Shared Memory

- The shared memory operations allow two or more processes to **share** a segment of **physical memory**.
- 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.
- They provide the fastest and most efficient of all IPC mechanisms since there is no intermediation. All shared memory operations require special **hardware support**. If that is not present, then the shared memory operations will not be available.
- Shared memory should only be used when efficiency is important, since you will be limiting the range systems on which the application will run.

Internal and User Data Structures

- As with message queues and semaphore sets, the kernel maintains a special internal data structure for each shared memory segment which exists within its addressing space. This structure is of type **shmid_ds**, and is defined in **linux/shm.h** as follows:

```
/* One shmid data structure for each shared memory segment in the system. */
struct shmid_ds {
        struct ipc_perm shm_perm;       /* operation perms */
        int     shm_segsz;              /* size of segment (bytes) */
        time_t  shm_atime;              /* last attach time */
        time_t  shm_dtime;              /* last detach time */
        time_t  shm_ctime;              /* last change time */
        unsigned short  shm_cpid;       /* pid of creator */
        unsigned short  shm_lpid;       /* pid of last operator */
        short   shm_nattch;             /* no. of current attaches */

                                /* the following are private */

        unsigned short   shm_npages;    /* size of segment (pages) */
        unsigned long   *shm_pages;     /* array of ptrs to frames -> SHMMAX */
        struct vm_area_struct *attaches; /* descriptors for attaches */
};
```

- Operations on this structure are performed by a special system call, and should not be performed . The following is are descriptions of the more pertinent fields:

- **shm_perm**
    - This is an instance of the `ipc_perm` structure, which is defined for us in `linux/ipc.h`. This holds the permission information for the segment, including the access permissions, and information about the creator of the segment (uid, etc).

- **shm_segsz**
    - Size of the segment (measured in bytes).

- **shm_atime**
    - Time the last process attached the segment.

- **shm_dtime**
    - Time the last process detached the segment.

- **shm_ctime**
    - Time of the last change to this structure (mode change, etc).

- **shm_cpid**
    - The PID of the creating process.

- **shm_lpid**
    - The PID of the last process to operate on the segment.

- **shm_nattch**
    - Number of processes currently attached to the segment.

The *shmget* and *shmop* system calls

- In order to create a new message queue, or access an existing queue, the ***shmget()*** system call is used.

  int shmget (key_t key, int size, int shmflg);

- The function returns a shared memory segment identifier on success or -1 on error:

- **errno** = EINVAL (Invalid segment size specified)
  EEXIST (Segment exists, cannot create)
  EIDRM (Segment is marked for deletion, or was removed)
  ENOENT (Segment does not exist)
  EACCES (Permission denied)
  ENOMEM (Not enough memory to create segment)

- This call closely corresponds to ***msgget*** and ***semget***. The first argument to ***shmget()*** 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 shared memory segments.

- At that point, the open or access operation is dependent upon the contents of the shmflg argument.

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

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

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

- If IPC_EXCL is used along with IPC_CREAT, then either a new segment is created, or if the segment 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 segment is opened for access.

- The parameter size gives the required minimum size (in bytes) of the memory segment.

- Once again, an optional octal mode may be OR'd into the mask.

- The following is a wrapper function for locating or creating a shared memory segment :

```
int open_segment( key_t keyval, int segsize )
{
        int     shmid;

        if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(shmid);
}
```

- The memory segment created is part of physical memory, and not the process's logical data space.

- To use it, the process (and any cooperating process) must explicitly attach the memory segment to its logical data space using the ***shmat*** call, which is part of the ***shmop*** group.

```
int shmat ( int shmid, char *shmaddr, int shmflg);
```

- The function returns an address at which segment was attached to the process, or -1 on error:

- **errno** = EINVAL (Invalid IPC ID value or attach address passed)
  ENOMEM (Not enough memory to attach segment)
  EACCES (Permission denied)

- ***shmat*** associates the memory segment identified by **shm_id** (which will have come from a ***shmget*** call) with a valid address for the calling process. This is the returned value.

- **shmaddr** gives the programmer some control over the address selected by the call. If it is zero, i.e. (char *)0, the segment is attached at the first available address chosen by the system recommended!).

