Topic to be covered

- Pipes
  - Theoretical concept
  - Pipe creation
  - Closing pipe end
- Writing to a file descriptor
- Reading from a file descriptor

Objectives

- Students will be able to do communicate between two or more processes having Parent-child or sibling relations using UNIX ordinary/unnamed pipes.

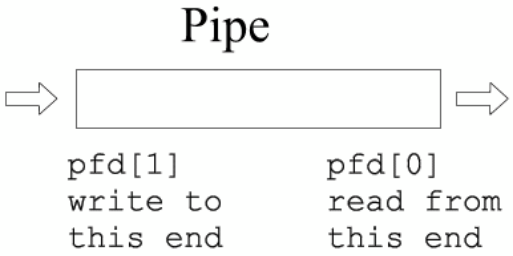
Systems Calls

- Pipe(1)
- Close(1)
- Write(3)
- Read(3)

Pipe Creation

The pipe function has the prototype

```
int pipe(int pfd[2]);
```



pipe() creates a pipe, a unidirectional data channel that can be used for inter-process communication. The array "pfd" is used to return two file descriptors referring to the ends of the pipe. pfd[0] refers to the read end of the pipe. pfd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, type "man pipe" on the shell. Pipe() is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors.

- **On Success:-**
  - It returns zero.

- **On Failure**
  - It returns -1 and sets errno to indicate the reason for failure

Errors:-

EMFILE	Too many file descriptors are in use by the process.
ENFILE	The system file table is full.
EFAULT	The file descriptor is not valid.

I/O with a pipe:

These two file descriptors can be used for a block I/O

ssize_t **write** (int *fildes*, const void **buf*, size_t *nbyte*);

The write() function shall attempt to write nbyte bytes from the buffer pointed to by buf to the file associated with the open file descriptor, fildes.
It may return: Complete: ret=nbyte or Partial: ret<nbyte

ssize_t **read** (int *fildes*, **void ***buf**, size_t** *nbyte***);**

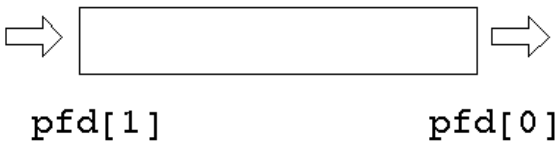
The read() function shall attempt to read nbyte bytes from the file associated with the open file descriptor, fildes, into the buffer pointed to by buf. The behavior of multiple concurrent reads on the same pipe, FIFO, or terminal device is unspecified.

Upon successful completion, read()shall return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions shall return -1 and set *errno* to indicate the error.

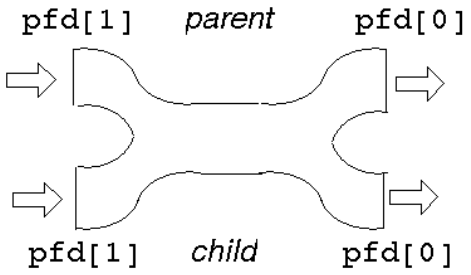
Fork and a pipe:

A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using fork ( ). A pipe opened before the fork becomes shared between the two processes.

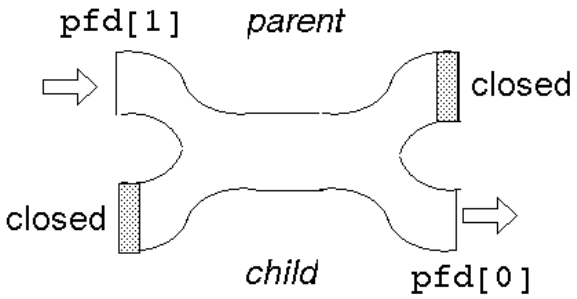
Before fork



After fork



This gives *two* read ends and *two* write ends. The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.
Either process can write into the pipe, and either can read from it. Which process will get what is not known. For predictable behavior, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again.



Suppose the parent wants to write down a pipeline to a child. The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end.
When the processes have ceased communication, the parent closes its write end. This means that the child gets eof on its next read, and it can close its read end. Read() system call gets blocked until the data is available. And the write call is blocked only when the pipe is completely filled.

