ASSIGNMENT NO.: 07

AIM:

Inter Process Communication (IPC) in Linux using following.

A. FIFOS: 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 umber 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.

B. Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

PREREQUISITE:

1. C Programming
2. Fundamentals of Data Structure

OBJECTIVE:

To study

1. Communication (interface) between processes.
2. FIFO method of process communication.
3. Shared memory process of communication

THEORY:

Inter process communication in Linux

First in first out (FIFO)

A FIFO (First In First Out) is a one-way flow of data. FIFOs have a name, so unrelated processes can share the FIFO. FIFO is a named pipe. Any process can open or close the FIFO. FIFOs are also called named pipes.

Properties:

1. After a FIFO is created, it can be opened for read or write.
2. Normally, opening a FIFO for read or write, it blocks until another process opens it for write or read.
3. A read gets as much data as it requests or as much data as the FIFO has, whichever is less.
4. A write to a FIFO is atomic, as long as the write does not exceed the capacity of the FIFO.
5. Two processes must open FIFO; one opens it as reader on one end, the other opens it as sender on the other end. The first/earlier opener has to wait until the second/later opener to come. This is somewhat like a hand shaking.

Creating a FIFO

A FIFO is created by the mkfifo function. Specify the path to the FIFO on the command line. For example, create a FIFO in /tmp/fifo by invoking this:

```
#include <sys/types.h>
#include <sys/stat.h>
intmkfifo(const char *pathname, mode_t mode);
```

pathname: a UNIX pathname (path and filename). The name of the FIFO

mode: the file permission bits. It 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.

FIFO can also be created by the mknod system call,

e.g., mknod("fifo1", S_IFIFO|0666, 0) is same as mkfifo("fifo1", 0666).

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);
```

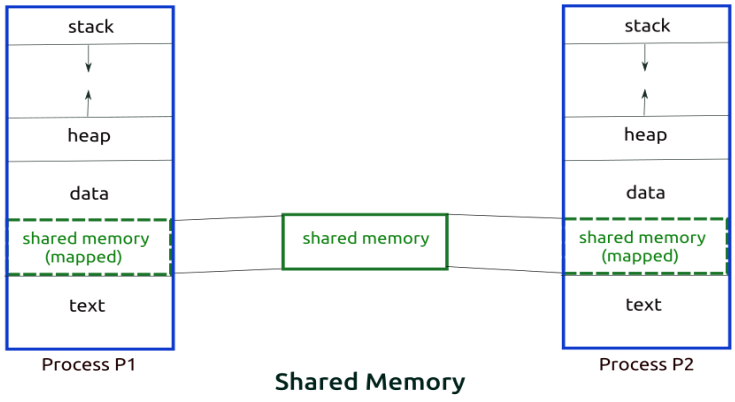
A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of PIPE_BUF (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

Close: To close an open FIFO, use close().

Unlink : To delete a created FIFO, use unlink().

Inter-process Communication using Shared Memory using System V.

Shared memory is one of the three inter-process communication (IPC) mechanisms available under Linux and other Unix-like systems. The other two IPC mechanisms are the message queues and semaphores. In case of shared memory, a shared memory segment is created by the kernel and mapped to the data segment of the address space of a requesting process. A process can use the shared memory just like any other global variable in its address space.



In the inter-process communication mechanisms like the pipes, fifos and message queues, the work involved in sending data from one process to another is like this. Process *P1* makes a system call to send data to Process *P2*. The message is copied from the address space of the first process to the kernel space during the system call for sending the message. Then, the second process makes a system call to receive the message. The message is copied from the kernel space to the address space of the second process. The shared memory mechanism does away with this copying overhead. The first process simply writes data into the shared memory segment. As soon as it is written, the data becomes available to the second process. Shared memory is the fastest mechanism for inter-process communication.

We know that to communicate between two or more processes, we use shared memory but before using the shared memory what needs to be done with the system calls, let us see this −

- Create the shared memory segment or use an already created shared memory segment (shmget())
- Attach the process to the already created shared memory segment (shmat())
- Detach the process from the already attached shared memory segment (shmdt())
- Control operations on the shared memory segment (shmctl())

Let us look at a few details of the system calls related to shared memory.

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


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

The above system call creates or allocates a System V shared memory segment. The arguments that need to be passed are as follows −

The **first argument, key,** recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok(). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process

communiation. If the client wants to use shared memory with this key, then it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory.

The **second argument, size,** is the size of the shared memory segment rounded to multiple of PAGE_SIZE.

The **third argument, shmflg,** specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.

This call would return a valid shared memory identifier (used for further calls of shared memory) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

#include <sys/shm.h>

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

The above system call performs shared memory operation for System V shared memory segment i.e., attaching a shared memory segment to the address space of the calling process. The arguments that need to be passed are as follows −

shmid is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

shmaddr is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.

shmflg is specifies the required shared memory flag/s such as SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing

mapping in the range specified by shmaddr and continuing till the end of segment).

This call would return the address of attached shared memory segment on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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


int shmdt(const void *shmaddr)
```

he above system call performs shared memory operation for System V shared memory segment of detaching the shared memory segment from the address space of the calling process. The argument that needs to be passed is −

The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

The above system call performs control operation for a System V shared memory segment. The following arguments needs to be passed −

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, cmd, is the command to perform the required control operation on the shared memory segment.