- If **shmaddr** is non-zero, the segment will be attached at, or near, the address held within it, the exact action depending on flags held in the **shmflags** argument.

- **shmflag** is constructed from the two flags SHM_RDONLY and SHM_RND which are defined in <linux/shm.h>. SHM_RDONLY requests that the segment is attached for reading only.

- SHM_RND affects the way ***shmat*** treats a non-zero value for **shmaddr**. If it is set, the call will round **shmaddr** to a page boundary in memory. If not ***shmat*** will use the exact value of **shmaddr**.

- The following is a 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));
}
```

- There is one other ***shmop*** call, named ***shmdt***. It is the inverse of ***shmat*** and **detaches** a shared memory segment from the process's logical address space. It is called, straightforwardly,

```
char *memptr;
…….
retval = shmdt (memptr);
```

- **retval** is an integer and is 0 on success, -1 on error.

The *shmctl* system call

- This particular call is modeled directly after the ***msgctl*** call for message queues.

```
int shmctl ( int shmqid, int cmd, struct shmid_ds *buf );
```

- The function returns 0 on success and -1 on error:

- **errno** = EACCES (No read permission and cmd is IPC_STAT)
  EFAULT (Address pointed to by buf is invalid with IPC_SET and IPC_STAT commands)
  EIDRM (Segment was removed during retrieval)
  EINVAL (shmqid invalid)
  EPERM (IPC_SET or IPC_RMID command was issued, but calling process does not have write (alter) access to the segment)

- This exactly parallels ***msgctl***, and command can take the values IPC_STAT, IPC_SET and IPC_RMID.

- **IPC_STAT**
  - Retrieves the **shmid_ds** structure for a segment, and stores it in the address of the **buf** argument

- **IPC_SET**
  - Sets the value of the **ipc_perm** member of the **shmid_ds** structure for a segment. Takes the values from the **buf** argument.

- **IPC_RMID**
    - Marks a segment for removal.

- The **IPC_RMID** command doesn't actually remove a segment from the kernel. Rather, it marks the segment for removal.
- The actual removal itself occurs when the last process currently attached to the segment has properly detached it. Of course, if no processes are currently attached to the segment, the removal seems immediate.

A shared memory example:

- We will now construct a simple program ***shmcopy*** to demonstrate a practical use of shared memory.
- ***shmcopy*** just copies its standard input to its standard output, but gets around the UNIX property that all read and write calls block until they complete.
- Each invocation of ***shmcopy*** results in two processes, a reader and a writer, that share two buffers implemented as shared memory segments.
- To coordinate the two processes, and so to prevent the writer from writing to a buffer before the reader has filled it, we will use two binary semaphores.
- BUFSIZE is defined in stdio.h and gives the system's disk blocking factor.
- The template databuf shows the structure we will impose on each shared memory segment. In particular, the member d_nread will enable the reader process to pass the number of characters read to the writer via the memory segments.
- The next file contains routines for initializing the two shared memory segments and semaphore set.
- It also contains the routine remove that deletes the various IPC objects at the end of program execution.
- Note in particular the way **shmat** is called to attach memory segments to the process's address space.
- The next file shows the main function for shmcopy. It simply calls initialization routines, then creating the reading (parent) and writing (child) processes.
- Notice that it is the writing process that calls remove when the program finishes.
- main created the IPC objects before the fork. The addresses that identify the shared memory segments (held in buf1 and buf2) will be meaningful in both processes.
- The first routine reader, takes its input from standard input, i.e. file descriptor 0. It is passed the semaphore set identifier in semid, and the addresses of the two shared memory segments in buf1 and buf2.

- The sembuf structures here just define p() and v() operations for a two semaphore set. This time they are used to synchronize the reader and writer.

- reader uses v1 to signal that a read has been completed, and waits, by calling semop with p2, for the writer to signal that a write has been completed.

- The final routine called by shmcopy is writer. Again the semaphore set is used to coordinate reader and writer.

- This time writer uses v2 to signal and waits on p1. It is also important to note that the values for buf1->d_nread and buf2->d_nread are set by the reading process.