Pipes use the buffer cache just as ordinary files do. Therefore, the benefits of writing and reading pipes in units of a block (usually 512 bytes) are just as great. A

single **write** execution is atomic, so if 512 bytes are written with a single system call, the corresponding read will return with 512 bytes (if it requests that many). It will not return with less than the full block. However, if the writer is not writing complete blocks, but the reader is trying to read complete blocks, the reader may keep getting partial blocks anyhow. This won't happen if the writer is faster than the reader since then the writer will be able to fill the pipe with a complete block before the reader gets around to reading anything. Still, it's best to buffer writes and reads on pipes, and this is what the Standard I/O Library does automatically.

Given that we have two processes, how can we connect them so that one can read from a pipe what the other writes? We can't. Once the processes are created they can't be connected, because there's no way for the process that makes the pipe to pass a file descriptor to the other process. It can pass the file descriptor number, of course, but that number won't be valid in the other process. But if we make a pipe in one process *before creating* the other process, it will inherit the pipe file descriptors, and they will be valid in both processes. Thus, two processes communicating over a pipe can be parent and child, or two children, or grandparent and grandchild, and so on, but they must be related, and the pipe must be passed on at birth. In practice, this may be a severe limitation, because if a process dies there's no way to recreate it and reconnect it to its pipes -- the survivors must be killed too, and then the whole family has to be recreated.

In general, then, here is how to connect two processes with a pipe:

- Make the pipe.
- Fork to create the reading child.
- Close the read end of the pipe in the writer process and write end of the pipe in the reader process. The process who is not using pipe at all closes both ends of the pipe.
- The writer process writes to the write end of the pipe using the write() system call.
- The reader process reads the read-end of the pipe using the read() system call.

Closing a File Descriptor

int **close**(int fd);

close() closes a file descriptor so that it no longer refers to any file and may be reused. close() returns zero on success. On error, -1 is returned, and errno is set appropriately.

Sample Code 1

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
        int data_processed=0;
        int file_pipes[2];
        const char some_data[] = "123";
        char buffer[BUFSIZ + 1];
        memset(buffer, '\0', sizeof(buffer));
        if (pipe(file_pipes) == 0)
         {
                data_processed = write(file_pipes[1], some_data,
        strlen(some_data));
                printf("Wrote %d bytes\n", data_processed);
                data_processed = read(file_pipes[0], buffer, BUFSIZ);
                printf("Read %d bytes: %s\n", data_processed, buffer);
                exit(EXIT_SUCCESS);
        }
        exit(EXIT_FAILURE);
}
```

Output

Here's a small program that uses a pipe to allow the parent to read a message from its child:

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

#define  READ   0
#define  WRITE  1

main ( )
{
char*  phrase  =  "This is OS lab time" ;
int     fd [2],  bytesread ;
char    message [100] ;

pipe ( fd ) ;
if ( fork (  )  ==  0 )                                  /*  child, writer */
{
close ( fd [READ] ) ;                                  /*  close unused end */
write  ( fd [WRITE], phrase, strlen (phrase) + 1) ;
close ( fd [WRITE] ) ;                                  /*  close used end  */
}
else                                                          /*  parent, reader  */
{
close ( fd [WRITE] ) ;                /* close unused end  */
bytesread = read (fd [READ], message, 100) ;
printf ("Read %d bytes : %s\n", bytesread, message) ;
close ( fd [READ] ) ;                                  /*  close used end  */
}
}
```

Output

Graded Task 1:

Write a C program in which the parent process receives an integer array through command-line arguments and passes it to its child process using pipe, The child process sorts that array (using the sorting algorithm of your choice) and return that array back to parent process using a separate pipe. The parent processes finally prints that sorted array on the terminal.

Graded Task 2:

Write a C program that inputs N numbers from the user and passes N/2 numbers to each of its two child processes through pipes. The child processes search their own list for the smallest number and send those numbers to their parent process through pipes. The parent process finally prints the smallest of them.

Graded Task 3:

Modify the solution of Task2 in such a way that the child processes wait for a signal (i.e. a number) from the parent process, after receiving data, to proceed. And after sending back the processed data, the child processes must wait for permission to be granted from the parent process to exit.