ASSIGNMENT NO: 7-A

Title : Inter process communication in Linux using FIFOs

AIM:. Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.

THEORY:

- Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process. This usually occurs only in one system.
- Communication can be of two types: Between related processes initiating from only one process, such as parent and child processes. Between unrelated processes, or two or more different processes.

FIFOs

- FIFO is Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.
- A first-in, first-out (FIFO) file is a pipe that has a name in the filesystem.
- Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other.
- FIFOs are also called named pipes.
- You can make a FIFO using the mkfifo command.
- Specify the path to the FIFO on the command line.
- For example, create a FIFO in /tmp/fifo by invoking this:

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo prw-rw-rw-
1 samuel users 0 Jan 16 14:04 /tmp/fifo
```

- The first character of the output from ls is p, indicating that this file is actually a FIFO (named pipe). In one window, read from the FIFO by invoking the following:

```
% cat < /tmp/fifo
```

- In a second window, write to the FIFO by invoking this:

```
% cat > /tmp/fifo
```

- Then type in some lines of text.
- Each time you press Enter, the line of text is sent through the FIFO and appears in the first window.
- Close the FIFO by pressing Ctrl+D in the second window.
- Remove the FIFO with this line:

```
% rm /tmp/fifo
```

Creating a FIFO

- Create a FIFO programmatically using the mkfifo function.
- The first argument is the path at which to create the FIFO; the second parameter specifies the pipe's owner, group, and world permissions, and a pipe must have a reader and a writer, the permissions must include both read and write permissions.
- If the pipe cannot be created (for instance, if a file with that name already exists), mkfifo returns –

  1. Include<sys/types.h> and <sys/stat.h> if you call mkfifo.

Accessing a FIFO

- Access a FIFO just like an ordinary file.
- To communicate through a FIFO, one program must open it for writing, and another program must open it for reading.
- Either low-level I/O functions like open, write, read, close or C library I/O functions (fopen, fprintf, fscanf, fclose, and soon) may be used.
- For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:
```
intfd = open (fifo_path, O_WRONLY); write (fd, data, data_length);
close (fd);
```
- To read a string from the FIFO using C library I/O functions, you could use this code:
```
FILE* fifo = fopen (fifo_path, “r”); fscanf (fifo, “%s”, buffer);
fclose (fifo);
```
- A FIFO can have multiple readers or multiple writers.
- Bytes from each writer are written atomically up to a maximum size of PIPE_BUF (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

ASSIGNMENT No: 7-B

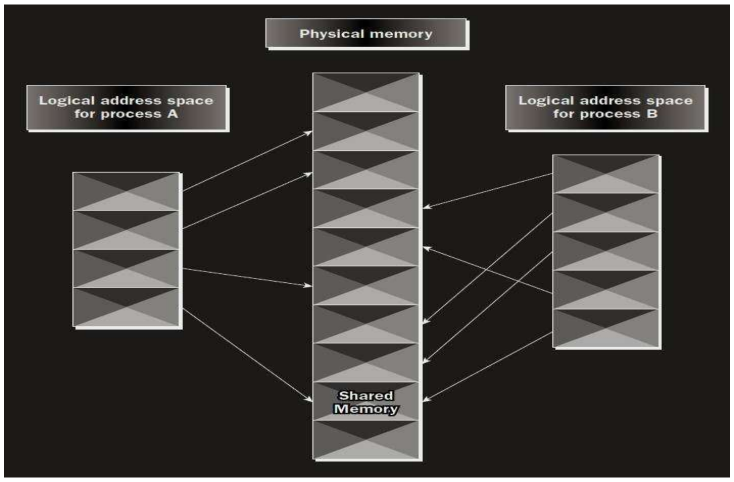
TITLE: Inter-process Communication using Shared Memory using System V.

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 shows an illustration of how shared memory works.


Physical memory
Logical address space for process A
Logical address space for process B
Shared Memory

- 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.
  Toenable 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 writing 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.

- 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.