Implement POSIX Semaphore APIs using System V Semaphores APIs

Expand semaphore support beyond standard POSIX API platforms with System V Semaphore APIs

Suchitra Venugopal  
Shikha Maheshwari  
Adarsh Thampan  

October 16, 2012

Need to use POSIX APIs but the development platform doesn't support them (such as z/OS)? Don't let that hold you back. Learn how you can implement POSIX Semaphore APIs using System V Semaphore APIs to bring your code to more platforms and keep it maintainable.

Overview

A major challenge that you face when you port code to z/OS platform is the unavailability of the POSIX semaphore APIs on z/OS. Typically most programs running on Linux®/UNIX™/Windows® platforms are developed using POSIX semaphores.

One solution for this issue uses the System V APIs for z/OS implementation instead of POSIX APIs. This involves significant code changes spread across the code and a lot of effort to code and test such an implementation. Another better solution implements the POSIX APIs in z/OS using the available System V APIs. In this approach, the code changes for port are minimal. These code changes are excluded from the application code and included as a separate semaphore header and C file. The header and C file are generic ones which you can use on any platform.

This article explains the implementation of POSIX Semaphore APIs using System V Semaphore APIs.

This article will be of immense help to developers who need to use POSIX APIs but the development platform supports only System V APIs (for example z/OS). Also this article significantly distinguishes the POSIX and System V semaphore from a development perspective.

Comparing POSIX Semaphore and System V Semaphore

Semaphore can be of two types: POSIX Semaphore or System V semaphore.
You can operate on semaphores as individual units or as elements in a set. Because System V IPC semaphores can be in a large array, they are extremely heavy weight. A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements. The System V IPC semaphore functions are `semget()`, `semop()` and `semctl()`.

- `semget()` - Create a new semaphore set or access an existing set with the `semget()` system call.
- `semop()` - Perform semaphore operations.
- `semctl()` - Change its ownership or permissions if you are the semaphore creator.

POSIX semaphores are much lighter weight than are System V semaphores. A POSIX semaphore structure defines a single semaphore, not an array of semaphores. The POSIX semaphore functions are:

- `sem_open()` - Connects to, and optionally creates, a named semaphore
- `sem_init()` - Initializes a semaphore structure (internal to the calling program, so not a named semaphore)
- `sem_close()` - Ends the connection to an open semaphore
- `sem_unlink()` - Ends the connection to an open semaphore and removes the semaphore when the last process closes it
- `sem_destroy()` - Initializes a semaphore structure (internal to the calling program, so not a named semaphore)
- `sem_getvalue()` - Copies the value of the semaphore into the specified integer
- `sem_wait()`, `sem_trywait()` - Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process
- `sem_post()` - Increments the count of the semaphore

POSIX comes with simple semantics for creating, initializing, and performing operations on semaphores. They provide an efficient way to handle interprocess communication. System V semaphores are beneficial if you need to implement atomic operations with multiple increments-decrements in a single step. Other than this, stick with POSIX semaphores.

### Limits of Semaphore on z/OS

z/OS supports only System V Semaphores and not POSIX semaphore.

None of the POSIX APIs are available on z/OS. z/OS has only 3 System V APIs for semaphore, those are `semop()`, `semget()` and `semctl()`. If you need to use semaphore APIs in z/OS, use System V APIs alone.

### Table 1. Semaphore support on various platforms

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>AIX®</th>
<th>zLinux</th>
<th>HP</th>
<th>Solaris</th>
<th>Mac OS</th>
<th>z/OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>System V</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
Implementing the POSIX Semaphore APIs using System V Semaphore APIs

All POSIX semaphore functions and types are prototyped or defined in semaphore.h. On certain legacy platforms (say z/OS), only System V semaphores functions are available. Hence, semaphore.h (to define POSIX Semaphores functions) is not available on z/OS. System V semaphore functions are prototyped in sys/sem.h

This section illustrates an approach to implement POSIX semaphore APIs using System V Semaphore APIs. This is an approach implemented and tested well in z/OS and AIX with POSIX semaphore.

Two separate files, Semaphore_zOS.h and Semaphore_zOS.c, implement the System V semaphore APIs in z/OS using the POSIX APIs. The functions we implemented are:

```c
int sem_init(int *semid, int pshared, unsigned int value)
```

Implement this function uses the semget and semctl APIs of System V semaphore (see Listing 1). The error code returned in both semget() function and semctl() function are the same as in POSIX implementation.

**Listing 1. sem_init function implementation**

```c
int sem_init(int *sem, int pshared, unsigned int value)
{
    // get the semaphore first
    semid = semget(key, nsems, IPC_CREAT | IPC_EXCL | 0666);
    if(semid == -1)
        /*assign err code and return*/
    else
        int ret = semctl(*semid, 0, SETVAL, value);  // initialize the semaphore
        if(ret == -1)
            /* assign errcode for semctl and return */
        return ret;
}
```

```c
int sem_destroy(int *semid)
```

Listing 2 shows how to implement sem_destroy using the semctl() function:

**Listing 2. sem_destroy function implementation**

```c
int ret = semctl(*semid, 0, IPC_RMID);
```

To remove the specified semaphore set, use IPC_RMID.

```c
int sem_wait(int *semid)
```

This function places a lock on semaphore. Listing 3 shows how to implement it with semop().

**Listing 3. sem_wait function implementation**

```c
sb.sem_num=0;
sb.sem_op=-1; //Allocate resources
sb.sem_flg=0;
if (semop(*semid, sb, 1) == -1)
```
int sem_trywait(int *semid)

This function also places a lock on semaphore and implemented using semop(). Implementation is similar to sem_wait except for the sem_flg=IPC_NOWAIT. See Listing 4.

Listing 4. sem_trywait function implementation

```c
sb.sem_num=0;
sb.sem_op=-1; //Allocate resources
sb.sem_flg= IPC_NOWAIT;
if (semop(*semid, sb, 1) == -1)
```

This function only locks the semaphore if the semaphore is currently available to be locked. The sem_wait function waits until a lock is got on the semaphore, while sem_trywait will not wait. It just checks whether the lock is available currently, otherwise it just returns.

int sem_post(int *semid)

This function releases the lock on semaphore using semop(). See Listing 5.

Listing 5. sem_post function implementation

```c
sb.sem_num=0;
sb.sem_op=1; // Release resources.
sb.sem_flg=0;
if (semop(*semid, sb, 1) == -1)
```

The other POSIX APIs like sem_open(), sem_close(), sem_unlink(), and sem_getvalue() have not been implemented as it was not required. You also can implement these in the same way!

To set the error codes for these implementations, look at the error codes in POSIX implementation in other platforms.

Test applications

The flow chart in Figures 1, 2, and 3 shows a simple test application that is generated to test the semaphore APIs. (Figures 1, 2, and 3 form a single flow chart but are show in sections for formatting purposes.) The test app is implemented using System V semaphores. The test application creates an array of threads and uses the semaphore POSIX APIs like sem_init(), sem_post(), sem_destroy(), sem_wait(), sem_trywait(), and others.
Figure 1. Flow chart of test application, Part 1

Figure 2. Flow chart of test application, Part 2
Figure 3. Flow chart of test application, Part 3

Case 1 - Test on AIX using POSIX semaphore APIs and the semaphore APIs implemented using System V

This section shows the result for test applications, one using the already available POSIX semaphore APIs and the other using POSIX APIs implemented using System V APIs on an AIX platform. The order of thread which acquires the semaphore varies as it depends on the platform and the priority.

If the results from both the test applications are exactly equal then you can safely conclude that the POSIX API implementation using System V APIs works well.

Listing 6 shows the output of the test application which is implemented using POSIX implementation using System V semaphore APIs:

Listing 6. Test Application output using System V semaphore APIs

bash-3.2# ./test1posiximpln
Thread no 0 id = 258 created
Threadid id =258(ThreadFunction1), locked and incrementing the count
Count (ThreadFunction1) = 1
Thread no 1 id = 515 created
Thread no 2 id = 772 created
Thread no 3 id = 1029 created
Thread id 1029 is in ThreadFunction1, semaphore is already locked and not waiting.
Hence will not increment the counter
Thread no 4 id = 1286 created
Thread no 5 id = 1543 created
Thread no 6 id = 1800 created
Thread id 1800 is in ThreadFunction1, semaphore is already locked and not waiting.
Hence will not increment the counter
Thread no 7 id = 2057 created
Thread no 8 id = 2314 created
Thread no 9 id = 2571 created
Thread id 2571 is in ThreadFunction1, semaphore is already locked and not waiting.
Hence will not increment the counter
Thread 258 going to release lock
Thread id = 515(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 2
Thread id 515 going to release lock
Thread id = 772(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 3
Thread id 772 going to release lock
Thread id = 1286(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 4
Thread id 1286 going to release lock
Thread id = 1543(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 5
Thread id 1543 going to release lock
Thread id = 2057(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 6
Thread id 2057 going to release lock
Thread id = 2314(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 7
All threads joined, Final count = 7
bash-3.2#

Listing 7 shows the output of the test application which is implemented using the already existing POSIX APIs:

