8 Experiment 8: Inter Process Communication (IPC)

8.1 Objective

The aim of this laboratory is to introduce the Interprocess communication (IPC) mechanism of operating system to allow the processes to communicate with each other.

8.2 Reading Material[1][3][2]

8.2.1 Interprocess communication

Interprocess communication (IPC) involves methods and tools that enable different processes on a computer system to exchange data, coordinate activities, and synchronize operations. It allows programs to communicate with each other, facilitating collaboration and data sharing between processes running concurrently. Examples of IPC methods include pipes, sockets, shared memory, and message queues.

8.2.2 Pipes

In C programming, pipes are a form of interprocess communication (IPC) that allows communication between two processes, with one process writing data into the pipe and the other process reading from it. Pipes are one-way communication channels that can be either anonymous or named.

Function	Description	Programming Syntax
pipe()	Creates an anonymous pipe, returning two file descriptors - one for reading and one for writing.	`int pipe(int filedes[2]);`
mkfifo()	Creates a named pipe (FIFO) in the file system.	`int mkfifo(const char *pathname, mode_t mode);`
dup()	Duplicates a file descriptor, creating a copy of the specified descriptor.	`int dup(int oldfd);`
close()	Closes a file descriptor.	`int close(int fd);`

Table 8: Functions related to pipes

8.2.3 Unnamed Pipes (Anonymous Pipes)

Functioning: Unnamed pipes are created using the pipe() system call. They provide a one-way communication channel between two related processes, typically a parent and its child process. They allow communication by connecting the standard output (stdout) of one process to the standard input (stdin) of another.

- Syntax: `int pipe(int filedes[2]);`
- `filedes[0]` refers to the read end of the pipe.
- `filedes[1]` refers to the write end of the pipe.

Sample Program

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int pipe_fd[2];
    char data[100];

    if (pipe(pipe_fd) < 0) {
        prinft("pipe creation failed");
        return 1;
    }

    pid_t pid = fork();

    if (pid < 0) {
        printf("fork failed\n");
        return 1;
    }

    if (pid > 0) { // Parent process
        // Write to pipe
        close(pipe_fd[0]); // Close the reading end
        char message[] = "Hello, Child Process!";
        write(pipe_fd[1], message, sizeof(message));
        close(pipe_fd[1]); // Close the writing end
    } else { // Child process
        // Read from pipe
        close(pipe_fd[1]); // Close the writing end
        read(pipe_fd[0], data, sizeof(data));
        printf("Received message in child: %s\n", data);
        close(pipe_fd[0]); // Close the reading end
    }

    return 0;
}
```

8.2.4 Named Pipes

Named pipes, also known as FIFOs (First In, First Out), are created using the mkfifo() system call. They are files residing in the file system and allow communication between unrelated processes. Named pipes provide bi-directional communication.

- Syntax: `int mkfifo(const char *pathname, mode_t mode);`
- `pathname` is the path and name of the named pipe.

- mode specifies the permissions for the named pipe.

Sample Code[3][1]

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main() {
    char *fifo = "/tmp/myfifo"; // Path to the named pipe

    mkfifo(fifo, 0666); // Creating a named pipe

    int fd;
    char data[100];

    fd = open(fifo, O_WRONLY); // Open the pipe for writing
    write(fd, "Hello from writer!", sizeof("Hello from writer!"));
    close(fd);

    fd = open(fifo, O_RDONLY); // Open the pipe for reading
    read(fd, data, sizeof(data));
    printf("Received message: %s\n", data);
    close(fd);

    return 0;
}
```

8.2.5 Shared Memory

Shared memory in C programming is a mechanism that allows multiple processes to share a region of memory. This shared memory segment is created by one process and can be accessed by multiple processes, enabling efficient inter-process communication (IPC).
The primary steps involved in using shared memory are:

1. Allocation: A process allocates a shared memory segment using the `shmget()` system call. This call either creates a new shared memory segment or accesses an existing one based on a provided key and size.
2. Attachment: After allocation, the process attaches the shared memory segment to its address space using `shmat()`. This attaches the segment to a virtual address in the process's memory, allowing it to read from and write to the shared memory.
3. Usage: Processes that share this segment can read from and write to it, treating it like any other memory region. Synchronization mechanisms such as semaphores or mutexes are typically used to control access and prevent race conditions.

→ In this two type of functions
① Shmget - Shared memory get - use to create shared memory segment
② Shmat - shared memory attach - to attach the shared segment to address space of a process

① shmget, 1st parameter - unique number which identify shared segment (any number)
2nd parameter - size of shm segment

4. Detachment: When the process finishes using the shared memory, it detaches the segment using `shmdt()`. This detaches the shared memory segment from the process's address space.

5. Control Operations: The `shmctl()` function allows for control operations on the shared memory segment, such as removing or modifying it.

