LAB NO.: 6 **Date:**

INTERPROCESS COMMUNICATION

Objectives:

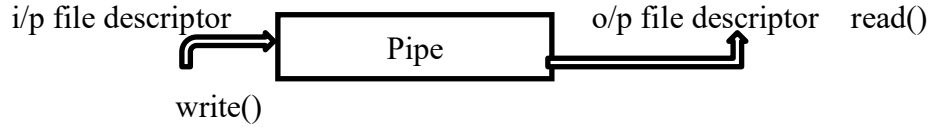
In this lab, student will be able to:

1. Gain knowledge as to how IPC (Interprocess Communication) happens between two processes.
2. Execute programs with respect to IPC using the different methods of message queues, pipes and shared memory.

Inter-Process communication (IPC), is the mechanism whereby one process can communicate with another process, i.e exchange of data. IPC in Linux can be implemented by using a pipe, shared memory and message queue.

Pipe

- Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process. A pipe is created using the system call pipe that returns a pair of file descriptors.



- Call to the pipe () function which returns an array of file descriptors fd[0] and fd [1]. fd [1] connects to the write end of the pipe, and fd[0] connects to the read end of the pipe. Anything can be written to the pipe, and read from the other end in the order it came in.
- A pipe is one directional providing one-way flow of data and it is created by the pipe() system call.

```
int pipe ( int *filedes ) ;
```

- Array of two file descriptors are returned- fd[0] which is open for reading , and fd[1] which is open for writing. It can be used only between parent and child processes.

  PROTOTYPE: int pipe(int fd[2]);

  RETURNS: 0 on success

  -1 on error: errno = EMFILE (no free descriptors)

EMFILE (system file table is full)
EFAULT (fd array is not valid)

fd[0] is set up for reading, fd[1] is set up for writing. i.e., the first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing.

```
#include <stdlib.h>
#include <stdio.h>      /* for printf */
#include <string.h>     /* for strlen */

int main(int argc, char **argv)
{
        int n;
        int fd[2];
        char buf[1025];
        char *data = "hello... this is sample data";
        pipe(fd);
        write(fd[1], data, strlen(data));
        if ((n = read(fd[0], buf, 1024)) >= 0) {
                buf[n] = 0;       /* terminate the string */
                printf("read %d bytes from the pipe: \"%s\"\n", n, buf);
        }
        else
                perror("read");
        exit(0);
}
```

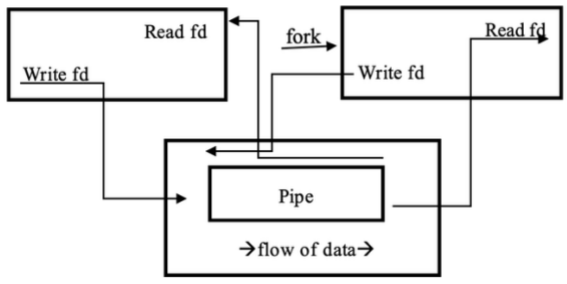


Fig. 6.1: Working of pipe in single process which is immediately after fork()

