CS531 PROJECT REPORT

Submitted in fulfillment for the award of the grade in

CS531 – FUNDAMENTALS OF SYSTEM PROGRAMMING

in

System V versus POSIX semaphores

by

KRISHNA SINDHURI VEMURI (G01024066)

DEPARTMENT OF COMPUTER SCIENCE

GEORGE MASON UNIVERSITY

FAIRFAX, VA

Abstract

Inter-process communication (IPC) refers to the mechanisms provided by the operating system, which allow different processes to manage shared data object. The UNIX systems provide certain facilities under inter-process communication like the pipes, shared memory, message queues, semaphores etc. In UNIX systems there are two different implementations of the IPC, the System V IPC, and POSIX IPC. The System V IPC is an older version and most widely used. The POSIX IPC was introduced later. In this paper, the difference between the System V and POSIX semaphores is illustrated.

A semaphore can be defined a counter which provides an access to a shared data object for multiple processes. The semaphores are used to synchronize the processes and threads. The semaphores are implemented differently in the System V IPC and POSIX IPC.

Table of Contents

List of Figures

1. Inter-process communication

Inter-process communication refers to the mechanism provided by the operating system, that allows cooperating processes to manage shared data object. The processes try to exchange information while making sure they ensure synchronization.

There are different mechanisms provided by the inter-process communication. Those are Pipes, FIFOs, Signals, Message queues, Shared memory, Semaphores. In this paper, we would be considering the working of the semaphores and the difference between the System V and POSIX semaphores.

2. Semaphores

When two or more processes communicate with each also trying to access a shared resource, it is important to regulate the access to that shared resource. If proper regulation is not ensured it might lead to corruption of data and result in the undesirable circumstances. Thus, semaphores were introduced to enable the proper regulation of the access to the shared data and ensure synchronization of these processes.

Semaphores were invented by the noted Dutch computer scientist, Edsger W. Dijkstra. A semaphore can be defined a counter which provides an access to a shared data object for multiple processes. The semaphores are used to synchronize the processes and threads.

Once a semaphore is created, it can perform two operations ***wait*** and ***signal***. The ***wait*** and ***signal*** operations are said to be ***atomic***. The semaphores control the shared resource and regulate the access of the shared resource by the other processes.

To obtain a shared resource, a process needs to test the semaphore that controls the resource. If the value of the semaphore is greater than 0, the process can use the resource, and the process decrements the value of the semaphore by an integer I, this indicates that I units of the resource have been used by a process. If the value of the semaphore is 0 or less than 0, the process blocks until the semaphore value is greater than 0. When the process finally gets access to the shared resource, it increments the value of the semaphore by I units. So that the processes that are blocked may use the resource now.

In simple words, a thread/process waits for permissions to proceed and then signals that it has proceeded by performing a **wait** operation on the semaphore. The rules of the operation are such that the thread must wait until the semaphore's value is positive, then decrement the value of the semaphore by I. When the thread completes its execution, it performs the **signal** operation and increments the semaphore's value by I.

```
function wait(semaphore S, integer I){
while (S <=0);                  //blocks if S is not positive
S = S – I;                      //decrement the value of the semaphore by I
}
function signal(semaphore S, integer I){
   S = S + I;                   //increment the value of the semaphore by I
}
```

Figure 1: Code fragment depicting the wait and signal operations

The number of semaphores and the value of the I(increment/decrement) operations differentiate the type of the semaphore. There are essentially two types of semaphores, the ***binary semaphores***, and the ***counting semaphores***. The binary semaphores only take values either ***0 or 1***. The binary semaphore can only be incremented/decremented by the value 1. Few scientists debate that the ***binary semaphores*** are ***mutex*** locks. But the mutex is like binary semaphores but has a significant difference, that being the principle of ownership. Ownership is a simple concept that when a task locks only a mutex can unlock it. If a task tries to unlock a mutex it has not locked then an error condition is encountered and most importantly, the mutex is not unlocked. In the binary semaphores there is no concept of ownership and thus it can't be called a **mutex**.

On the other hand, the semaphores that allow an ***arbitrary resource count*** that is, the number of resources processes can have access to, are called ***counting semaphores***. The counting semaphores can be incremented/ decremented by any value.

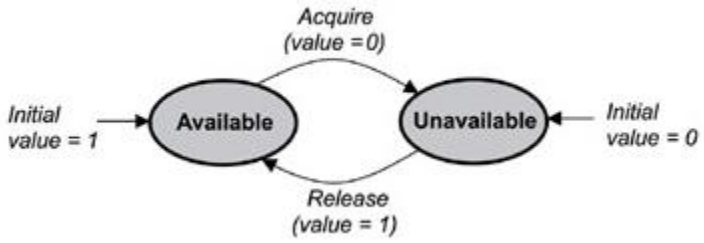

Figure 2: The state diagram of a binary semaphore

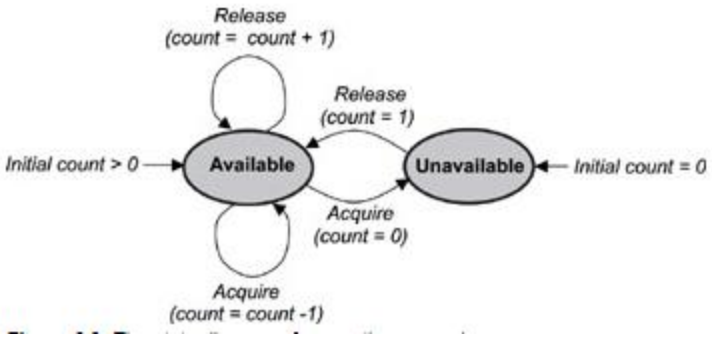

Figure 3: The state diagram of a counting semaphore

This paper discusses the System V and POSIX semaphores and states the difference between them.

3. System V semaphores

The System V IPC supports three types of inter-process communication that are better than pipes and FIFOs (also known as named pipes). They are the message queues, semaphores, and the shared memory.

The message queues, semaphores, and shared memory have the read and write permissions but not the execute operation for the owner, group, and the others (like files). The creating process identifies the default owner like files. The creating process can assign ownership of the facility to another user, it can also revoke an ownership property.

The System V semaphores are completely implemented in the main memory. They require a key value to fetch the proper identifier of the desired IPC resource. The IPC allows creating a semaphore set which consists of a control structure and an array of semaphores. Each semaphore set can contain up to 25 semaphores. The semaphore value can be incremented or decremented by an arbitrary value. The following are the steps involved in ensuring the working of a System V semaphore.

a) Creating an IPC key

We need a System V IPC key to create a System V semaphore. This can be done with the help of ***ftok*** function.

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok (const char *pathname, int proj_id);
```

Figure 4: Code snippet to create a key using ftok function

A System V IPC key of key_t is returned. Here the pathname must be an existing file that can be accessible. The ftok function converts the file name to a unique key value within the system.

b) Initializing a Semaphore Set

The ***semget*** function initializes or gains access to a semaphore. It returns the semaphore id called the ***semid***, when the call succeeds. It returns -1 with an ***errno***, when it fails.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

Figure 5: Code snippet showing the structure of semget function

If the key has the value **IPC_PRIVATE**, a new set of semaphores is created, using the last nine bits of ***semflg*** for permissions. ***nsems*** is a number of semaphores requested. If the last parameter, ***semflg*** specifies **IPC_CREAT** and no semaphore set is associated with the key, a new semaphore set is created. If ***semflg*** specifies **IPC_CREAT | IPC_EXCL**, and a semaphore set exists for the key, ***semget*** fails with ***errno*** set to **EEXIST**.

The ***semget*** function does not initialize the semaphores, we need to use the ***semctl*** function for that.

c) Controlling Semaphores

The ***semctl*** is used for the semaphore control operations as specified by ***cmd.*** This call can be used either for all semaphores in the semaphore set when only the ***semid*** is set or for a

semaphore in the semaphore set when ***semnum*** (starting from 0) along with ***semid*** is set. The fourth argument is optional, or it must be explicitly declared in the application program.

```
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 */
 };
```

Figure 6: Explicit declaration of the fourth argument in the ***semctl*** function call

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

Figure 7: Code snippet showing the structure of ***semctl*** function

When ***semctl*** is successful, it returns 0 otherwise -1 with an ***errno.*** The ***cmd*** argument is one of the following control flags, GETVAL, SETVAL, GETPID, GETNCNT, GETALL, SETALL, IPC_STAT, IPC_SET and IPC_RMID.

The important commands are, **IPC_STAT**, for getting the data from kernel data structures for the semaphore set into the struct ***semid_ds*** pointed by ***buf*** described below, **IPC_SET**, for setting some of the fields in the struct ***semid_ds*** for the semaphore set, **GETALL**, for getting values of all semaphores of the set in the array described below, **SETALL**, for setting initial values of all semaphores in the set, **SETVAL**, for setting an initial value for a semaphore, and **IPC_RMID** for removing the semaphore set.

d) Semaphore Operations

The wait and signal operations on the System V semaphores are performed using the ***semop*** system call.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, size_t nsops);
```

Figure 8: Code snippet showing the structure of ***semop*** function

The ***semop*** function is used to perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by ***semid.*** The ***sops*** argument is a pointer to the array of semaphore-operation structures. The ***nsops*** argument is the number of such structures in the array.

4. POSIX semaphores

POSIX inter-process communication is a variation of System V inter-process communication. Like System V objects, POSIX IPC objects have read and write, but not execute permissions for the owner, the owner's group, and for others. There is no way for the owner of a POSIX IPC object to assign a different owner. Unlike the System V IPC interfaces, the POSIX IPC interfaces are all multithread safe. The POSIX semaphore is a single semaphore that takes values 0 or 1. The POSIX semaphores are of two types the named and the unnamed semaphores.

a) Named Semaphores

The named semaphores can be utilized by the processes that do not share an address space. They provide synchronization between related or unrelated processes and threads. They are also Kernel persistent, they exist in the Kernel until explicitly removed. The named semaphores use the following function calls.

```
sem_open(char* name, int flags, mode_t mode, unsigned int value)
sem_close(sem_t *sem)
sem_unlink(char* name)
```

Figure 9: The system calls used by named semaphores

A named semaphore is identified by a name of the form ***/somename***; that is, a null-terminated string of characters consisting of an initial slash, followed by one or more characters, none of which are slashes.