Function	Description	Programming Syntax
`shmget()`	Allocates a new shared memory segment or accesses an existing one.	`int shmget(key_t key, size_t size, int shmflg);`
`shmat()`	Attaches the shared memory segment to the address space of the calling process.	`void *shmat(int shmid, const void *shmaddr, int shmflg);`
`shmdt()`	Detaches the shared memory segment from the calling process.	`int shmdt(const void *shmaddr);`
`shmctl()`	Performs control operations on the shared memory segment, such as removing or modifying it.	`int shmctl(int shmid, int cmd, struct shmid_ds *buf);`

Table 9: Functions related to shared memory

8.2.6 Sample program to demonstrate shared memory segment creation and data addition

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <string.h>
#include <errno.h>

#define SHM_SIZE 1024

int main() {
    void *shm;
    char buf[100];
    int shmid;

    // Creating a shared memory segment
    shmid = shmget((key_t)123, SHM_SIZE, 0666 | IPC_CREAT);
    if (shmid == -1) {
        printf("shmget - Error\n");
        exit(EXIT_FAILURE);
    }
    printf("The Key value of shared memory is %d\n", shmid);

    // Attaching the process to the shared memory segment
    shm = shmat(shmid, NULL, 0);
```

```
    if (shm == (void *)-1) {
        printf("shmat - Error\n");
        exit(EXIT_FAILURE);
    }
    printf("Process attached to the address of %p\n", shm);

    printf("Write the data to shared memory (max 99 characters): ");
    fgets(buf, sizeof(buf), stdin);
    buf[strcspn(buf, "\n")] = '\0'; // Removing newline character if present

    strncpy((char *)shm, buf, SHM_SIZE); // Ensuring data doesn't exceed SHM_SIZE
    printf("The stored data in shared memory is: %s\n", (char *)shm);

    // Detaching shared memory
    if (shmdt(shm) == -1) {
        printf("shmdt Error\n");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

8.2.7 Message Queues

Function	Description	Programming Syntax
`msgget()`	Creates a new message queue or gets the identifier of an existing queue.	`int msgget(key_t key, int msgflg);`
`msgsnd()`	Sends a message to a message queue.	`int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);`
`msgrcv()`	Receives a message from a message queue.	`ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);`
`msgctl()`	Performs control operations on a message queue, such as deleting or modifying it.	`int msgctl(int msqid, int cmd, struct msqid_ds *buf);`

Table 10: Funtions related to message queues

8.2.8 Message Queue Implementation in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

#define MAX_MSG_SIZE 100

struct msg_buffer {
    long msg_type;
    char msg_text[MAX_MSG_SIZE];
};

int main() {
    key_t key;
    int msg_id;
    struct msg_buffer message;

    // Manually generate a key without using ftok
    if ((key = 0x12345678) == -1) {
        printf("key Error\n");
        exit(EXIT_FAILURE);
    }

    if ((msg_id = msgget(key, 0666 | IPC_CREAT)) == -1) {
        printf("msgget Error\n");
        exit(EXIT_FAILURE);
    }

    printf("Message Queue Created with ID: %d\n", msg_id);

    printf("Enter a message to send to the queue: ");
    fgets(message.msg_text, MAX_MSG_SIZE, stdin);
    message.msg_type = 1;

    if (msgsnd(msg_id, &message, sizeof(message), 0) == -1) {
        printf("msgsnd Error\n");
        exit(EXIT_FAILURE);
    }

    printf("Message Sent to the Queue\n");

    if (msgrcv(msg_id, &message, sizeof(message), 1, 0) == -1) {
        printf("msgrcv Error\n");
        exit(EXIT_FAILURE);
    }
```

```
    printf("Message Received from the Queue: %s\n", message.msg_text);
    if (msgctl(msg_id, IPC_RMID, NULL) == -1) {
        printf("msgctl Error\n");
        exit(EXIT_FAILURE);
    }
    printf("Message Queue Removed\n");

    return 0;
}
```

Video Reference:

https://youtube.com/playlist?list=PLWjmN065fOfGdAZrlP6316HVHh8jlve

References

[1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts.* Wiley, 10 edition, 2018.

[2] Sumitabha Das. *Unix Concepts And Applications.* Wiley, 4 edition, 2006.

[3] Richard Stones Neil Matthew. *Beginning Linux Programming.* Wiley, 4 edition, 2007.

8.3 Lab Exercises

Exercise 1: Develop a program that demonstrates Inter-Process Communication (IPC) using named pipes.
Tasks:

- Create a pair of named pipes: one for sending data and another for receiving data.
- Develop a sender program that writes a message to the sending pipe.
- Create a receiver program that reads from the receiving pipe and displays the received message.

Exercise 2: Demonstrate the usage of Shared Memory for IPC.
Tasks: