The Intricacies of the System V Semaphore

This article explains the semaphore related data structures and suggests suitable programs to implement the semaphore set.

Semaphore is a principal synchronisation mechanism for concurrent access of shared resources. Semaphores are classified as binary and counting semaphores. A binary semaphore is used to implement mutual exclusion and the counting semaphore is used when many shared resources are to be synchronised.

For example, if we want to book a train ticket, the train data is in the central database so that many users can access it at the same time. This is why we have many counters to book our tickets. But if two people are allowed to book a ticket to the same destination, on the same date and for the same train, then it could lead into problems if both receive a ticket with the same seat number. To avoid this, when the first user enters the records to book the ticket, the other is blocked until the first user exits. To accomplish this kind of environment, a binary semaphore is used.

Another approach of semaphores is resource counting. If there are many shared resources to be synchronised, then a counting semaphore is a right choice. Consider an example of a university that announces the results of graduating students on its website. If a count is set to a specific number, say 1,000, this indicates that only 1,000 students can log into the website concurrently and check their results. Thereafter, any subsequent access will be locked till the count is less than a thousand.

In Linux, the System V Semaphore creates a semaphore set where many semaphores can be created, each being a binary or a counting semaphore, depending on the requirement.

Now let's consider creating a semaphore set with one binary and one counting semaphore. To understand this, let us consider another real world scenario: a team of 10 members working offshore have to take up a conference call from a client who is on-site. They all get connected to the conference call, so we need to have a counting semaphore that will have to maintain a count of the team members and synchronise access to common resources between them. But it has been noticed that during a conference call, only one person is given access to talk at a particular time and the rest are not allowed access. So a binary semaphore is required in order to achieve mutual exclusion amongst the team members.

Considering this scenario, this article explains the semaphore related data structures and suggests suitable programs to implement the semaphore set, which may be of the binary or counting type (or a combination of the two), depending on the requirements.

In our previous article, "The Basics of the System V Semaphore", *LFY, July 2006*, page 86-89, we explained the binary semaphore's primitives; this article will explain the intricacies of complex semaphore programs.

Different functions are required to create and access a semaphore. To create a semaphore set, the *semget* function is used:

```
int semget ( key_t key , int Number of Semaphores,int SemaphoreFlag);
```

The first argument key is the unique identifier for an IPC resource. The second argument is the number of semaphores in the semaphore set, and the third argument is the semaphore flag. The 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( ) is a system call used to perform control operations on the semaphore set. The syntax of 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;
   }
```

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 greater than 1 then the semaphore is of the counting type and the initial value of the first member is set to be equivalent to the number of shared resources. The second member, the buffer, is a data structure describing a set of semaphores. The next member is the array, which is used to set or get the value for all the created semaphores by passing the SETALL or GETALL command correspondingly to *semctl* function.

The following code snippet [Program 1] explains how to create two semaphores in the semaphore set, in which one is a binary and the other a counting semaphore with an initial value of 2.

```
1  /* SemMulCr.c -This program demonstartes
2   * creation of multiple semaphores */
3  #include <stdio.h>
4  #include <sys/sem.h>
5
6  union semun {
7   int val;
8   unsigned short int *array;
9  }arg;
10
11  main() {
12   int key,semid,i,ret;
13   static ushort semarray[2]={1,2};
14
15   key=ftok(".",'m');
16   semid=semget(key,2,IPC_CREAT|0744);
17   arg.array=semarray;
18   semctl(semid,0,SETALL,arg);
19   for(i=0;i<2;i++){
20    ret=semctl(semid,i,GETVAL,0);
21    printf("Sem %d=%d\n",i,ret);
22   }
23  }
```

Notice that the semaphores have been created by passing the SETALL command as seen in Line 18 in the *semctl* function, and are retrieved through the GETVAL function in Line 20.

Now let us look at how to use these semaphores for synchronisation between two different processes. Locking or unlocking a critical section by means of decrementing or incrementing the semaphore's count value is done by the *semop* system call. The increment and the decrement operations are executed atomically on selected members of the semaphore set indicated by the SemaphoreID. The second argument of the *semop* system call 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;
      };
```

A set of operations takes place, each being performed on every semaphore in the semaphore set, where the first member of the sembuf structure is the semaphore number (here it is the first semaphore of the set, so it is Semaphore 0); and the second member, *sem_op*, specifies whether the semaphore operation is incrementing or decrementing the value of the semaphore. The third member is the *sem_flg*, which is the operation flag that is used to specify whether the calling process can wait for the semaphore or not.

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

A particular process can use a binary semaphore to access a critical section, but what if the process encounters more than one critical section? In that case, there should be more than one semaphore to synchronise multiple critical sections. Linux provides a semaphore set that can contain more than one semaphore to access multiple critical sections.

The program seen in the following code snippet [Program 2] demonstrates how the semaphore set can be used for multiple critical sections using two semaphores.

```
1  /* SemMulAcc.c - This program will demonstrate synchronize
2   * mulitple critical sections using multiple semaphores  */
3
4  #include <stdio.h>
5  #include <sys/sem.h>
6
7  main() {
8   int key,semid;
9   key = ftok(“.”,’m’);
10   semid = semget(key,2,0);
11   struct sembuf s[2]={{0,-1,0|SEM_UNDO},{1,-1,0|SEM_UNDO}};
12
13   printf(“locking semaphore1\n”);
14   semop(semid,&s[0],1);
15   printf(“Inside critical section :----semaphore1\n”);
16   printf(“ENTER to unlock semaphore1\n”);
17   getchar();
18   s[0].sem_op = 1;
19   semop(semid,&s[0],1);
20   printf(“semaphore1 unlocked\n”);
21
22   printf(“locking semaphore2\n”);
23   semop(semid,&s[1],1);
24   printf(“Inside critical section :----semaphore2\n”);
25   printf(“ENTER to unlock semaphore2\n”);
26   getchar();
27   s[1].sem_op = 1 ;
28   semop(semid,&s[1],1);
29   printf(“semaphore2 unlocked\n”);
30  }
```

As the semaphore set has already been created with two semaphores in Program 1, in Line 10 of Program 2 – in the *semget* system call—0 is passed as the last argument since the semaphore has already been created.

In Line 11 the structure is initialised with values. For the first structure, 0 is passed to indicate the first semaphore, and 1 to indicate the second semaphore in the second structure. The second member indicates that the value of the semaphore is decrementing, and the third member determines 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, but returns with an error.

Sometimes, if a process locks the semaphore and exits abnormally without unlocking, then the other processes that are waiting for the semaphore have to wait forever (in the case of *sem_flg=0*). The created semaphore set becomes unusable and the processes need to be killed forcibly. To solve this issue, SEM_UNDO can be OR'ed with 0. This SEM_UNDO flag will automatically release the semaphore if a process exits without releasing it.

The locking and unlocking operation can be performed by the *semop* system call.

Lines 17-19 are the virtual critical section-1. 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 15, as the critical section-1 is already locked. The second terminal user cannot see the output of Line 17. After the critical section-1 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 called, so that the next waiting process for the semaphore can enter the critical section after decrementing the value of the semaphore. Once the first process leaves the critical section-1, the first terminal user can see the output of Line 22, which indicates that the semaphore is released; so now the second terminal user enters the critical section and can see the output of Line 17.

Now that the binary semaphore is handled, the second semaphore, which is the counting type, should be considered. Accessing the second semaphore indicated as 1 in the *sembuf* structure, the process enters the critical section-2 after leaving the critical section-1, and the first terminal user can see the output of Line 26. The other terminal process that is accessing the 1st critical section then leaves it and enters the critical section-2. So the user in the other terminal also can see the output of Line 26. Now it can be seen that both the processes are able to enter the critical section-2, which is because a counting semaphore was created with a count of 2. If another terminal is opened and the same program is executed, and if the process tries to enter the critical section-2, then it will not be given access and at this terminal the user cannot see the output of Line 26 as it will be blocked from the second critical section.

Consider a scenario where two binary semaphores are created. This can be understood from the following code snippet [Program 3]:

```
/* SemDeadLockCr.c--This program creates two binary semaphores */

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

union semun{
 int val;
 struct semid_ds *buf;
 unsigned short int *array;
  }arg;

main() {
  int key,semid,i,ret;
 static ushort semarray[2]={1,1};

 key=ftok(".",'h');
 semid=semget(key,2,IPC_CREAT|0744);
 arg.array=semarray;
 semctl(semid,0,SETALL,arg);
 for(i=0;i<2;i++){
 ret=semctl(semid,i,GETVAL,0);
 printf("sem %d=%d\n",i,ret);
 }
}
```

We'll now create Program 4 and 5 to create the 'dead lock' situation as can be understood from Table 1.

```
/*SemDeadLock1.c----This program demonstrates the improper
 * sequence of locking leading to deadlock situation*/

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

main(){
  int key,semid;
 key = ftok(".",'h');
   semid = semget(key,2,0);
   struct sembuf s[2]={{0,-1,0|SEM_UNDO},{1,-1,0|SEM_UNDO}};

   printf("locking semaphore1\n");
   semop(semid,&s[0],1);
   printf("Inside critical section :----semaphore1\n");
   getchar();

   printf("\n locking semaphore2\n");
   semop(semid,&s[1],1);
   printf("Inside critical section :----semaphore2\n");
   getchar();

   s[0].sem_op = 1;
   semop(semid,&s[0],1);
   printf("semaphore1 unlocked\n");

   s[1].sem_op = 1 ;
   semop(semid,&s[1],1);
   printf("semaphore 2 unlocked\n");
}
```

```
/*SemDeadLock2.c----This program demonstrates the improper
 * sequence of locking leading to deadlock situation*/

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

main(){
  int key,semid;
 key = ftok(".",'h');
   semid = semget(key,2,0);
   struct sembuf s[2]={{0,-1,0|SEM_UNDO},{1,-1,0|SEM_UNDO}};

   printf("locking semaphore2\n");
   semop(semid,&s[1],1);
   printf("Inside critical section :----semaphore2\n");
   getchar();

   printf("\n locking semaphore1\n");
   semop(semid,&s[0],1);
   printf("Inside critical section :----semaphore1\n");
   getchar();

   s[1].sem_op = 1;
   semop(semid,&s[1],1);
   printf("semaphore2 unlocked\n");

   s[0].sem_op = 1 ;
   semop(semid,&s[0],1);
   printf("semaphore1 unlocked\n");
}
```

Program 4	Program 5
Lock S1	Lock S2
getchar ( )	getchar ( )
Lock S2	Lock S1
getchar ( )	getchar ( )
Unlock S1	Unlock S2
Unlock S2	Unlock S1

Table 1

Implementation: If the first process locks the first critical section using the first semaphore (S1) and later locks the second critical section using the second semaphore (S2), then unlock S1 and S2. The second process locks the first critical section using S2 and the second critical section is locked by S1. In this scenario, both the processes wait indefinitely. This leads to the classical race condition. This is depicted in Program 4 and Program 5. So, if you know how to create a 'dead lock' situation, then obviously you know how to avoid the unwanted dead lock. To avoid dead lock, don't use more than one lock inside a critical section. But if you want to do that, you should preserve the same sequence of locking in all the processes. END

Acknowledgement

The authors would like to thank Mr V Jayasurya for his critical reading of this manuscript.

By: Shobana V and Dr B Thangaraju

The authors work with Talent Transformation, Wipro Technologies, Bangalore. They can be reached at shobana.venkatesan@wipro.com and balat.raju@wipro.com, respectively.