- First, a process creates a pipe and then forks to create a copy of itself.
- The parent process closes the read end of the pipe.
- The child process closes the write end of the pipe.
- The fork system call creates a copy of the process that was executing.
- The process which executes the fork is called the parent process and the new process is which is created is called the child process.

```
#include <sys/wait.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int pfd[2];
   pid_t cpid;
   char buf;
   assert(argc == 2);
   if (pipe(pfd) == -1) { perror("pipe");
      exit(EXIT_FAILURE); }
   cpid = fork();
   if (cpid == -1) { perror("fork");
   exit(EXIT_FAILURE); }

   if (cpid == 0) {    /* Child reads from pipe */
      close(pfd[1]);          /* Close unused write end */
      while (read(pfd[0], &buf, 1) > 0)
       write(STDOUT_FILENO, &buf, 1);
       write(STDOUT_FILENO, "\n", 1);
      close(pfd[0]);
      exit(EXIT_SUCCESS);

   } else {            /* Parent writes argv[1] to pipe */
      close(pfd[0]);          /* Close unused read end */
```

```
        write(pfd[1], argv[1], strlen(argv[1]));
        close(pfd[1]);          /* Reader will see EOF */
        wait(NULL);             /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

Message Queues

- It is an IPC facility. Message queues are similar to named pipes without the opening and closing of pipe. It provides an easy and efficient way of passing information or data between two unrelated processes.
- The advantages of message queues over named pipes is, it removes few difficulties that exists during the synchronization, the opening and closing of named pipes.
- A message queue is a linked list of messages stored within the kernel. A message queue is identified by a unique identifier. Every message has a positive long integer type field, a non-negative length, and the actual data bytes. The messages need not be fetched on FCFS basis. It could be based on type field.

Creating a Message Queue

- In order to use a message queue, it has to be created first. The msgget() system call is used for that. This system call accepts two parameters - a queue key and flags.
- IPC_PRIVATE - use to create a private message queue. A positive integer - used to create or access a publicly accessible message queue.

The message queue function definitions are

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

msgget

We create and access a message queue using the msgget function:

int msgget(key_t key, int msgflg);

The program must provide a key value that, as with other IPC facilities, names a particular message queue. The special value IPC_PRIVATE creates a private queue, which in theory is accessible only by the current process. The second parameter, msgflg, consists of nine permission

flags. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new message queue. It's not an error to set the IPC_CREAT flag and give the key of an existing message queue. The IPC_CREAT flag is silently ignored if the message queue already exists.

The msgget function returns a positive number, the queue identifier, on success or –1 on failure.

msgsnd

The msgsnd function allows us to add a message to a message queue:

int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

The structure of the message is constrained in two ways. First, it must be smaller than the system limit, and second, it must start with a long int, which will be used as a message type in the receive function. When you're using messages, it's best to define your message structure something like this:

```
struct my_message {
long int message_type;
/* The data you wish to transfer */
}
```

The first parameter, msqid, is the message queue identifier returned from a msgget function. The second parameter, msg_ptr, is a pointer to the message to be sent, which must start with a long int type as described previously. The third parameter, msg_sz, is the size of the message pointed to by msg_ptr. This size must not include the long int message type. The fourth parameter, msgflg, controls what happens if either the current message queue is full or the system wide limit on queued messages has been reached. If msgflg has the IPC_NOWAIT flag set, the function will return immediately without sending the message and the return value will be –1. If the msgflg has the IPC_NOWAIT flag clear, the sending process will be suspended, waiting for space to become available in the queue. On success, the function returns 0, on failure –1. If the call is successful, a copy of the message data has been taken and placed on the message queue.

msgrcv

The msgrcv function retrieves messages from a message queue:

int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);

The first parameter, msqid, is the message queue identifier returned from a msgget function. The second parameter, msg_ptr, is a pointer to the message to be received, which must start with a long int type as described above in the msgsnd function. The third parameter, msg_sz, is the size of the message pointed to by msg_ptr, not including the long int message type. The fourth parameter, msgtype, is a long int, which allows a simple form of reception priority to be implemented. If msgtype has the value 0, the first available message in the queue is retrieved. If it's greater than zero, the first message with the same message type is retrieved. If it's less than

zero, the first message that has a type the same as or less than the absolute value of msgtype is retrieved. This sounds more complicated than it actually is in practice. If you simply want to retrieve messages in the order in which they were sent, set msgtype to 0. If you want to retrieve only messages with a specific message type, set msgtype equal to that value. If you want to receive messages with a type of n or smaller, set msgtype to -n. The fifth parameter, msgflg, controls what happens when no message of the appropriate type is waiting to be received. If the IPC_NOWAIT flag in msgflg is set, the call will return immediately with a return value of –1. If the IPC_NOWAIT flag of msgflg is clear, the process will be suspended, waiting for an appropriate type of message to arrive. On success, msgrcv returns the number of bytes placed in the receive buffer, the message is copied into the user-allocated buffer pointed to by msg_ptr, and the data is deleted from the message queue. It returns –1 on error.

msgctl

The final message queue function is msgctl.

int msgctl(int msqid, int command, struct msqid_ds *buf);

The msqid_ds structure has at least the following members:

```
struct msqid_ds {
uid_t msg_perm.uid;
uid_t msg_perm.gid
mode_t msg_perm.mode;
}
```

The first parameter, msqid, is the identifier returned from msgget. The second parameter, command, is the action to take. It can take three values:

Command Description

Command	Description
IPC_STAT	Sets the data in the msqid_ds structure to reflect the values associated with the message queue.
IPC_SET	If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid_ds data structure.
IPC_RMID	Deletes the message queue.

0 is returned on success, –1 on failure. If a message queue is deleted while a process is waiting in a msgsnd or msgrcv function, the send or receive function will fail.

Receiver program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct my_msg_st {
        long int my_msg_type;
        char some_text[BUFSIZ];
};

int main()
{
        int running = 1;
        int msgid;
        struct my_msg_st some_data;
        long int msg_to_receive = 0;
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if (msgid == -1) {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }
        while(running) {
                if (msgrcv(msgid, (void *)&some_data, BUFSIZ,
                        msg_to_receive, 0) == -1) {
                        fprintf(stderr, "msgrcv failed with error: %d\n", errno);
                        exit(EXIT_FAILURE);
                }
                printf("You wrote: %s", some_data.some_text);
                if (strncmp(some_data.some_text, "end", 3) == 0) {
                        running = 0;
                }
        }
        if (msgctl(msgid, IPC_RMID, 0) == -1) {
                fprintf(stderr, "msgctl(IPC_RMID) failed\n");
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
```

```
}
```

Sender Program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512

struct my_msg_st {
        long int my_msg_type;
        char some_text[MAX_TEXT];
};

int main()
{
        int running = 1;
        struct my_msg_st some_data;
        int msgid;
        char buffer[BUFSIZ];
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if (msgid == -1) {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }
        while(running) {
                printf("Enter some text:");
                fgets(buffer, BUFSIZ, stdin);
                some_data.my_msg_type = 1;
                strcpy(some_data.some_text, buffer);
                if (msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0) == -1) {
                        fprintf(stderr, "msgsnd failed\n");
```

```
                exit(EXIT_FAILURE);
            }
            if (strncmp(buffer, "end", 3) == 0) {
                running = 0;
            }
        }
        exit(EXIT_SUCCESS);
}
```

Shared memory

Shared memory allows two or more processes to access the same logical memory. Shared memory is an efficient of transferring data between two running processes. Shared memory is a special range of addresses that is created by one process and the Shared memory appears in the address space of that process. Other processes then attach the same shared memory segment into their own address space. All processes can then access the memory location as if the memory had been allocated just like 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.

The functions for shared memory are,

#include <sys/shm.h>

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

The include files sys/types.h and sys/ipc.h are normally also required before shm.h is included.

shmget

We create shared memory using the shmget function:

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

The argument key 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.

The permission flags are very useful with shared memory because they allow a process to create shared memory that can be written by processes owned by the creator of the shared memory but only read by processes that other users have created. We can use this to provide efficient read-only access to data by placing it in shared memory without the risk of its being changed by other users.

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_RND, which, in conjunction with shm_addr, controls the address at which the shared memory is attached, and SHM_RDONLY, which makes the attached memory read-only. It's very rare to need to control the address at which shared memory is attached; you should normally allow the system to choose an address for you, as doing otherwise will make the application highly hardware-dependent. If the shmat call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.

The shared memory will have read or write access depending on the owner (the creator of the shared memory), the permissions, and the owner of the current process. Permissions on shared memory are similar to the permissions on files. An exception to this rule arises if shmflg & SHM_RDONLY is true. Then the shared memory won't be writable, even if permissions would have allowed write access.

shmdt

The shmdt function detaches the shared memory from the current process. It takes a pointer to the address returned by shmat. On success, it returns 0, on error –1. Note that detaching the shared memory doesn't delete it; it just makes that memory unavailable to the current process.

shmctl

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

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:

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 shmid_ds structure has the following members:

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

The third parameter, buf, is a pointer to structure containing the modes and permissions for the shared memory. On success, it returns 0, on failure returns –1.

We will write a pair of programs shm1.c and shm2.c. The first will create a shared memory segment and display any data that is written into it. The second will attach into an existing shared memory segment and enters data into shared memory segment.

First, we create a common header file to describe the shared memory we wish to pass around. We call this shm_com.h.

