Assignment no -07

Title : Interprocess communication in Linux

Problem statement :

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 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.

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.

Theory :

FIFO Special Files

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system by calling mkfifo.

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

The mkfifo function is declared in the header file sys/stat.h. Function:

int mkfifo (const char *filename, mode_t mode)

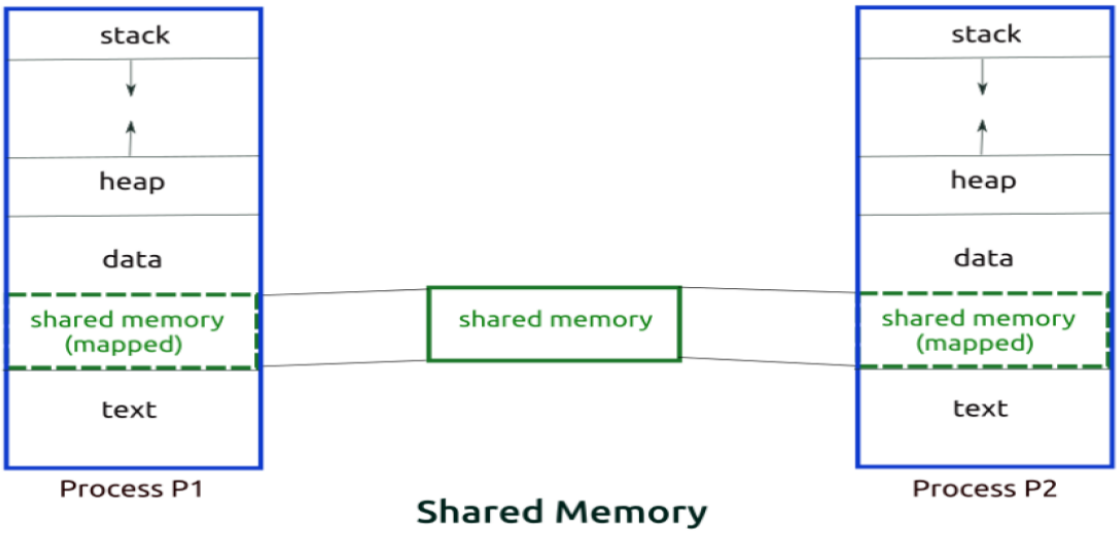
The mkfifo function makes a FIFO special file with name filename. The mode argument is used to set the file's permissions; see Setting Permissions.

The normal, successful return value from mkfifo is 0. In the case of an error, -1 is returned.

Shared Memory

Shared memory is one of the three interprocess 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 interprocess 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 interprocess communication.



A shared memory segment is identified by a unique integer, the shared memory ID. The shared memory itself is described by a structure of type shmid_ds in header file sys/shm.h. To use this file, files sys/types.h and sys/ipc.h must be included. Therefore, program should start with the following lines:

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

A general scheme of using shared memory is the following:

For a server, it should be started before any client. The server should perform the following tasks:

1. Ask for a shared memory with a memory key and memorize the returned shared memory ID.

This is performed by system call shmget( ).

2. Attach this shared memory to the server's address space with system call shmat( ).

3. Initialize the shared memory, if necessary.

4. Do something and wait for all clients' completion.

5. Detach the shared memory with system call shmdt( ).

6. Remove the shared memory with system call shmctl( ).

For the client part, the procedure is almost the same:

1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.

2. Attach this shared memory to the client's address space.

3. Use the memory.

4. Detach all shared memory segments, if necessary.

5. Exit.

shmget( )

The system call that requests a shared memory segment is shmget( ). It is defined as follows:

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

In the above definition, k is of type key_t or IPC_PRIVATE. It is the numeric key to be assigned to the returned shared memory segment. size is the size of the requested shared memory. The purpose of flag is to specify the way that the shared memory will be used. For our purpose, only the following two values are important:

1. IPC_CREAT | 0666 for a server (i.e., creating and granting read and write access to the server)

2. 0666 for any client (i.e., granting read and write access to the client)

If shmget( ) can successfully get the requested shared memory, its function value is a non-negative integer, the shared memory ID; otherwise, the function value is negative.

Keys:

UNIX requires a key of type key_t defined in file sys/types.h for requesting resources such as shared memory segments, message queues and semaphores. A key is simply an integer of type key_t; however, you should not use int or long, since the length of a key is system dependent.

There are three different ways of using keys, namely:

1. a specific integer value (e.g., 123456)

2. a key generated with function ftok( )

3. a uniquely generated key using IPC_PRIVATE (i.e., a private key).

The first way is the easiest one; however, its use may be very risky since a process can access your resource as long as it uses the same key value to request that resource. The following example assigns 1234 to a key:

key_t SomeKey;

SomeKey = 1234;

The ftok( ) function has the following prototype:

key_t ftok (const char *path, int id);

Function ftok( ) takes a character string that identifies a path and an integer (usually a character) value, and generates an integer of type key_t based on the first argument with the value of id in the most significant position.

Attaching a Shared Memory Segment to an Address Space -

shmat( ) :

Suppose process 1, a server, uses shmget( ) to request a shared memory segment successfully. That shared memory segment exists somewhere in the memory, but is not yet part of the address space of process 1. Similarly, if process 2 requests the same shared memory segment with the same key value, process 2 will be granted the right to use the shared memory segment; but it is not yet part of the address space of process 2. To make a requested shared memory segment part of the address space of a process, use shmat( ).

After a shared memory ID is returned, the next step is to attach it to the address space of a process. This is done with system call shmat( ). The use of shmat( ) is as follows:

shm_ptr = shmat (int shm_id, char *ptr, int flag);

System call shmat( ) accepts a shared memory ID, shm_id, and attaches the indicated shared memory to the program's address space. The returned value is a pointer of type (void *) to the attached shared memory. Thus, casting is usually necessary. If this call is unsuccessful, the return value is -1. Normally, the second parameter is NULL. If the flag is SHM_RDONLY, this shared memory is attached as a read-only memory; otherwise, it is readable and writable.

Detaching and Removing a Shared Memory Segment - shmdt( ) and shmctl( ):

System call shmdt( ) is used to detach a shared memory. After a shared memory is detached, it cannot be used. However, it is still there and can be re-attached back to a process's address space, perhaps at a different address. To remove a shared memory, use shmctl( ).

The only argument to shmdt( ) is the shared memory address returned by shmat( ). Thus, the following code detaches the shared memory from a program:

shmdt (shm_ptr);

where shm_ptr is the pointer to the shared memory. This pointer is returned by shmat( ) when the shared memory is attached. If the detach operation fails, the returned function value is non-zero.

To remove a shared memory segment, use the following code:

shmctl (shm_id, IPC_RMID, NULL);

where shm_id is the shared memory ID. IPC_RMID indicates this is a remove operation. Note that after the removal of a shared memory segment, if you want to use it again,

you should use shmget( ) followed by shmat( ).

Conclusion

Write conclusion in your own words