Semaphores

- Semaphores are best described as counters that can be used to control access to shared resources by multiple processes.
- They are most often used as a **locking mechan**ism to prevent processes from accessing a particular resource while another process is performing operations on it.
- As we shall see, semaphores have a close working relationship with shared memory segments, acting as a **watchdog** to prevent multiple writes to the same memory segment.
- A semaphore **sem** can be seen as an integer variable on which operations such as **wait** and **signal** (not to be confused with the UNIX call ***signal***) can be performed.

p(sem) or wait(sem) :

```
if (sem != 0)
        decrement sem by one
else
        wait until sem becomes non-zero
```

v(sem) or signal(sem):

```
if (queue of waiting processes not empty)
        restart first process in wait queue
else
        increment sem by one
```

- Semaphores, when thought of as resource counters, may be initialized to **any positive** integer value, and are not limited to either being zero or one.
- Semaphores can be used in a variety of ways. Most simply, they can be used to ensure mutual exclusion, where only one process can execute a particular region of code at any one time.
- Consider the following program skeleton:

```
p(sem);

something interesting ....

v(sem);
```

- Let us assume further that the initial value of **sem** is one. From the semaphore invariant, we can see that

(number of completed p operations - number of v operations) <= initial value of semaphore

or

(number of completed p operations - number of v operations) <= 1

- In other words, only one process can execute the group of statements between these particular p and v operations at any one time. Such an area of a program is often called a critical section.

- Just as with message queues, the kernel maintains a special internal data structure for each semaphore set that exists within its addressing space.

- This structure is of type **semid_ds**, and is defined in **linux/sem.h** as follows:

```
/* One semid data structure for each set of semaphores in the system. */
struct semid_ds {
        struct ipc_perm sem_perm;       /* permissions .. see ipc.h */
        time_t          sem_otime;      /* last semop time */
        time_t          sem_ctime;      /* last change time */
        struct sem      *sem_base;      /* ptr to first semaphore in array */
        struct wait_queue *eventn;
        struct wait_queue *eventz;
        struct sem_undo  *undo;         /* undo requests on this array */
        ushort          sem_nsems;      /* no. of semaphores in array */
};
```

- **sem_perm**
    - This is an instance of the **ipc_perm** structure, which is defined for us in linux/ipc.h.
    - It holds the permission information for the semaphore set, including the access permissions, and information about the creator of the set (uid, etc).
- **sem_otime**
    - Time of the last ***semop()*** operation (more on this in a moment)
- **sem_ctime**
    - Time of the last change to this structure (mode change, etc)
- **sem_base**
    - Pointer to the first semaphore in the array (see **sem** structure)
- **sem_undo**
    - Number of **undo** requests in this array
- **sem_nsems**
    - Number of semaphores in the semaphore set (the array)

- In the **semid_ds** structure, there exists a pointer to the base of the semaphore array itself. Each array member is of the **sem** structure type.

- It is also defined in **linux/sem.h**:

```
/* One semaphore structure for each semaphore in the system. */
struct sem {
        short   sempid;         /* pid of last operation */
        ushort  semval;         /* current value */
        ushort  semncnt;        /* num procs awaiting increase in semval */
        ushort  semzcnt;        /* num procs awaiting semval = 0 */
};
```

- **sem_pid**
    - The PID (process ID) that performed the last operation
- **sem_semval**
    - The current value of the semaphore
- **sem_semncnt**
    - Number of processes waiting for resources to become available
- **sem_semzcnt**
    - Number of processes waiting for 100% resource utilization

- The UNIX system V implementation of semaphores is based upon these ideas, although the actual facilities offered are rather more general. The first routines we shall examine are ***semget*** and ***semctl***:

- In order to create a new semaphore set, or access an existing set, the ***semget()*** system call is used.

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

- The function returns a semaphore set IPC identifier on success, or -1 on error:

- **errno** = EACCESS (permission denied)
  EEXIST (set exists, cannot create (IPC_EXCL))
  EIDRM (set is marked for deletion)
  ENOENT (set does not exist, no IPC_CREAT was used)
  ENOMEM (Not enough memory to create new set)
  ENOSPC (Maximum set limit exceeded)

- The first argument to ***semget()*** is the key value (in our case returned by a call to ***ftok()***). This key value is then compared to existing key values that exist within the kernel for other semaphore sets.

- At that point, the open or access operation is dependent upon the contents of the **semflg** argument:

- **IPC_CREAT**
    - Create the semaphore set if it doesn't already exist in the kernel.

- **IPC_EXCL**
    - When used with IPC_CREAT, fail if semaphore set already exists.

- If **IPC_CREAT** is used alone, ***semget()*** either returns the semaphore set identifier for a newly created set, or returns the identifier for a set which exists with the same key value.

- If IPC_EXCL is used along with IPC_CREAT, then either a new set is created, or if the set exists, the call fails with -1. IPC_EXCL is useless by itself, but when combined with IPC_CREAT, it can be used as a facility to guarantee that no existing semaphore set is opened for access.

- As with the other forms of System V IPC, an optional octal mode may be OR'd into the mask to form the permissions on the semaphore set.

- The **nsems** argument specifies the number of semaphores that should be created in a new set.

- The maximum number of semaphores in a set is defined in “linux/sem.h” as:

  #define SEMMSL 32   /* <=512 max num of semaphores per id */

- Note that the nsems argument is ignored if you are explicitly opening an existing set.

- The following is a wrapper function for opening or creating semaphore sets:

```
int open_semaphore_set (key_t keyval, int numsems)
{
        int     sid;

        if ( ! numsems )
                return(-1);

        if((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(sid);
}
```

- The ***semget*** call is analogous to ***msgget***. The extra parameter **nsems** gives the number of semaphores required in the semaphore set; this raises an important point - the UNIX semaphore operations are geared to work with **sets of semaphores**, not single objects.

- The return value from a successful ***semget*** call is a semaphore set identifier, which acts much like a message queue identifier.

- Following normal C usage, an index into a semaphore set can run from **0** to **nsems-1**.

- Associated with each semaphore in the set are the following values defined in **semid_ds**:

    - **semval** - The semaphore value, always a positive integer. This must be set via the semaphore system calls.

    - **sempid** - This is the pid of the process that last acted on the semaphore.

    - **semncnt** - Number of processes that are 'waiting' for the semaphore to reach a value greater than its current value.

    - **semzcnt** - Number of processes that are 'waiting' for the semaphore to reach the value zero.

- The ***semctl*** system call is used to perform control operations on a semaphore set. This call is analogous to the ***msgctl*** system call that is used for operations on message queues:

  int semctl (int semid, int semnum, int cmd, union semun arg);

- The function returns a positive integer on success or -1 on error:

- **errno** = EACCESS (permission denied)
  EFAULT (invalid address pointed to by arg argument)
  EIDRM (semaphore set was removed)
  EINVAL (set doesn't exist, or semid is invalid)
  EPERM (EUID has no privileges for cmd in arg)
  ERANGE (semaphore value out of range)

- From its definition, it is apparent that the ***semctl*** function is considerably more complicated than ***msgctl***.

- With semaphore operations, not only does the IPC key need to be passed, but the target semaphore within the set as well.

- Both system calls utilize a ***cmd*** argument, for specification of the command to be performed on the IPC object. The remaining difference lies in the final argument to both calls.

- In ***msgctl***, the final argument represents a copy of the internal data structure used by the kernel. Recall that we used this structure to retrieve internal information about a message queue, as well as to set or change permissions and ownership of the queue.

- With semaphores, additional operational commands are supported, thus requiring a more complex data type as the final argument.

- The first argument to ***semctl()*** is the key value (in our case returned by a call to ***semget***).
- The second argument (**semun**) is the semaphore number that an operation is targeted towards.
- Essentially this can be thought of as an **index** into the semaphore set, with the first semaphore (or only one) in the set being represented by a value of zero (0).
- The **cmd** argument represents the command to be performed against the set. This can be set to the usual IPC_STAT/IPC_SET commands, along with a set of additional commands specific to semaphore sets:
- **IPC_STAT**
    - Retrieves the **semid_ds** structure for a set, and stores it in the address of the buf argument in the **semun** union.
- **IPC_SET**
    - Sets the value of the **ipc_perm** member of the **semid_ds** structure for a set. Takes the values from the **buf** argument of the **semun** union.
- **IPC_RMID**
    - Removes the set from the kernel.
- **GETALL**
    - Used to obtain the values of all semaphores in a set. The integer values are stored in an array of unsigned short integers pointed to by the *array* member of the union.
- **GETNCNT**
    - Returns the number of processes currently waiting for resources.
- **GETPID**
    - Returns the PID of the process which performed the last *semop* call.
- **GETVAL**
    - Returns the value of a single semaphore within the set.
- **GETZCNT**
    - Returns the number of processes currently waiting for 100% resource utilization.
- **SETALL**
    - Sets all semaphore values with a set to the matching values contained in the *array* member of the union.

- **SETVAL**
    - Sets the value of an individual semaphore within the set to the *val* member of the union.

- The **arg** argument represents an instance of type **semun**. This particular union is declared in **linux/sem.h** as follows:

```
/* arg for semctl system calls. */
union semun {
        int val;                    /* value for SETVAL */
        struct semid_ds *buf;       /* buffer for IPC_STAT & IPC_SET */
        ushort *array;              /* array for GETALL & SETALL */
        struct seminfo *__buf;      /* buffer for IPC_INFO */
        void *__pad;
};
```

- **val**
    - Used when the SETVAL command is performed. Specifies the value to set the semaphore to.

- **buf**
    - Used in the IPC_STAT/IPC_SET commands. Represents a copy of the internal semaphore data structure used in the kernel.

- **array**

  A pointer used in the GETALL/SETALL commands. Should point to an array of integer values to be used in setting or retrieving all semaphore values in a set.

- The remaining arguments **__buf** and **__pad** are Linux specific and are used internally in the semaphore code within the kernel. They of little or no use to the application developer.

- The following code fragment returns the value of the passed semaphore. The final argument (the union) is ignored when the **GETVAL** command is used:

```
int get_sem_val( int sid, int semnum )
{
        return (semctl (sid, semnum, GETVAL, 0));
}
```

- One important use of ***semctl*** is to set the initial values of semaphores, since ***semget*** does not allow a process to do this.

- Consider the following function, which could be used to initialize a new semaphore value:

```
void init_semaphore( int sid, int semnum, int initval)
{
        union semun semopts;

        semopts.val = initval;
        semctl( sid, semnum, SETVAL, semopts);
}
```

- The example given is a function that can be used by a program to either create a single semaphore, or simply obtain the semaphore set identifier associated with it.
- If the semaphore is indeed created, it is assigned an initial value of one with ***semctl***.

Semaphore Operations: the *semop* call

- ***semop*** is the call that actually performs the fundamental semaphore operations.

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

- The function returns 0 on success (all operations performed), or -1 on error:
- errno = E2BIG (nsops greater than max number of ops allowed atomically)
  EACCESS (permission denied)
  EAGAIN (IPC_NOWAIT asserted, operation could not go through)
  EFAULT (invalid address pointed to by sops argument)
  EIDRM (semaphore set was removed)
  EINTR (Signal received while sleeping)
  EINVAL (set doesn't exist, or semid is invalid)
  ENOMEM (SEM_UNDO asserted, not enough memory to create the undo structure necessary)
  ERANGE (semaphore value out of range)

- The first argument to ***semget()*** is the key value (in our case returned by a call to ***semget***).
- The second argument (**sops**) is a pointer to an array of **operations** to be performed on the semaphore set, while the third argument (**nsops**) is the number of operations in that array.
- Again, the stress is on actions on sets of semaphores, the ***semop*** function allowing a group of operations to be performed atomically.
- This means that, if one of the operations can't be done, then none will be done. Unless otherwise specified, the process will then normally block until it can do all the operations at once.

- The **sops** argument points to an array of type **sembuf**. This structure is declared in linux/sem.h as follows:

```
/* semop system call takes an array of these */
struct sembuf {
        ushort  sem_num;        /* semaphore index in array */
        short   sem_op;         /* semaphore operation */
        short   sem_flg;        /* operation flags */
};
```

- **sem_num**
    - The number of the semaphore you wish to deal with
- **sem_op**
    - The operation to perform (positive, negative, or zero)
- **sem_flg**
    - Operational flags
- If **sem_op** is **negative**, then its value is subtracted from the semaphore. This correlates with obtaining resources that the semaphore controls or monitors access of.
- If IPC_NOWAIT is not specified, then the calling process sleeps until the requested amount of resources are available in the semaphore (another process has released some).
- If **sem_op** is **positive**, then it's value is added to the semaphore. This correlates with returning resources back to the application's semaphore set.
- Resources should always be returned to a semaphore set when they are no longer needed!
- If **sem_op** is **zero** (0), then the calling process will ***sleep()*** until the semaphore's value is 0. This correlates to waiting for a semaphore to reach 100% utilization.
- A good example of this would be a daemon running with superuser permissions that could dynamically adjust the size of the semaphore set if it reaches full utilization.
- Consider the following example that loads up a **sembuf** array to perform an operation:

```
struct sembuf sem_lock = { 0, -1, IPC_NOWAIT };
```

- The above initialized structure dictates that a value of “-1” will be added to semaphore number 0 in the semaphore set.
- In other words, one unit of resources will be obtained from the only semaphore in our set (0th member). **IPC_NOWAIT** is specified, so the call will either go through immediately, or fail if another process acquired the semaphore.

- The following is an example of using this initialized **sembuf** structure with the **semop** system call:

  If ((semop(sid, &sem_lock, 1) == -1)
  perror("semop");

- The third argument (**nsops**) says that we are only performing one (1) operation (there is only one **sembuf** structure in our array of operations).

- The sid argument is the IPC identifier for our semaphore set.

- We can then return the resources back to the semaphore set as follows:

  struct sembuf sem_unlock = { 0, 1, IPC_NOWAIT };

- The above simply states that a value of "1" will be added to semaphore number 0 in the semaphore set. In other words, one unit of resources will be returned to the set.

A semaphore example

- The example centers around the two functions **p()** and **v()** which are implementations of the traditional semaphore operations.

- The two functions can then be used to perform mutual exclusion as shown in the program **testsem**.

- **testsem** spawns three child processes which use p() and v() to stop more than one of them executing a critical section at the same time.