
BAHIR DAR UNIVERSITY

FACULTY OF COMPUTING

DEPARTMENT OF SOFTWARE ENGINEERING

**Individual assignment on "Operating System and System Programing"**


| **NAME** | **ID NO** |
|---|---|
| **Abreham Simachew……..……..……..... BDU1308245** | |


**Submitted to lecturer Wendmu B.**

**Submission Date June 24/2022**

Table of Contents

1. System Call ..... 2
2. Parameters and Flags of the system Call ..... 3
3. Code implementation of the System Call ..... 6
3.1 Return Value ..... 8
3.2 Errors ..... 9
3.3 Notes ..... 10
3.4 Implementation ..... 10

## 1. What / Why / How, this system call?

A system call is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.

System call offers the services of the operating system to the user programs via API (Application Programming Interface). System calls are the only entry points for the kernel system.

In these section we will see the semctl( ) System call in detail.

```
SYSTEM CALL: semctl();
PROTOTYPE: int semctl ( int semid, int semnum, int cmd, union semun arg );
  RETURNS: positive integer on success
           -1 on error: errno = EACCESS (permission denied)
                                EFAULT (invalid address pointed to by arg argument)
                                EIDRM (semaphore set was removed)
                                EINVAL (set doesn't exist, or semid is invalid)
                                EPERM (EUID has no privileges for cmd in arg)
                                ERANGE (semaphore value out of range)
```

The semctl system call is used to perform control operations on a semaphore set. This call is analogous to the msgctl system call which is used for operations on message queues. If you compare the argument lists of the two system calls, you will notice that the list for semctl varies slightly from that of msgctl. Recall that semaphores are actually implemented as sets, rather than as single entities. With semaphore operations, not only does the IPC key need to be passed, but the target semaphore within the set as well.

Both system calls utilize a cmd argument, for specification of the command to be performed on the IPC object. The remaining difference lies in the final argument to both calls. In msgctl, the final argument represents a copy of the internal data structure used by the kernel. Recall that we used this structure to retrieve internal information about a message queue, as well as to set or change permissions and ownership of the queue. With semaphores, additional operational commands are supported, thus requiring a more complex data type as the final argument. The use of a union confuses many neophyte semaphore programmers to a substantial degree. We will dissect this structure carefully, in an effort to prevent any confusion.

## 2. Briefly describe about the list of parameters and flag

This function has three or four arguments, depending on cmd. When there are four, the fourth has the type union semun. The calling program must define this union as follows:

```
union semun {
   int            val;    /* Value for SETVAL */
   struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
   unsigned short  *array;  /* Array for GETALL, SETALL */
   struct seminfo  *__buf;  /* Buffer for IPC_INFO
                       (Linux specific) */
};
```

The first argument to semctl( ) is the key value (in our case returned by a call to semget). The semid_ds data structure is defined in <sys/sem.h> as follows:

```
struct semid_ds {
   struct ipc_perm sem_perm;  /* Ownership and permissions
   time_t          sem_otime; /* Last semop time */
   time_t          sem_ctime; /* Last change time */
   unsigned short  sem_nsems; /* No. of semaphores in set */
};
```

The second argument (semun) is the semaphore number that an operation is targeted towards. In essence, this can be thought of as an index into the semaphore set, with the first semaphore (or only one) in the set being represented by a value of zero (0).

The *ipc_perm* structure is defined in <sys/ipc.h> as follows (the highlighted fields are settable using **IPC_SET**):

```
struct ipc_perm {
   key_t key;          /* Key supplied to semget() */
   uid_t uid;          /* Effective UID of owner */
   gid_t gid;          /* Effective GID of owner */
   uid_t cuid;         /* Effective UID of creator */
   gid_t cgid;         /* Effective GID of creator */
   unsigned short mode;  /* Permissions */
   unsigned short seq;   /* Sequence number */
};
```

The cmd argument represents the command to be performed against the set and valid values for cmd are:

**IPC_STAT**
Retrieves the semid_ds structure for a set, and stores it in the address of the buf argument in the semun union.

**IPC_SET**
Sets the value of the ipc_perm member of the semid_ds structure for a set. Takes the values from the buf argument of the semun union.

**IPC_RMID**
Removes the set from the kernel.

**GETALL**
Used to obtain the values of all semaphores in a set. The integer values are stored in an array of unsigned short integers pointed to by the array member of the union.

**GETNCNT**
Returns the number of processes currently waiting for resources.

**GETPID**
Returns the PID of the process which performed the last semop call.

**GETVAL**
Returns the value of a single semaphore within the set.

**GETZCNT**
Returns the number of processes currently waiting for 100% resource utilization.

**SETALL**
Sets all semaphore values with a set to the matching values contained in the array member of the union.

**SETVAL**
Sets the value of an individual semaphore within the set to the val member of the union.

The arg argument represents an instance of type semun. This particular union is declared in linux/sem.h as follows:

```
/* arg for semctl system calls. */
union semun {
        int val;                /* value for SETVAL */
        struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
        ushort *array;          /* array for GETALL & SETALL */
        struct seminfo *__buf;  /* buffer for IPC_INFO */
        void *__pad;
};
```

**val**
Used when the SETVAL command is performed. Specifies the value to set the semaphore to.
**buf**
Used in the IPC_STAT/IPC_SET commands. Represents a copy of the internal semaphore data structure used in the kernel.
**array**
A pointer used in the GETALL/SETALL commands. Should point to an array of integer values to be used in setting or retrieving all semaphore values in a set.

