**Name:Rohan Nivas Chavan**

**PRN:2019BTEIT00084**

**UOS LAB**

##### Write 2 programs that will communicate via shared memory and semaphores. Data will be exchanged via memory and semaphores will be used to synchronize and notify each process when operations such as memory loaded and memory read have been performed.

**Objectives**:

1. To learn about IPC through message queue.
2. Use of system call and IPC mechanism to write effective application programs.

#### Theory:

*Shared Memory* is an efficeint means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.

Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

The server maps a shared memory in its address space and also gets access to a synchronization mechanism. The server obtains exclusive access to the memory using the synchronization mechanism and copies the file to memory. The client maps the shared memory in its address space. Waits until the server releases the exclusive access and uses the data.

To use shared memory, we have to perform 2 basic steps:

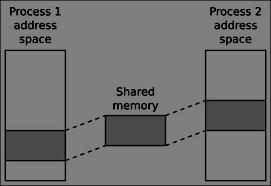
* + Request to the operating system a memory segment that can be shared between processes. The user can create/destroy/open this memory using a **shared memory object**: *An object that represents memory that can be mapped concurrently into the address space of more than one process.*.
  + Associate a part of that memory or the whole memory with the address space of the calling process. The operating system looks for a big enough memory address range in the calling process' address space and marks that address range as an special range.

A process creates a shared memory segment using shmget()

The original owner of a shared memory segment can assign ownership to another user with shmctl().

A shared segment can be attached to a process address space using shmat(). It can be detached using shmdt().

Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory.



#### Data Dictionary:

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Variable/function | Data Type | Use |
| 1 | operations | Struct sembuf | Used to Store Operations |
| 2 | shm\_address | void | Shared memory address |
| 3 | semid | int | Used to store semaphore id |
| 4 | shmid | int | Used to store Shared memory id |
| 5 | semkey | key\_t | Store key of semaphores |
| 6 | shmkey | key\_t | Store key of SHM |

**Program:**

//client side program #include <stdio.h> #include <string.h> #include <sys/ipc.h> #include <sys/sem.h> #include <sys/shm.h>

#define SEMKEYPATH "/dev/null" #define SEMKEYID 1

#define SHMKEYPATH "/dev/null" #define SHMKEYID 1

#define NUMSEMS 2

#define SIZEOFSHMSEG 50

int main(int argc, char \*argv[])

{

struct sembuf operations[2]; void \*shm\_address;

int semid, shmid, rc; key\_t semkey, shmkey;

semkey = ftok(SEMKEYPATH,SEMKEYID); if ( semkey == (key\_t)-1 )

{

printf("main: ftok() for sem failed\n"); return -1;

}

shmkey = ftok(SHMKEYPATH,SHMKEYID); if ( shmkey == (key\_t)-1 )

{

printf("main: ftok() for shm failed\n"); return -1;

}

semid = semget( semkey, NUMSEMS, 0666); if ( semid == -1 )

{

printf("main: semget() failed\n"); return -1;

}

shmid = shmget(shmkey, SIZEOFSHMSEG, 0666); if (shmid == -1)

{

printf("main: shmget() failed\n"); return -1;

}

shm\_address = shmat(shmid, NULL, 0); if ( shm\_address==NULL )

{

printf("main: shmat() failed\n"); return -1;

}

operations[0].sem\_num = 0;

operations[0].sem\_op = 0;

operations[0].sem\_flg = 0;

operations[1].sem\_num = 0;

operations[1].sem\_op = 1;

operations[1].sem\_flg = 0;

rc = semop( semid, operations, 2 ); if (rc == -1)

{

printf("main: semop() failed\n"); return -1;

}

strcpy((char \*) shm\_address, "Hello from Client"); operations[0].sem\_num = 0;

operations[0].sem\_op = -1;

operations[0].sem\_flg = 0;

operations[1].sem\_num = 1;

operations[1].sem\_op = 1;

operations[1].sem\_flg = 0;

rc = semop( semid, operations, 2 ); if (rc == -1)

{

printf("main: semop() failed\n"); return -1;

}

rc = shmdt(shm\_address); if (rc==-1)

{

printf("main: shmdt() failed\n"); return -1;

}

if(operations[0].sem\_op == -1)

printf("\nread performed by server"); return 0;

}

//server side program #include <stdio.h> #include <string.h> #include <sys/ipc.h> #include <sys/sem.h> #include <sys/shm.h>

#define SEMKEYPATH "/dev/null" #define SEMKEYID 1

#define SHMKEYPATH "/dev/null" #define SHMKEYID 1

#define NUMSEMS 2

#define SIZEOFSHMSEG 50

#define NUMMSG 2

int main(int argc, char \*argv[])

