Introduction to Shared Memory

- **Shared Memory** allows two or more processes to share a memory region or segment of memory for reading and writing purposes
- The problem with pipes, fifo and message queue is that mode switches are involved as the data has to pass from one process buffer to the kernel buffer and then to another process buffer
- Since access to user-space memory does not require a mode switch, therefore, shared memory is considered as one of the quickest means of IPC

APIs to Shared Memory

Interface	System-V API	POSIX API
Header file	`<sys/shm.h>`	`<mqueue.h>`
Data Structure	`shmid_ds`	File descriptor
Create/open	`shmget(), shmat()`	`shm_open()`
Close	`shmdet()`	`shm_unlink()`
Perform IPC	Access memory	`mmap(),memcpy()`
Control operations	`shmctl()`	

Creating/Opening Shared Memory Segment

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

- The **`shmget()`** system call creates a new shared memory segment or obtains the identifier of an existing segment. The contents of a newly created shared memory segment are initialized to 0. The return value is the ID of the shared memory segment
- The first argument `key` can be the constant `IPC_PRIVATE` or can be achieved using `ftok()` library call (as discussed in MQ session)
- The second argument `size` specifies the desired size of the segment in bytes. Kernel round it up to next multiple of the system page size. If we are using `shmget()` to obtain the identifier of an existing segment, then size has no effect on the segment
- The `shmflg` argument specifies the permissions to be placed on a new shared memory segment or checked against an existing segment. In addition, it can be a bit wise OR of constants like `IPC_CREAT` and `IPC_EXCL`

Using Shared Memory Segment

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- The **shmat()** system call attaches the shared memory segment identified by `shmid` to the address space of the calling process
- The second argument `shmaddr` is the address where the memory segment will be attached. If we want the OS Kernel to select a suitable address, we keep it `NULL`
- The third argument `shmflg` can be `SHM_RDONLY` to attach the shared memory segment for read-only access. We can place a zero over there for giving both read and write access
- On success `shmat()` returns the address at which the shared memory segment is attached, which can be treated like a normal C pointer. We can assign the return value from `shmat()` to a pointer of some intrinsic data type or a programmer defined structure

Using Shared Memory Segment

```
int shmdt(const void *shmaddr);
```

- When a process no longer needs to access a shared memory segment, it can call `shmdt()` to detach the segment from its address space
- The only argument to the call `shmaddr` identifies the segment to be detached. It should be a value returned by a previous call to `shmat()`
- Detaching a shared memory segment is not the same as deleting it. Deletion can be performed using the `shmctl()`
- A child created by `fork()` inherits its parent's attached shared memory segments. Thus, shared memory provides an easy method of IPC between parent and child. However, after an `exec()`, all attached shared memory segments are detached
- Shared memory segments are also automatically detached on process termination

Using Shared Memory Segment

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

- The `shmctl()` system call is used to perform control operations on the shared memory segment specified in its first argument `shmid`
- One of the basic control operation is deletion of the shared memory segment. This can be done by giving `IPC_RMID` as the `cmd` in the second argument. This will destroy the memory segment after the last process detaches it
- For deletion operation of shared memory the third argument is kept `NULL`