The ***sem_open*** system call is used to open a named semaphore. It creates the semaphore if it does not already exist. The ***mode*** specifies access permissions. The ***value*** defines the initial value of the semaphore. Two processes can operate on the same named semaphore by passing the same name to ***sem_open***.

The ***sem_close*** closes semaphore for the process that calls it but the semaphore still exists in the kernel. Thus, named semaphores are Kernel persistent.

On using the ***sem_unlink*** system call the name is removed immediately. The semaphore exists until all processes that have it open call ***sem_close.*** Named semaphores persist in the kernel until ***sem_unlink*** is called even if no processes are accessing it.

b) ***Unnamed Semaphores***

The unnamed semaphores are the memory based semaphores. There is no name associated with these semaphores. They are placed in main memory that is shared between processes or threads. The unnamed semaphores are like the Dijkstra's semaphores.

```
sem_init(sem_t *sem, int pshared, unsigned int value)

sem_destroy(sem_t *sem)
```

Figure 10: The system calls used by unnamed semaphores

The ***sem_init*** is used to initialize an unnamed semaphore. The value of ***pshared*** is zero for sharing among threads, non-zero for sharing among processes. If the semaphores are shared between processes, one must explicitly allocate some memory using the ***shm_open*** function call. The ***value*** is the initial value of the semaphore.

The ***sem_destroy*** call destroys the semaphore pointed to by ***sem***. Unlike named semaphores, the unnamed semaphores are not persistent in the kernel.

c) Semaphore Operations

The POSIX semaphores are different from the System V semaphores in many ways. The significant difference is that the System V semaphores are heavy weighted and handle a set of

semaphores whereas POSIX handles only one semaphore and thus said to be light weighted. The semaphore operations in POSIX are as follows.

```
int sem_post(sem_t *sem)
int sem_wait(sem_t *sem)
int sem_trywait(sem_t *sem)
int sem_timedwait(sem_t *sem, struct timespec *timeout)
int sem_getvalue(sem_t *sem, int *val)
```

Figure 11: The semaphore operations of POSIX IPC semaphores

The ***sem_post*** is synonymous with the ***signal*** operation. It increments the semaphore by one. If another process/thread is waiting on the semaphore, one is woken up. This call returns zero on success or -1 on failure. It always recommended checking return value on this system call. As, if ignored, an inconsistency will exist

The ***sem_wait*** is synonymous with ***wait*** operation. It decrements the semaphore by one. If ***sem*** is positive, decrements and immediately else if ***sem*** is zero, blocks until it becomes positive or a signal is received

The ***sem_trywait*** function is used when a thread/process might desire not to block if the semaphore is not positive. This is useful for situations where other work can be done if decrement cannot be immediately performed, call returns an error instead of blocking

The ***sem_timedwait*** call is the thread/process might be willing to wait a while, but not block indefinitely on ***sem_wait***. It is useful for situations where other work might become available later, but right now there is nothing to do. This call blocks on the semaphore for a set amount of time. It uses ***struct timespec*** to specify the absolute time when the timeout will occur. It returns -1 if timeout occurs before semaphore is incremented or sets ***errno*** to ETIMEDOUT

The function ***sem_getval*** is used to determine the current value of the semaphore. It copies the value of ***sem*** into ***val***. It should not be used for synchronization purposes as ***val*** might not reflect the current ***sem*** when the call returns.

5. Conclusion

Several industries still use the System V standard semaphores even when it is heavily weighted and complex, unlike the POSIX standard semaphores which is lightweight. The significant differences between the System V and POSIX semaphores would be that the semaphore count

can be increased or decreased by any value in the System V IPC whereas in POSIX the count can either be incremented or decremented by 1. System V creates an array of semaphores, but POSIX only creates just one. POSIX semaphores do not allow manipulation of semaphore permissions whereas System V semaphores allow you to change the permissions of semaphores to a subset of the original permission. The scalability of the POSIX semaphores using the unnamed semaphores is much higher than the System V semaphores. From the user's perspective, POSIX semaphores are easy to implement than the System V semaphores.

6. Bibliography

1. W. Richard Stevens and Stephen A. Rago, "Advanced Programming in the UNIX environment", Second edition
2. https://www.softprayog.in/programming/semaphore-basis
3. https://docs.oracle.com/cd/E19455-01/806-4750/6jdqdfltg/index.html
4. https://docs.oracle.com/cd/E19455-01/806-4750/6jdqdfltf/index.html
5. https://www.ibm.com/developerworks/library/l-semaphore/index.html
6. http://www.embeddedlinux.org.cn/rtconforembsys/5107final/LiB0037.html
7. https://en.wikipedia.org/wiki/Semaphore_%28programming%29