The remaining arguments __buf and __pad are used internally in the semaphore code within the kernel, and are of little or no use to the application developer. As a matter of fact, these two arguments are specific to the Linux operating system, and are not found in other UNIX implementations.

## 3. List the flags, their purpose with code implementation (give Example source code with output)

Since this particular system call is arguably the most difficult to grasp of all the System V IPC calls, we'll examine multiple examples of it in action.
The following snippet returns the value of the passed semaphore. The final argument (the union) is ignored when the GETVAL command is used:

```
int get_sem_val( int sid, int semnum )
{
        return( semctl(sid, semnum, GETVAL, 0));
}
```

To revisit the printer example, let's say the status of all five printers was required:

```
#define MAX_PRINTERS 5

printer_usage()
{
        int x;

        for(x=0; x<MAX_PRINTERS; x++)
                printf("Printer %d: %d\n\r", x, get_sem_val( sid, x ));
}
```

Consider the following function, which could be used to initialize a new semaphore value:

```
void init_semaphore( int sid, int semnum, int initval)
{
        union semun semopts;

        semopts.val = initval;
        semctl( sid, semnum, SETVAL, semopts);
}
```

Note that the final argument of semctl is a copy of the union, rather than a pointer to it. While we're on the subject of the union as an argument, allow me to demonstrate a rather common mistake when using this system call.
Recall from the msgtool project that the IPC_STAT and IPC_SET commands were used to alter permissions on the queue. While these commands are supported in the semaphore implementation, their usage is a bit different, as the internal data structure is retrieved and

copied from a member of the union, rather than as a single entity. Can you locate the bug in this code?

```
/* Required permissions should be passed in as text (ex: "660") */

void changemode(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */
        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
        {
                perror("semctl");
                exit(1);
        }

        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

        /* Change the permissions on the semaphore */
        sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");
}
```

The code is attempting to make a local copy of the internal data structure for the set, modify the permissions, and IPC_SET them back to the kernel. However, the first call to semctl promptly returns EFAULT, or bad address for the last argument (the union!). In addition, if we hadn't checked for errors from that call, we would have gotten a memory fault. Why?

Recall that the IPC_SET/IPC_STAT commands use the buf member of the union, which is a pointer to a type semid_ds. Pointers are pointers are pointers are pointers! The buf member must point to some valid storage location in order for our function to work properly. Consider this revamped version:

```
void changemode(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */

        /* Point to our local copy first! */
        semopts.buf = &mysemds;

        /* Let's try this again! */
        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
        {
                perror("semctl");
                exit(1);
        }

        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

        /* Change the permissions on the semaphore */
        sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");
}
```

**RETURN VALUE**

On failure **semctl**( ) returns -1 with *errno* indicating the error.

Otherwise the system call returns a nonnegative value depending on *cmd* as follows:

| Tag | Description |
|---|---|
| GETNCNT | the value of **semncnt**. |
| GETPID | the value of **sempid**. |
| GETVAL | the value of **semval**. |
| GETZCNT | the value of **semzcnt**. |
| IPC_INFO | the index of the highest used entry in the kernel's internal array recording information about all semaphore sets. (This information can be used with repeated **SEM_STAT** operations to obtain information about all semaphore sets on the system.) |
| SEM_INFO | As for **IPC_INFO**. |

| | |
|---|---|
| SEM_STAT | the identifier of the semaphore set whose index was given in *semid*. |

All other *cmd* values return 0 on success.

**ERRORS**

On failure, *errno* will be set to one of the following:

| Tag | Description |
|---|---|
| EACCES | The argument *cmd* has one of the values **GETALL**, **GETPID**, **GETVAL**, **GETNCNT**, **GETZCNT**, **IPC_STAT**, **SEM_STAT**, **SETALL**, or **SETVAL** and the calling process does not have the required permissions on the semaphore set and does not have the **CAP_IPC_OWNER** capability. |
| EFAULT | The address pointed to by *arg.buf* or *arg.array* isn't accessible. |
| EIDRM | The semaphore set was removed. |
| EINVAL | Invalid value for *cmd* or *semid*. Or: for a **SEM_STAT** operation, the index value specified in *semid* referred to an array slot that is currently unused. |
| EPERM | The argument *cmd* has the value **IPC_SET** or **IPC_RMID** but the effective user ID of the calling process is not the creator (as found in *sem_perm.cuid*) or the owner (as found in *sem_perm.uid*) of the semaphore set, and the process does not have the **CAP_SYS_ADMIN** capability. |
| ERANGE | The argument *cmd* has the value **SETALL** or **SETVAL** and the value to which **semval** is to be set (for some semaphore of the set) is less than 0 or greater than the implementation limit **SEMVMX**. |

**NOTES**

The **IPC_INFO**, **SEM_STAT** and **SEM_INFO** operations are used by the **ipcs**(8) program to provide information on allocated resources. In the future these may modified or moved to a /proc file system interface.

