

ASSIGNMENT No: 7-B

TITLE: Inter-process Communication using Shared Memory using System V.

AIM: Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and write the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

THEORY:

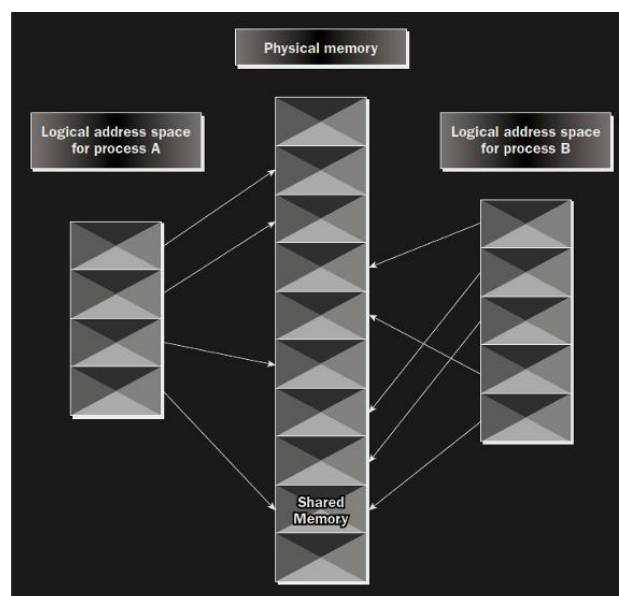
Shared Memory

Shared memory allows two unrelated processes to access the same logical memory. Shared memory is a very efficient way of transferring data between two running processes. Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process.

Other processes can then “attach” the same shared memory segment into their own address space. All processes can access the memory locations just as if the memory had been allocated by malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.

Shared memory provides an efficient way of sharing and passing data between multiple processes. By itself, shared memory doesn’t provide any synchronization facilities. Because it provides no synchronization facilities, we usually need to use some other mechanism to synchronize access to the shared memory.

Typically, we use shared memory to provide efficient access to large areas of memory and pass small messages to synchronize access to that memory. There are no automatic facilities to prevent a second process from starting to read the shared memory before the first process has finished writing to it. It’s the responsibility of the programmer to synchronize access. Figure below shows an illustration of how shared memory works.



The arrows show the mapping of the logical address space of each process to the physical memory available. In practice, the situation is more complex because the available memory actually consists of a mix of physical memory and memory pages that have been swapped out to disk. The functions for shared memory resemble those for semaphores:

```
#include <sys/shm.h>
```

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

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

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

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

As with semaphores, the include files `sys/types.h` and `sys/ipc.h` are normally automatically included by `shm.h`.

`shmget()` It is used to create shared memory

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

As with semaphores, the program provides `key`, which effectively names the shared memory segment, and the `shmget` function returns a shared memory identifier that is used in subsequent shared memory functions. There's a special key value, `IPC_PRIVATE`, that creates shared memory private to the process. The second parameter, `size`, specifies the amount of memory required in bytes. The third parameter, `shmflg`, consists of nine permission flags that are used in the same way as the mode flags for creating files. A special bit defined by `IPC_CREAT` must be bitwise ORed with the permissions to create a new shared memory segment. It's not an error to have the `IPC_CREAT` flag set and pass the key of an existing shared memory segment. The `IPC_CREAT` flag is silently ignored if it is not required. If the shared memory is successfully created, `shmget` returns a nonnegative integer, the shared memory identifier. On failure, it returns `-1`.

shmat()

When we first create a shared memory segment, it's not accessible by any process. To enable access to the shared memory, we must attach it to the address space of a process. We do this with the `shmat` function:

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

The first parameter, `shm_id`, is the shared memory identifier returned from `shmget`. The second parameter, `shm_addr`, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears. The third parameter, `shmflg`, is a set of bitwise flags. The two possible values are `SHM_R` for reading and `SHM_W` for write access. If the `shmat` call is successful, it returns a pointer to the first byte of shared memory. On failure `-1` is returned.

shmctl()

it is used for controlling functions for shared memory.

```
int shmctl(int shm_id, int command, struct shmid_ds *buf);
```

The shmid_ds structure has at least the following members:

```
struct shmid_ds {uid_t shm_perm.uid;uid_t shm_perm.gid;  
mode_t shm_perm.mode;}
```

The first parameter, shm_id, is the identifier returned from shmget. The second parameter,command, is the action to take. It can take three values, shown in the following table.

Command Description

IPC_STAT : Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.

IPC_SET: Sets the values associated with the shared memory to those provided in the shmid_ds data structure, if the process has permission to do so.

IPC_RMID: Deletes the shared memory segment.

The third parameter, buf, is a pointer to the structure containing the modes and permissions for the shared memory.