Let us consider the following sample program.

- Create two processes, one is for writing into the shared memory (shm_write.c) and another is for reading from the shared memory (shm_read.c)
- The program performs writing into the shared memory by write process (shm_write.c) and reading from the shared memory by reading process (shm_read.c)
- In the shared memory, the writing process, creates a shared memory of size 1K (and flags) and attaches the shared memory
- The write process writes 5 times the Alphabets from 'A' to 'E' each of 1023 bytes into the shared memory. Last byte signifies the end of buffer
- Read process would read from the shared memory and write to the standard output
- Reading and writing process actions are performed simultaneously
- After completion of writing, the write process updates to indicate completion of writing into the shared memory (with complete variable in struct shmseg)
- Reading process performs reading from the shared memory and displays on the output until it gets indication of write process completion (complete variable in struct shmseg)
- Performs reading and writing process for a few times for simplication and also in order to avoid infinite loops and complicating the program

Code for write process (Writing into Shared Memory – File: shm_write.c)

```
/* Filename: shm_write.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
```

```
#define BUF_SIZE 1024
#define SHM_KEY 0x1234

struct shmseg {
   int cnt;
   int complete;
   char buf[BUF_SIZE];
};
int fill_buffer(char * bufptr, int size);

int main(int argc, char *argv[]) {
   int shmid, numtimes;
   struct shmseg *shmp;
   char *bufptr;
   int spaceavailable;
   shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);
   if (shmid == -1) {
      perror("Shared memory");
      return 1;
   }

   // Attach to the segment to get a pointer to it.
   shmp = shmat(shmid, NULL, 0);
   if (shmp == (void *) -1) {
      perror("Shared memory attach");
      return 1;
   }
      /* Transfer blocks of data from buffer to shared memory */
   bufptr = shmp->buf;
   spaceavailable = BUF_SIZE;
```

```
   for (numtimes = 0; numtimes < 5; numtimes++) {
      shmp->cnt = fill_buffer(bufptr, spaceavailable);
      shmp->complete = 0;
      printf("Writing Process: Shared Memory Write: Wrote %d bytes\n", shmp-
>cnt);
      bufptr = shmp->buf;
      spaceavailable = BUF_SIZE;
      sleep(3);
   }
   printf("Writing Process: Wrote %d times\n", numtimes);
   shmp->complete = 1;

   if (shmdt(shmp) == -1) {
      perror("shmdt");
      return 1;
   }

   if (shmctl(shmid, IPC_RMID, 0) == -1) {
      perror("shmctl");
      return 1;
   }
   printf("Writing Process: Complete\n");
   return 0;
}

int fill_buffer(char * bufptr, int size) {
   static char ch = 'A';
   int filled_count;
 //printf("size is %d\n", size);
   memset(bufptr, ch, size - 1);
   bufptr[size-1] = '\0';
   if (ch > 122)
```

```
  ch = 65;
  if ( (ch >= 65) && (ch <= 122) ) {
    if ( (ch >= 91) && (ch <= 96) ) {
      ch = 65;
    }
  }
  filled_count = strlen(bufptr);
    //printf("buffer count is: %d\n", filled_count);
  //printf("buffer filled is:%s\n", bufptr);
  ch++;
  return filled_count;
}
```

Compilation and Execution Steps

```
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Wrote 5 times
Writing Process: Complete
```

Code for read process (Reading from the Shared Memory and writing to the standard output – File: shm_read.c)

```
/* Filename: shm_read.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>

#define BUF_SIZE 1024
#define SHM_KEY 0x1234
```

```
struct shmseg {
   int cnt;
   int complete;
   char buf[BUF_SIZE];
};
int main(int argc, char *argv[]) {
   int shmid;
   struct shmseg *shmp;
   shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);
   if (shmid == -1) {
      perror("Shared memory");
      return 1;   }
  // Attach to the segment to get a pointer to it.
   shmp = shmat(shmid, NULL, 0);
   if (shmp == (void *) -1) {
      perror("Shared memory attach");
      return 1;   }
 /* Transfer blocks of data from shared memory to stdout*/
   while (shmp->complete != 1) {
      printf("segment contains : \n\"%s\"\n", shmp->buf);
      if (shmp->cnt == -1) {
         perror("read");
         return 1;       }
      printf("Reading Process: Shared Memory: Read %d bytes\n", shmp->cnt);
      sleep(3);
   }
   printf("Reading Process: Reading Done, Detaching Shared Memory\n");
   if (shmdt(shmp) == -1) {
      perror("shmdt");
      return 1;
   }
```

```
    printf("Reading Process: Complete\n");
    return 0;
}
```

CONCLUSION:

Thus, we studied inter process communication using FIFOs