

CSIS0230A Principles of Operating Systems(Class A)
Notes for Tutorial 5
Semaphore

When multiple threads shares memory, **race conditions** can occur, i.e., wrong results may occur if the multiple threads access the shared memory in certain order. Race conditions can be solved using semaphores. In the tutorial, you will do some experiment on semaphore: using semaphore to implement “Reader and Writer problem”. In this note, we will explain what is System V semaphore supported by OS, and how to use it.

1. Semaphore

The lecture introduces semaphore as a value that can be decremented (sometimes called wait or p) and incremented (sometimes called signal or v), while maintaining that the semaphore is non-negative all the time by forcing some process to wait.

System V (one of the two main commercial Unix flavours, the other being BSD) provides system calls to support user-level semaphores. To use it, programs should include `<sys/types.h>`, `<sys/ipc.h>` and `<sys/sem.h>`. System V semaphores extends normal semaphores in two important aspects. Firstly, **semaphores are created and destroyed in sets**. Values in the same set can be modified **atomically**. Secondly, semaphore operations can **add and subtract arbitrary numbers**, instead of always 1.

int *semget*(*key_t* key, *int* nsems, *int* semflg);

A set of semaphores is created by invoking the function *semget*(). Like System V IPC shared memory, a semaphore set has two integer names: a programmer-supplied *key* and a system-provided identifier. This function maps *key* to an identifier, possibly creating the semaphore set. If no key is needed, use *IPC_PRIVATE* as the key.

The size of the semaphore set is specified by *nsems*, and *semflg* specifies the permission information (we use 0700 specifying all permission for the user creating the semaphore and no permission for everyone else). 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 exists, in this case *IPC_EXCL* specifies that the function should fail if the semaphore already exists.

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

int *semop*(*int* semid, *struct sembuf* *sops, *size_t* nsops);

This function performs a set of semaphore operations on the semaphores set associated with *semid*. The *sops* argument should be an array of *nsops* semaphore-operation structures of type *struct sembuf*. This structure is defined in 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 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; /* for SETVAL */  
    struct semid_ds *buf; /* for IPC_STAT and IPC_SET */  
    ushort_t *array; /* for GETALL and SETALL */  
} 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.

2. Reader and Writer Problem

Reader and Writer problem occurs in most applications using shared memory. In the general form, a data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes need only to read the shared object, whereas others may also update (that is, to read and write) it. We call these processes readers and writers respectively. Two or more readers can access the shared object simultaneously, while a writer must have exclusive access to the shared object.

The textbook describes a solution using 2 semaphores *wrt* and *mutex*. It is reproduced here for easy reference. Try to understand it completely.

Writer

```
wait(wrt);  
...  
writing is performed  
...  
signal(wrt);
```

Reader

```
wait(mutex);  
++readcount;  
if (readcount==1)  
    wait(wrt);  
signal(mutex);  
...  
reading is performed  
...  
wait(mutex);  
--readcount;  
if (readcount==0)  
    signal(wrt);  
signal(mutex);
```