{

int rc, semid, shmid, i; key\_t semkey, shmkey; void \*shm\_address;

struct sembuf operations[2]; struct shmid\_ds shmid\_struct; short sarray[NUMSEMS];

semkey = ftok(SEMKEYPATH,SEMKEYID); if ( semkey == (key\_t)-1 )

{

printf("main: ftok() for sem failed\n");

return -1;

}

shmkey = ftok(SHMKEYPATH,SHMKEYID); if ( shmkey == (key\_t)-1 )

{

printf("main: ftok() for shm failed\n"); return -1;

}

semid = semget( semkey, NUMSEMS, 0666 | IPC\_CREAT | IPC\_EXCL ); if ( semid == -1 )

{

printf("main: semget() failed\n"); return -1;

}

sarray[0] = 0;

sarray[1] = 0;

rc = semctl( semid, 1, SETALL, sarray); if(rc == -1)

{

printf("main: semctl() initialization failed\n"); return -1;

}

shmid = shmget(shmkey, SIZEOFSHMSEG, 0666 | IPC\_CREAT | IPC\_EXCL); if (shmid == -1)

{

printf("main: shmget() failed\n"); return -1;

}

shm\_address = shmat(shmid, NULL, 0); if ( shm\_address==NULL )

{

printf("main: shmat() failed\n"); return -1;

}

printf("Ready for client jobs\n"); for (i=0; i < NUMMSG; i++)

{

operations[0].sem\_num = 1;

operations[0].sem\_op = -1;

operations[0].sem\_flg = 0;

operations[1].sem\_num = 0;

operations[1].sem\_op = 1; operations[1].sem\_flg = IPC\_NOWAIT; rc = semop( semid, operations, 2 );

if (rc == -1)

{

printf("main: semop() failed\n"); return -1;

}

printf("Server Received : \"%s\"\n", (char \*) shm\_address); operations[0].sem\_num = 0;

operations[0].sem\_op = -1; operations[0].sem\_flg = IPC\_NOWAIT; rc = semop( semid, operations, 1 );

if (rc == -1)

{

printf("main: semop() failed\n"); return -1;

}

}

rc = semctl( semid, 1, IPC\_RMID ); if (rc==-1)

{

printf("main: semctl() remove id failed\n"); return -1;

}

rc = shmdt(shm\_address); if (rc==-1)

{

printf("main: shmdt() failed\n"); return -1;

}

rc = shmctl(shmid, IPC\_RMID, &shmid\_struct); if (rc==-1)

{

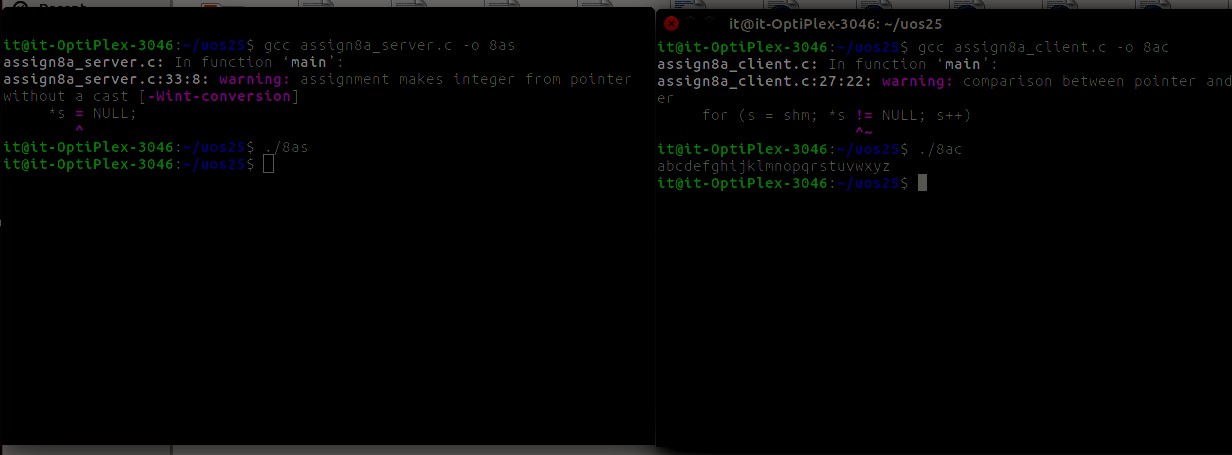
printf("main: shmctl() failed\n"); return -1;

}

return 0;

}

#### Output:



**Conclusion:**

Communication using Shared Memory IPC between client and server established and implemented using shm functions.

#### References:

Dave’s Programming in C Tutorials