read() attempts to reads up to count bytes of data from the file associated with the file descriptor*fd*and places them into the buffer starting at*buf*.

RETURN VALUE

It returns the number of data bytes actually read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. An error on the call will cause it to return -1.

3W RITE() SYSTEM CALL

write- write to a file descriptor

```
#include<unistd.h>
ssize_t write (intfd ,const void *buf, size_t count)
```

write() writes up to*count*bytes from the buffer pointed*buf*to the file referred to by the file descriptor*fd*.

RETURN VALUE

It returns the number of bytes actually written. This may be less than count if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size. If the function returns 0, it means no data was written. If it returns -1, there has been an error in the write call.

4F ILE DESCRIPTORS EXAMPLE

```
char buffer[10];
//Read from standard input (by default it is keyboard)
read(0,buffer,10);
//Write to standard output (by default it is monitor))
write(1,buffer,10);
//By changing the file descriptors we can write to files
```

5R EADING AND WRITING FROM STANDARD INPUT/OUTPUT

```
#include<iostream>
#include<stdlib.h>
#include<unistd.h>
using namespacestd;
int main(){
        char buffer[10];
        cout<<"Enter string"<<endl;
        read(0,buffer,10);
        write(1,buffer,10);
        return 0;
}
```

6P IPES

Pipelines are the oldest form of UNIX inters process communication. They are used to establish one-way communication between processes that share a common ancestor. The pipe system call is used to create a pipeline:

```
intpipe (intpipefd [2])
```

The array*pipefd*is used to return two file descriptors referring to the ends of the pipe:

1. pipefd[0] is open for reading.
2. pipefd[1] is open for writing.

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.

EXPLANATION

1. Pipe system call opens a pipe, which is an area of main memory that is treated as a*virtual file*. The pipe can be used by the creating process, as well as all its child processes, for reading and writing.
2. One process can write to this*virtual file*or pipe and another related process can read from it.
3. If a process tries to read before something is written to the pipe, the process is suspended until something is written.
4. The pipe system call finds the first two available positions in theprocess'sopen file table and allocates them for the read and write ends of the pipe.
5. Implementation: A pipe can be implemented as a 10k buffer in main memory with 2 pointers, one for the FROM process and one for TO process.

7P IPES AFTER FORK

Typically, a process creates the pipeline, then uses "fork" to create a child process. Each process now has a copy of the file descriptor array; one process writes data into pipeline, while the other reads from it.

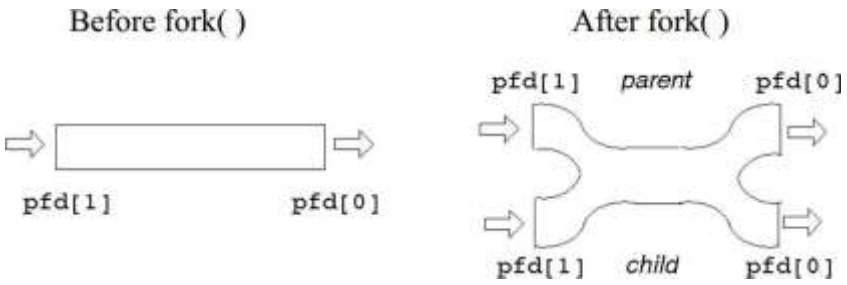

Figure 1: Structure of pipe after fork()

This gives two read ends and two write ends. Either process can write into the pipe, or either can read from it. