Shared Memory & Semaphores

Shared Memory

Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process. This is by far the fastest form of IPC, because there is no intermediation (i.e. a pipe, a message queue, etc). Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process. A segment can be created by one process, and subsequently written to and read from by any number of processes.

Creating and Accessing Shared Memory

In order to create a new shared memory segment, or access an existing one, the shmget() system call is used.

> PROTOTYPE: int shmget (key_t key, int size, int shmflg)
> RETURNS: shared memory segment identifier on success

The first argument to `shmget()` is the key value. A new shared memory segment is created if the key is not associated with an existing segment, and *IPC_CREATE* is asserted in the message flag. If *IPC_CREATE* is used alone, `shmget()` either returns the memory segment identifier for a newly created memory segment, or returns the identifier for a segment which exists with the same key value. If *IPC_EXCL* is used along with *IPC_CREATE*, then either a new segment is created, or if it exists, the call fails with -1. *IPC_EXCL* is useless by itself.

Attaching a Memory Segment

Once a process has a valid IPC identifier for a given segment, the next step is for the process to attach or map the segment into its own addressing space.

> PROTOTYPE: int shmat (int shmid, char *shmaddr, int shmflg);
> RETURNS: address at which segment was attached to the process, or -1 in case of error

If the shmaddr argument is zero (0), the kernel tries to find an unmapped region. This is the recommended method. An address can be specified, but is typically not recommended.
If the SHM_RDONLY flag is OR'd in with the flag argument, then the shared memory segment will be mapped in, but marked as readonly.

This call is perhaps the simplest to use. Consider this wrapper function, which is passed a valid IPC identifier for a segment, and returns the address that the segment was attached to:

```
char *attach_segment( int shmid )
{
      return(shmat(shmid, 0, 0));
}
```

Once a segment has been properly attached, and a process has a pointer to the start of that segment, reading and writing to the segment become as easy as simply referencing or dereferencing the pointer! Be careful not to lose the value of the original pointer! If this happens, you will have no way of accessing the base (start) of the segment.

Controlling a shared memory segment

To perform control operations on a shared memory segment, you use the *shmctl()* system call.

SYSTEM CALL: shmctl();
PROTOTYPE: PROTOTYPE: int shmctl (int shmqid, int cmd, struct shmid_ds *buf);
RETURNS: 0 on success

Possible values of *cmd* are:
IPC_STAT : Retrieves the shared memory segment data structure, and stores it in the address pointed to by buf .
*IPC_SET***:** Write the values of the shmid_ds structure pointed to by buff to the message queue data structure.
*IPC_RMID***:** Removes the shared memory segment from the kernel.

Detaching a Shared Memory Segment
After a shared memory segment is no longer needed by a process, it should be detached by calling the *shmdt* system call. This is not the same as removing the segment from the kernel

Semaphores

Semaphores can best be described as counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it. Semaphores are often dubbed the most difficult to grasp of the three types of System V IPC objects. The name *semaphore* is actually an old railroad term,

referring to the crossroad ``arms'' that prevent cars from crossing the tracks at intersections. The same can be said about a simple semaphore set. If the semaphore is *on* (the arms are up), then a resource is available (cars may cross the tracks). However, if the semaphore is *off* (the arms are down), then resources are not available (the cars must wait).

Creating and Accessing Semaphore Sets

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

PROTOTYPE: int semget (key_t key, int nsems, int semflg);
RETURNS: semaphore set IPC identifier on success

The first argument to *semget()* is the key value. 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. 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. The *nsems* argument specifies the number of semaphores that should be created in a new set.

Semaphores Operations

All semaphores operations are performed using the *semop*() function

PROTOTYPE: int semop (int semid, struct sembuf *sops, unsigned nsops);
RETURNS: 0 on success (all operations performed)

The first argument to *semop*() is the key value. 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.
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 */
    };
```

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 its 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! Finally, if sem_op is zero, then the calling process will sleep() until the semaphore's value is 0. This correlates to waiting for a semaphore to reach 100% utilization.

Controlling a semaphore set

To perform control operations on a semaphore set, you use the *semctl()* system call.

> PROTOTYPE: PROTOTYPE: int semctl (int semid, int semnum, int cmd, union semun arg);
> RETURNS: positive integer on success

The *semctl()* works in a fashion similar to other Sys V IPC control functions. The familiar *IPC_STAT/IPC_SET* commands are present, along with a wealth of additional commands specific to semaphore sets.