Various fields in a *struct semid_ds* were shorts under Linux 2.2 and have become longs under Linux 2.4. To take advantage of this, a recompilation under glibc-2.1.91 or later should suffice. (The kernel distinguishes old and new calls by an IPC_64 flag in *cmd*.)

In some earlier versions of glibc, the *semun* union was defined in <sys/sem.h>, but POSIX.1-2001 requires that the caller define this union. On versions of glibc where this union is *not* defined, the macro **_SEM_SEMUN_UNDEFINED** is defined in <sys/sem.h>.

The following system limit on semaphore sets affects a **semctl**( ) call:

| Tag | Description |
|---|---|
| SEMVMX | Maximum value for **semval**: implementation dependent (32767). |

For greater portability it is best to always call **semctl**( ) with four arguments.

Under Linux, **semctl**( ) is not a system call, but is implemented via the system call **ipc**(2).

**Code implementation with snapshot of each step and the output**

NOTE: to run the code/to show the implementation/ I used a free cloud supported virtual remote desktop on which virtual box is installed.

Create a new C file in the home folder of the Linux system with the utilization of the "touch" command. We have named this C file as "semctl.c". You can name it as you want. The query is as follows:

```
$ touch semctl.c
```

```
kalsoom@virtualbox:~$ touch semctl.c
```

The GNU nano editor will be used to edit and open the file. So, the stated below query has been used to open the "semctl.c" file.

```
$ nano semctl.c
```

```
kalsoom@virtualbox:~$ nano semctl.c
```

As shown in the image beneath that the file has already been opened in the editor. Write out the same code in your file to see the working of the "semctl( )" system call. The passed values 1 and 4 will be saved to these arguments.

```
 GNU nano 4.8                          semctl.c
#define _XOPEN_SOURCE
#include <sys/sem.h>
#include <unistd.h>
#include <stdio.h>
int main() {
int v = get_val(1, 4);
printf("value is %d\n", v);
}
int get_val (int sid, int semnum)
 {
 return (semctl(sid, semnum, GETVAL, 0));
 }
```

Let's compile the "semctl.c" file in the console. The compilation returns the warning, but there is no need for panic. Execute your file with the "a.out" command. The output shows that the current semaphore value is "-1".

```
$ gcc semctl.c
$ ./a.out
```

```
kalsoom@virtualbox:~$ gcc semctl.c
semctl.c: In function 'main':
semctl.c:6:9: warning: implicit declaration of function 'get_val' [-Wimplicit-f
unction-declaration]
    6 | int v = get_val(1, 4);
      |         ^~~~~~~
kalsoom@virtualbox:~$ ./a.out
value is -1
```

Let's open the file once again to make it different a little bit. Open it using the "nano" editor and define the variable "MAX_COMPS" with some value, e.g. 10. This time update the code with the shown below script. Start from the initialization of the main method. It contains the function call of the method comp_use( ). Now the control has been given to the method comp_use.

The function comp_use contains the "for" loop to use the variable "MAX_COMPS" value as a max value to the "for" loop. Until the loop ends, the print statement will show the computer number. It also gets and prints the current semaphore value from the method "get_val" by calling it here. The get_val method got executed and returned its value to the print statement of function comp_use() to display it on the screen. Save the code file once more and quit it to come back.

```
 GNU nano 4.8                          semctl.c
#define _XOPEN_SOURCE
#include <sys/sem.h>
#include <unistd.h>
#include <stdio.h>
#define MAX_COMPS 10
int main() {
comp_use();
}
int get_val (int sid, int semnum)
 {
 return (semctl(sid, semnum, GETVAL, 0));
 }
comp_use()
{
for (int i=0; i<MAX_COMPS; i++)
  {
printf("Computer %d: %d\n\r", i, get_val(5, i));
}}
```

Now we have to compile the updated code with the “GCC” command for compilation.

```
$ gcc semctl.c
$ ./a.out
```

```
kalsoom@virtualbox:~$ gcc semctl.c
semctl.c: In function 'main':
semctl.c:7:1: warning: implicit declaration of function 'comp_use' [-Wimplicit-
function-declaration]
    7 | comp_use();
      | ^~~~~~~~
semctl.c: At top level:
semctl.c:13:1: warning: return type defaults to 'int' [-Wimplicit-int]
   13 | comp_use()
      | ^~~~~~~~
kalsoom@virtualbox:~$ ./a.out
Computer 0: -1
Computer 1: -1
Computer 2: -1
Computer 3: -1
Computer 4: -1
Computer 5: -1
Computer 6: -1
Computer 7: -1
Computer 8: -1
Computer 9: -1
```