Lab Assignment 4

**Title:** **Process Management (Inter-process Communication)**

**Problem statement:**

Write a program to Implement Pipe and / Shared Memory Concept.

**Objectives:**

* To understand the concept of Pipe.
* To use read/write file descriptors.
* To understand Inter-Process Communication mechanism in Parent and child process.

**Theory:**

**Pipes:**

One of the most significant contributions of UNIX to the development of operating systems is the pipe. Inspired by the concept of co-routines [RITC84], a pipe is a circular buffer allowing two processes to communicate on the producer–consumer model. Thus, it is a first-in-first-out queue, written by one process and read by another. When a pipe is created, it is given a fixed size in bytes. When a process attempts to write into the pipe, the write request is immediately executed if there is sufficient room; otherwise the process is blocked. Similarly, a reading process is blocked if it attempts to read more bytes than are currently in the pipe; otherwise the read request is immediately executed. The OS enforces mutual exclusion: that is, only one process can access a pipe at a time. There are two types of pipes: named and unnamed. Only related processes can share unnamed pipes, while either related or unrelated processes can share named pipes.

Read/Write

Pipe

Write/Read

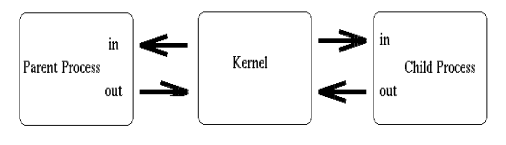
**File Descriptors:** When a process creates a pipe, the kernel sets up two file descriptors for use by the pipe. One descriptor is used to allow a path of input into the pipe (write), i.e. with value 1 while the other is used to obtain data from the pipe (read), i.e. with value 1. At this point, the pipe is of little practical use, as the creating process can only use the pipe to communicate with itself. Consider this representation of a process and the kernel after a pipe has been created:

**Messages**

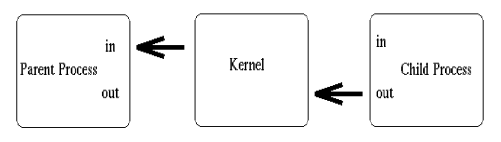
A message is a block of bytes with an accompanying type. UNIX provides msgsnd and msgrcv system calls for processes to engage in message passing. Associated with each process is a message queue, which functions like a mailbox. The message sender specifies the type of message with each message sent, and this can be used as a selection criterion by the receiver. The receiver can either retrieve messages in first-in-first-out order or by type. A process will block when trying to send a message to a full queue. A process will also block when trying to read from an empty queue. If a process attempts to read a message of a certain type and fails because no message of that type is present, the process is not blocked.

**Shared Memory**

The fastest form of interprocess communication provided in UNIX is shared memory. This is a common block of virtual memory shared by multiple processes. Processes read and write shared memory using the same machine instructions they use to read and write other portions of their virtual memory space. Permission is read-only or read-write for a process, determined on a per-process basis. Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory



Above, we see that both processes now have access to the file descriptors which constitute the pipeline. It is at this stage, that a critical decision must be made. In which direction do we desire data to travel? Does the child process send information to the parent, or vice-versa? The two processes mutually agree on this issue, and proceed to ``close'' the end of the pipe that they are not concerned with. For discussion purposes, let's say the child performs some processing, and sends information

back through the pipe to the parent. 

Construction of the pipeline is now complete! The only thing left to do is make use of the pipe. To access a pipe directly, the same system calls that are used for low-level file I/O can be used (recall that pipes are actually represented internally as a valid inode).

To send data to the pipe, we use the write() system call, and to retrieve data from the pipe, we use the read() system call. Remember, low-level file I/O system calls work with file descriptors! However, keep in mind that certain system calls, such as lseek(), do not work with descriptors to pipes.

**Algorithm:**

* 1. Read the number of elements n.
  2. Create a child process using fork, which returns zero to child process and some nonzero positive integer to parent.
  3. In child process print all the odd numbers from 1 to n.
  4. In child process use exit to terminate child.
  5. In parent process print all the even numbers from 1 to n.
  6. In parent process use wait system call to collect the exit status of child.
  7. Terminate parent

**Input:** Message passed from parent

**Output:** Message received by Child

**Conclusion**: Thus we have studied the concept of Inter Process Communication with the help of pipe mechanism.

**FAQs**

1. What is pipe and how does it work?
2. Which are the types of pipe? Expain.
3. What do you mean by broken pipe?