

**CSIS0230A Principle of Operating Systems(Class A)**  
**Notes for Tutorial 9**  
**Semaphore**

In the tutorial, you will do some experiment on semaphore. We use semaphore to implement “Reader and Writer problem”.

## 1. Semaphore

In System V (one of the two main commercial Unix flavours, the other being BSD), the following three functions control semaphores. Code using these three functions should include `<sys/types.h>`, `<sys/ipc.h>` and `<sys/sem.h>`. Semaphores are maintained in *sets*. Each time all semaphores within a set are created, modified and destroyed together.

**int** *semget*(*key\_t* key, **int** nsems, **int** semflg);

A set of semaphores is created by invoking the function *semget*(). Like other System V IPC constructs (message passing, shared memory), a *key* (an integer value) may be associated with the set. This allows other processes to access the created set of semaphores, by calling *semget*() with the same key value. It shares the same problem with other System V constructs: it is difficult to make sure that names won't clash. If no such key is needed, one can pass *IPC\_PRIVATE* as the key.

The second argument defines the size of the semaphore set, and the last argument specifies the permission information, which is an octal value of the usual Unix permission (e.g., 0600 specifies user read and write access). Two other flags may be bitwise-or'ed with the permission: *IPC\_CREAT* specifies that the IPC resource should be created if it does not already exist; *IPC\_EXCL* specifies that the function should fail if the semaphore already exists and the *IPC\_CREAT* flag is set.

If everything goes right, the function returns a **positive identifier** for the semaphore. Otherwise, it returns -1. All later semaphore operations require this identifier. The created semaphores have undefined values, and must be initialized using *semctl*().

**int** *semop*(**int** semid, **struct** sembuf \*sops, *size\_t* nsops);

The *semop*() function is used to perform atomically a set of semaphore operations on the semaphores set associated with *semid*. The *sops* argument should be an array of *nsops* semaphore-operation structures (*sembuf*), defined like this:

```
struct sembuf
{
    unsigned short int sem_num; /* semaphore number, 0 = first semaphore */
    short int sem_op;          /* semaphore operation */
    short int sem_flg;        /* operation flag, set to 0 for our purpose */
};
```

Here, *sem\_op* is an integer that defines the operation: if *sem\_op* is 0, the operation will wait until the semaphore value becomes zero. If it is positive, that value is added to the semaphore. If it is negative, the operation will wait until the semaphore value is at least the absolute value of *sem\_op*. Then that value is subtracted from the semaphore value.

**int** *semctl*(**int** semid, **int** semnum, **int** cmd, ...);

The *semctl*() function provides a variety of semaphore control operations as specified by *cmd*. The fourth argument is optional, depending upon the operation requested. If required, it is of type **union** *semun*, which must be explicitly declared by the application program:

```
union semun {  
    int val;  
    struct semid_ds *buf;  
    ushort_t *array;  
} arg;
```

For our purpose, the most important values for *cmd* are *SETVAL*, which set the value of the semaphore *semnum* within the semaphore set to *val* within the union; and *IPC\_RMID*, which removes the semaphore set.

The details of the three functions can be found in the manpage.

## 2. Reader and Writer Problem

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (that is, to read and write) the shared object. We call these processes readers and writers respectively. Two or more readers can access the shared object simultaneously. On the other hand, a writer must have exclusive access to the shared object, or chaos may ensue.

This synchronization problem is referred to as the readers-writers problem. We use semaphore to solve the problem. A semaphore is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait; from the Dutch *proberen*, to test) and V (for signal; from *verhogen*, to increment). One of solutions is the following:

For the structure of a writer process

```
wait(wrt); // wrt is the write semaphore  
...  
// writing is performed  
...  
signal(wrt);
```

For the structure of a reader process

```
wait(mutex); // mutex is the readcount semaphore  
readcount++;  
if (readcount==1)  
    wait(wrt); // wait until all writers are done, and stop new writers  
signal(mutex);  
...  
// reading is performed  
...  
wait(mutex);  
readcount--;  
if (readcount==0)  
    signal(wrt); // No more reader, so write can proceed  
signal(mutex);
```

The details of the function of **P** and **V** and Semaphore is in Chapter 7 from the book “Operating System Concepts”.