of semaphores in the semaphore set. The second member, *sem_op,* specifies whether the semaphore operation is incrementing or decrementing the value of the semaphore. The third member of the *sembuf* structure, *sem_flg,* is the operation flag that is used to specify whether the calling process can wait for the semaphore or not.

The third argument in the *semop* system call, *Nbuffer,* specifies the number of *sembuf* structures in the array.

The binary semaphore for mutual exclusion

Binary semaphores are used to provide exclusive locking for a thread on a resource. To implement access of a shared section of a code by only one process at a time, you need to write two programs—one for initialisation (which creates a semaphore set and sets the value of each semaphore) and a second program that can use the ID of the already created semaphore, and perform locking and unlocking operations.

Initialisation

```
1. #include <sys/sem.h>
2. main () {
3. int key, semid;
4. key = ftok(".", 'a');
5. semid = semget (key, 1,
IPC_CREAT|IPC_EXCL|0744);
```

Here, the *semget* system call creates a single semaphore in a semaphore set with access permission 744. Since IPC_EXCL is ORed with IPC_CREAT, when the same call is called again, it will not create a new semaphore ID, but will return an error message. After creation of the semaphore, a specific value has to be set to the semaphore. Programmers can decide the value of the semaphore depending on whether they have single or multiple shared resources.

```
6. semctl(semid, 0, SETVAL, 1));
```

The *semctl* function sets the particular semaphore's value in the set. Here, '0' is passed since only one semaphore is created in the semaphore set and the identification of the first semaphore is '0'. The value of the semaphore is 1, which is set to the semaphore by SETVAL command in the *semctl* function. Instead of the SETVAL command, a programmer can also use a corresponding numerical value, which is defined in *<bits/sem.h>*.

For example, the same *semctl* function can be written as:

```
6. semctl(semid, 0, 16, 1));
```

The other commands and their corresponding numerical values are given below:

```
#define GETPID          11
#define GETVAL          12
#define GETALL          13
#define GETNCNT         14
#define GETZCNT         15
#define SETALL          17
```

These six lines can create a binary semaphore and, after execution of the program, the programmer can check the existence of the semaphore and its details by executing the *ipcs –s* command in the shell.

Accessing a semaphore

```
1. #include <sys/sem.h>
2. main () {
3. int key, semid;
4. struct sembuf buf = {0, -1, 0};
```

The first member of the structure indicates the particular semaphore in the set; in this case, it is the first semaphore. The second member indicates that the value of the semaphore is decrementing, and the third member determines as what the *semop* function will do if the semaphore is busy.

If the value is 0: it will wait until the semaphore is available.

If the value is IPC_NOWAIT: it will not wait for the semaphore, returns with error.

Sometimes, if a process uses the semaphore and exits abnormally before releasing the semaphore, then the other processes that are waiting for the semaphore will wait forever (in the case of *buf.sem_flg = 0*). The waiting processes can be killed by the *SIGKILL* signal but the semaphore's value cannot be incremented. Therefore, the created semaphore set is unusable. To solve this issue,

SEM_UNDO is assigned instead of '0'. This *SEM_UNDO* flag will automatically release the semaphore if a process exits without releasing it.

```
5. key = ftok (".", 'a');
6. semid = semget (key, 1, 0);
```

Since the semaphore set is already created with a single semaphore in the initialisation program, therefore, in the *semget* system call, '0' is passed as the last argument.

```
7. printf (" Before Locking....Waiting
for unlock\n");
8. semop (semid, &buf, 1);
```

Here, the *semop* system call implements locking, the semaphore's decrementing operation is already specified in the *sembuf* structure, and the address of the structure is passed as an argument into the system call.

```
9. printf ("Inside locking.....Enter to
unlock...\n");
10. getchar();
```

The Line 10 is a virtual critical section. Programmers can write the critical section of their code here. The library function *getchar* waits for the *Enter* key, and if the same executable program is being executed by another terminal, the control stops at Line 8, as the critical section is already locked. The second terminal user cannot see the output of Line 9.

```
11. buf.sem_op = 1;
12. semop (semid, &buf, 1);
13. printf (".....unlocked...\n");
}
```

After the critical section access is over, the value of the semaphore must be incremented. First the value of the semaphore is reassigned in the *sembuf* structure and then the *semop* system call is implemented. In this case, the previous value of the semaphore has been zero and now 1 should be added to the previous value, so that the next waiting process for the semaphore can enter into the critical section after decrementing the value of the semaphore.

The basics of semaphores have been explained here with the help of a simple binary semaphore program. If you have to synchronise multiple critical sections in a project, then you need to create many semaphores in a set. Creation of many semaphores in a set, setting a value for each semaphore, the effective usage of each semaphore in a set with example programs, and the internals of the semaphore related system calls are topics that we will be taking up in forthcoming articles on semaphores.

By: Dr B. Thangaraju. *He can be reached at balat.raju@wipro.com*