```
#define TEXT_SZ 2048

struct shared_use_st {
        int written_by_you;
        char some_text[TEXT_SZ];
};

//shm1.c – Consumer process
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
#include “shm_com.h”

int main()
{
        int running = 1;
        void *shared_memory = (void *)0;
        struct shared_use_st *shared_stuff;
        int shmid;
        srand((unsigned int)getpid());
        shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
        if (shmid == -1) {
                fprintf(stderr, “shmget failed\n”);
                exit(EXIT_FAILURE);
        }
        shared_memory = shmat(shmid, (void *)0, 0);
        if (shared_memory == (void *)-1) {
                fprintf(stderr, “shmat failed\n”);
                exit(EXIT_FAILURE);
        }
        printf(“Memory attached at %X\n”, (int)shared_memory);
        shared_stuff = (struct shared_use_st *)shared_memory;
        shared_stuff->written_by_you = 0;
        while(running) {
                if (shared_stuff->written_by_you) {
                printf(“You wrote: %s”, shared_stuff->some_text);
                sleep( rand() % 4 ); /* make the other process wait for us ! */
                shared_stuff->written_by_you = 0;
                if (strncmp(shared_stuff->some_text, “end”, 3) == 0) {
                        running = 0;
                }
        }
        if (shmdt(shared_memory) == -1) {
                fprintf(stderr, “shmdt failed\n”);
                exit(EXIT_FAILURE);
        }
        if (shmctl(shmid, IPC_RMID, 0) == -1) {
                fprintf(stderr, “shmctl(IPC_RMID) failed\n”);
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

//shm2.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm_com.h"
int main()
{
        int running = 1;
        void *shared_memory = (void *)0;
        struct shared_use_st *shared_stuff;
        char buffer[BUFSIZ];
        int shmid;
        shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
        if (shmid == -1) {
                fprintf(stderr, "shmget failed\n");
                exit(EXIT_FAILURE);
        }
        shared_memory = shmat(shmid, (void *)0, 0);
        if (shared_memory == (void *)-1) {
                fprintf(stderr, "shmat failed\n");
                exit(EXIT_FAILURE);
        }
        printf("Memory attached at %X\n", (int)shared_memory);
        shared_stuff = (struct shared_use_st *)shared_memory;
        while(running) {
                while(shared_stuff->written_by_you == 1) {
                        sleep(1);
                        printf("waiting for client...\n");
                }
                printf("Enter some text:");
                fgets(buffer, BUFSIZ, stdin);
                strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
                shared_stuff->written_by_you = 1;
                if (strncmp(buffer, "end", 3) == 0) {
                        running = 0;
                }
        }
        if (shmdt(shared_memory) == -1) {
                fprintf(stderr, "shmdt failed\n");
                exit(EXIT_FAILURE);
```

```
        }
        exit(EXIT_SUCCESS);
}
```

Named Pipes: FIFOs

Pipes can share data between related processes, i.e. processes that have been started from a common ancestor process. We can use named pipe or FIFOs to overcome this. A named pipe is a special type of file that exists as a name in the file system but behaves like the unnamed pipes we have discussed already.We can create named pipes from the command line using

$ mkfifo filename

From inside a program, we can use

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

int mkfifo(const char *filename, mode_t mode);

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
    {
        int res = mkfifo("/tmp/my_fifo", 0777);
        if (res == 0) printf("FIFO created\n");
        exit(EXIT_SUCCESS);
    }
```

We can look for the pipe with

$ **ls -lF /tmp/my_fifo**

prwxr-xr-x 1 rick users 0 July 10 14:55 /tmp/my_fifo|

Notice that the first character of output is a p, indicating a pipe. The | symbol at the end is added by the ls command's -F option and also indicates a pipe. We can remove the FIFO just like a conventional file by using the rm command, or from within a program by using the unlink system

call.

Producer-Consumer Problem (PCP):

- Producer process produces information that is consumed by a consumer process. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by consumer. Two types of buffers can be used.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.
- For bounded-buffer PCP basic synchronization requirement is:
  - Producer should not write into a full buffer (i.e. producer must wait if the buffer is full)
  - Consumer should not read from an empty buffer (i.e. consumer must wait if the buffer is empty)
  - All data written by the producer must be read exactly once by the consumer

Following is a program for Producer-Consumer problem using named pipes.

//producer.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
        int pipe_fd;
        int res;
        int open_mode = O_WRONLY;
        int bytes_sent = 0;
        char buffer[BUFFER_SIZE + 1];
        if (access(FIFO_NAME, F_OK) == -1) {
```

```
                res = mkfifo(FIFO_NAME, 0777);
                if (res != 0) {
                        fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
                        exit(EXIT_FAILURE);
                }
        }
        printf("Process %d opening FIFO O_WRONLY\n", getpid());
        pipe_fd = open(FIFO_NAME, open_mode);
        printf("Process %d result %d\n", getpid(), pipe_fd);
        if (pipe_fd != -1) {
                while(bytes_sent < TEN_MEG) {
                        res = write(pipe_fd, buffer, BUFFER_SIZE);
                        if (res == -1) {
                                fprintf(stderr, "Write error on pipe\n");
                                exit(EXIT_FAILURE);
                        }
                        bytes_sent += res;
                }
                (void)close(pipe_fd);
        }
        else {
                exit(EXIT_FAILURE);
       }
        printf("Process %d finished\n", getpid());
        exit(EXIT_SUCCESS);
}
```

//consumer.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main()
{
        int pipe_fd;
        int res;
        int open_mode = O_RDONLY;
        char buffer[BUFFER_SIZE + 1];
        int bytes_read = 0;
        memset(buffer, '\0', sizeof(buffer));
        printf("Process %d opening FIFO O_RDONLY\n", getpid());
        pipe_fd = open(FIFO_NAME, open_mode);
        printf("Process %d result %d\n", getpid(), pipe_fd);
        if (pipe_fd != -1) {
                do {
                        res = read(pipe_fd, buffer, BUFFER_SIZE);
                        bytes_read += res;
                } while (res > 0);
                (void)close(pipe_fd);
        }
        else {
                exit(EXIT_FAILURE);
        }
        printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
        exit(EXIT_SUCCESS);
}
```

The Readers-Writers Problem:

- Concurrent processes share a file, record, or other resources
- Some may read only (readers), some may both read and write (writers)
- Two concurrent reads have no adverse effects
- Problems if
  - concurrent reads and writes
  - multiple writes

Two Variations

- First Readers-Writers problem: No reader be kept waiting unless a writer has already obtained exclusive write permissions (Readers have high priority)
- Second Readers-Writers problem: If a writer is waiting/ready , no new readers may start reading (Writers have high priority)

Lab Exercises:

1. Process A wants to send a number to Process B. Once received, Process B has to check whether the number is palindrome or not. Write a C program to implement this interprocess communication using message queue. .
2. Write a producer and consumer program in C using FIFO queue. The producer should write a set of 4 integers into the FIFO queue and the consumer should display the 4 integers.
3. Implement a parent process, which sends an English alphabet to child process using shared memory. Child process responds back with next English alphabet to the parent. Parent displays the reply from the Child.
4. Write a producer-consumer program in C in which producer writes a set of words into shared memory and then consumer reads the set of words from the shared memory. The shared memory need to be detached and deleted after use.

Additional Exercises:

1. Demonstrate creation, writing to and reading from a pipe.
2. Demonstrate creation of a process which writes through a pipe while the parent process reads from it.
3. Write a program which creates a message queue and writes message into queue which contains number of users working on the machine along with observed time in hours and minutes. This is repeated for every 10 minutes. Write another program which reads this information form the queue and calculates on average in each hour how many users are working.