

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

When multiple threads share 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 semaphores: to use semaphores to implement a solution to the “Reader and Writer problem”. The OS supports a variant of semaphores called “System V semaphores”, which is described by this note.

## 1. Semaphore

The lecture introduces semaphore as a value that can be decremented (down) and incremented (up), while maintaining that the semaphore is non-negative all the time. This is done by forcing a process calling decrement to wait if the value of the semaphore is already 0.

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 extend the concept of 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*(). 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 idea is that if a program can share the *identifier* in some way, like by forking from the same process which created the semaphore set, no *key* is needed. Otherwise, a fixed *key* can be used to address the semaphore set. The latter is not preferable in general, since different programs might accidentally use keys of the same value.

The size of the semaphore set is specified by *nsems*, and this results in semaphores numbered from 0 to *nsems*-1. *semflg* specifies the permission information (we use 0700 specifying that the user creating the semaphore can do anything and anyone else can do nothing). Other flags may be bitwise-or'ed with the permission: *IPC\_CREAT* specifies that the semaphore set should be created if it does not already exist, 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 the 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 semaphore set associated with *semid*. The *sops* argument should be an array of size *nsops*, each element being a semaphore-operation structure of type **struct** *sembuf*. This structure is 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, and leave the value unmodified (i.e., remains 0). 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. Note that regular semaphore operation for increment and decrement corresponds to the *sem\_op* values 1 and -1 respectively.

**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. For our purpose, the most important values for *cmd* are *SETVAL*, which sets the value of the semaphore *semnum* within the semaphore set to a specific integer value (the fourth argument); and *IPC\_RMID*, which removes the semaphore set (no fourth argument). All programs using semaphores should call *semctl()* for these two purposes.

## 2. The Reader and Writer Problem

The 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 only need 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*. The idea is that a semaphore *wrt* makes sure that at most one writer can be accessing the shared object at the same time, and the set of all readers is treated as a writer. The first reader will decrement the *wrt* mutex, and the last one will increment it. The current number of readers is stored in *readcount*, which is a shared variable protected by the semaphore *mutex*. The scheme is reproduced here for easy reference. Try to understand it completely before the tutorial.

Writer

```
down(wrt);
...
writing is performed
...
up(wrt);
```

Reader

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