PROGRAM – Server Process

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#define SHM_KEY 1234
#define SHM_SIZE 1024
int main()
{
    // Create a shared memory segment.
    int shmid = shmget(SHM_KEY, SHM_SIZE, 0666 | IPC_CREAT);

    if (shmid == -1)
    {
        perror("shmget");
        exit(1);
    }

    // Attach to the shared memory segment.
    char *shm = shmat(shmid, NULL, 0);

    if (shm == (char *)-1)
    {
        perror("shmat");
        exit(1);
    }

    // Write a message to the shared memory segment.
    strcpy(shm, "Hello from the server!");

    // Detach from the shared memory segment.
    shmdt(shm);

    return 0;
}
```

PROGRAM – Client Process

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#define SHM_KEY 1234
#define SHM_SIZE 1024
int main()
{
    // Create or access the shared memory segment.
    int shmid = shmget(SHM_KEY, SHM_SIZE, 0666);

    if (shmid == -1)
    {
        perror("shmget");
        exit(1);
    }

    // Attach to the shared memory segment.
    char *shm = shmat(shmid, NULL, 0);

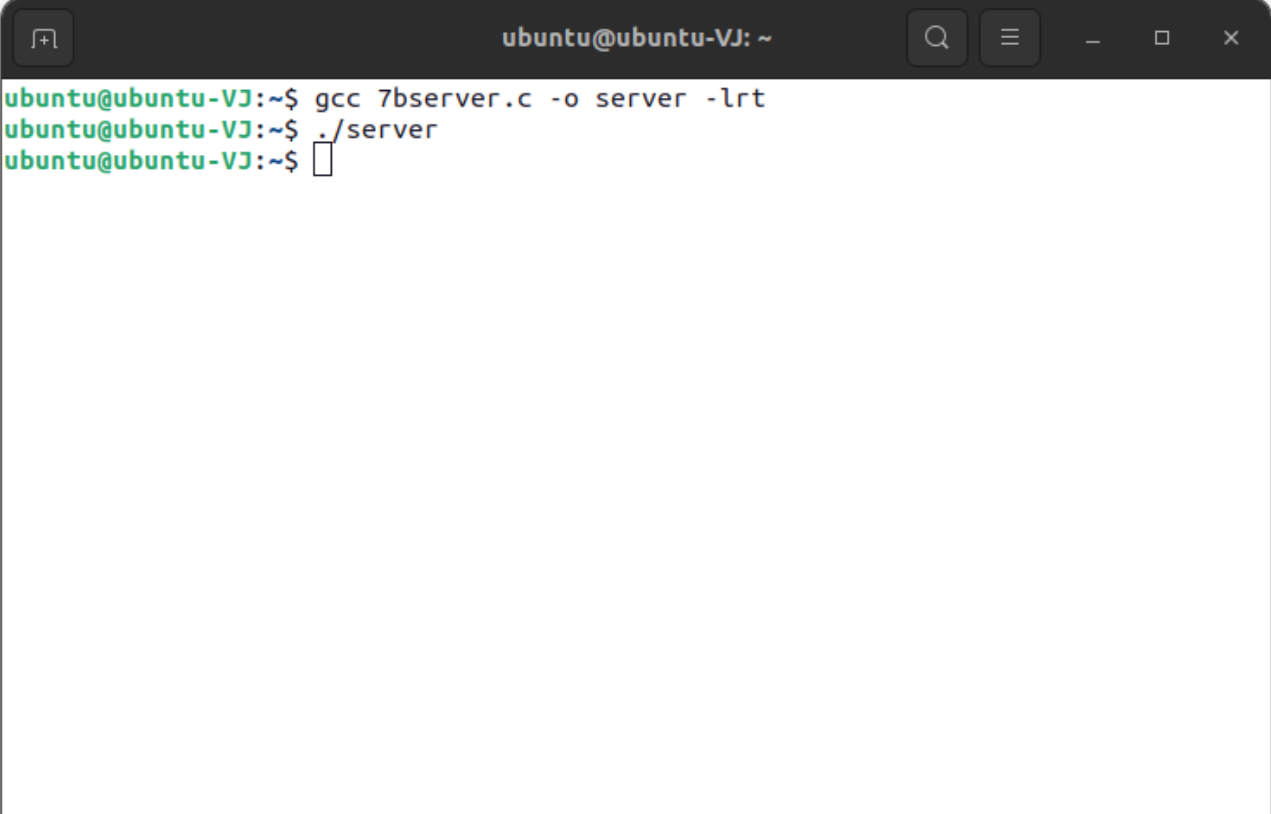
    if (shm == (char *)-1)
    {
        perror("shmat");
        exit(1);
    }