Listing 7. Test application output using existing POSIX APIs

bash-3.2# ./test1original
Thread no 0 id = 258 created
Threadid id =258(ThreadFunction1), locked and incrementing the count
Count (ThreadFunction1) = 1
Thread no 1 id = 515 created
Thread no 2 id = 772 created
Thread no 3 id = 1029 created
Thread no 4 id = 1286 created
Thread id 1029 is in ThreadFunction1, semaphore is already locked and not waiting. Hence will not increment the counter
Thread no 5 id = 1543 created
Thread no 6 id = 1800 created
Thread id 1800 is in ThreadFunction1, semaphore is already locked and not waiting. Hence will not increment the counter
Thread no 7 id = 2057 created
Thread no 8 id = 2314 created
Thread no 9 id = 2571 created
Thread id 2571 is in ThreadFunction1, semaphore is already locked and not waiting. Hence will not increment the counter
Thread 258 going to release lock
Thread id = 515(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 2
Thread id 515 going to release lock
Thread id = 772(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 3
Thread id 772 going to release lock
Thread id = 1286(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 4
Thread id 1286 going to release lock
Thread id = 1543(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 5
Thread id 1543 going to release lock
Thread id = 2057(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 6
Thread id 2057 going to release lock
Thread id = 2314(ThreadFunction2) , locked and incrementing the count
Count (ThreadFunction2) = 7
Thread id 2314 going to release lock
All threads joined, Final count = 7
Both of these applications give the same output in AIX.

**Case 2 - Test application on z/OS using the semaphore APIs implemented using System V**

We built and tested the semaphore files implemented using System V semaphores with the same test application in z/OS. Also on AIX, we ran another test application using already available POSIX APIs and compared the outputs.

If the result of the test z/OS and the results of the test on AIX are same then you can safely conclude that the POSIX semaphore implementation using System V semaphore APIs works fine.

**Listing 8** shows the output of the same test application program (as in AIX) in z/OS:

**Listing 8. Test application output using SYTEM V semaphore APIs**

```
bash-3.2#.

Listing 9 shows the output on AIX of test application using already available POSIX APIs:
Listing 9. Test application output using available POSIX APIs

bash-3.2# ./test1original
Thread no 0 id = 258 created
Threadid id =258(ThreadFunction1), locked and incrementing the count
Count (ThreadFunction1) = 1
Thread no 1 id = 515 created
Thread no 2 id = 772 created
Thread no 3 id = 1029 created
Thread no 4 id = 1286 created
Thread id 1029 is in ThreadFunction1, semaphore is already locked and not waiting.
Hence will not increment the counter
Thread no 5 id = 1543 created
Thread no 6 id = 1800 created
Thread id 1800 is in ThreadFunction1, semaphore is already locked and not waiting.
Hence will not increment the counter
Thread no 7 id = 2057 created
Thread no 8 id = 2314 created
Thread no 9 id = 2571 created
Thread id 2571 is in ThreadFunction1, semaphore is already locked and not waiting.
Hence will not increment the counter
Thread 258 going to release lock
Thread id = 515(ThreadFunction2), locked and incrementing the count
Count (ThreadFunction2) = 2
Thread id 515 going to release lock
Thread id = 772(ThreadFunction2), locked and incrementing the count
Count (ThreadFunction2) = 3
Thread id 772 going to release lock
Thread id = 1286(ThreadFunction2), locked and incrementing the count
Count (ThreadFunction2) = 4
Thread id 1286 going to release lock
Thread id = 1543(ThreadFunction2), locked and incrementing the count
Count (ThreadFunction2) = 5
Thread id 1543 going to release lock
Thread id = 2057(ThreadFunction2), locked and incrementing the count
Count (ThreadFunction2) = 6
Thread id 2057 going to release lock
Thread id = 2314(ThreadFunction2), locked and incrementing the count
Count (ThreadFunction2) = 7
Thread id 2314 going to release lock
All threads joined, Final count = 7
bash-3.2#

From the results, you see that the test application shows the same behavior on z/OS and on AIX.

Conclusion

To use the POSIX semaphore APIs in any legacy platform such as z/OS, download the semaphore.h and semaphore.c files and use them to call any of the POSIX APIs. We successfully tested this code fragment and proved it to be correct.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore test applications</td>
<td>semaphore_testapplns.zip</td>
<td>16KB</td>
</tr>
</tbody>
</table>
Related topics

- Find an introduction to Semaphore APIs in Beej's Guide to Unix IPC.
- Read the IPC Semaphores topics for a quick introduction to IPC Semaphores (POSIX and System V).
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.
- Follow developerWorks on Twitter.