    // Read and display the message from the shared memory segment.
    printf("Message from the server: %s\n", shm);

    // Detach from the shared memory segment.
    shmdt(shm);

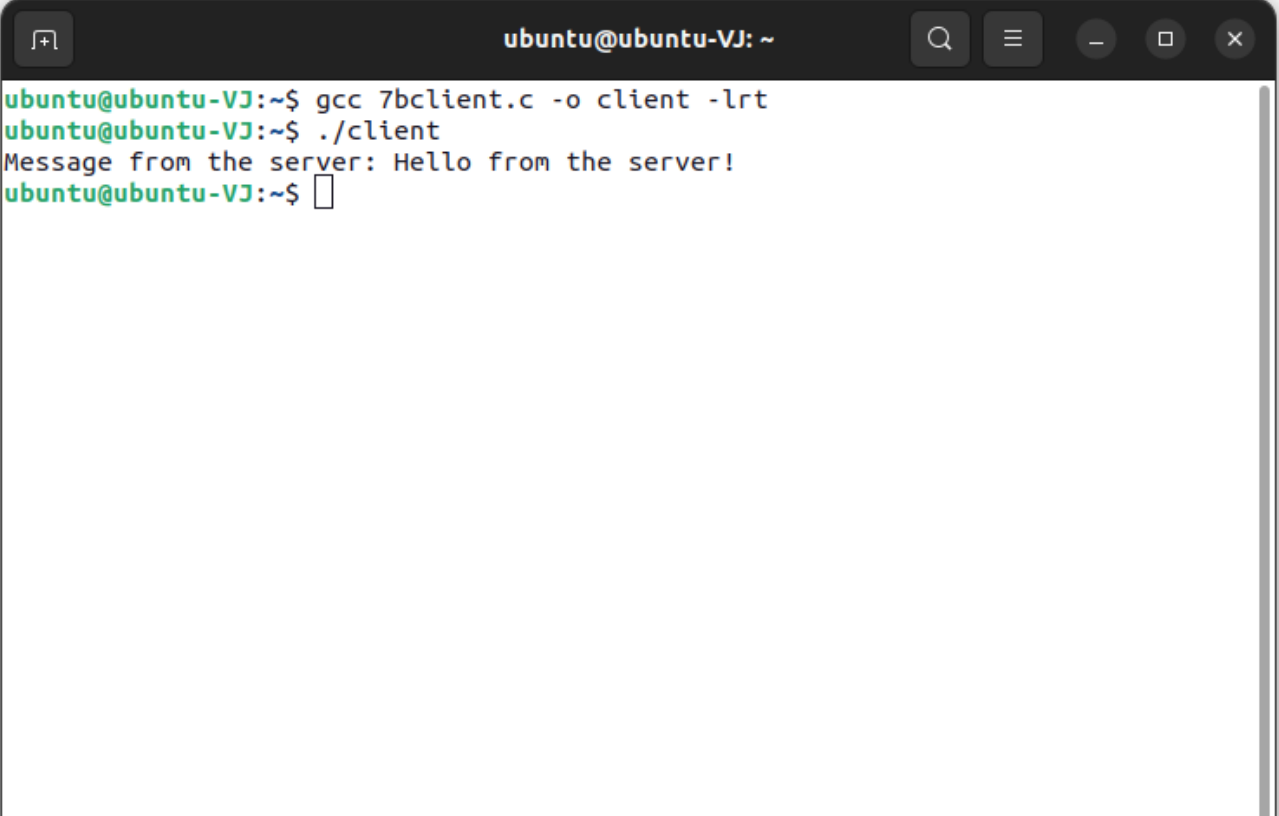
    return 0;
}
```

OUTPUT :-



```
ubuntu@ubuntu-VJ: ~  
ubuntu@ubuntu-VJ:~$ gcc 7bserver.c -o server -lrt  
ubuntu@ubuntu-VJ:~$ ./server  
ubuntu@ubuntu-VJ:~$
```

A terminal window with a dark title bar containing the text 'ubuntu@ubuntu-VJ: ~' and standard window controls. The terminal shows the compilation of '7bserver.c' into 'server' using 'gcc' with the '-lrt' flag, followed by the execution of './server'. The prompt returns to the shell after execution.



```
ubuntu@ubuntu-VJ: ~  
ubuntu@ubuntu-VJ:~$ gcc 7bclient.c -o client -lrt  
ubuntu@ubuntu-VJ:~$ ./client  
Message from the server: Hello from the server!  
ubuntu@ubuntu-VJ:~$
```

A terminal window with a dark title bar containing the text 'ubuntu@ubuntu-VJ: ~' and standard window controls. The terminal shows the compilation of '7bclient.c' into 'client' using 'gcc' with the '-lrt' flag, followed by the execution of './client'. The output of the client is 'Message from the server: Hello from the server!', and the prompt returns to the shell.

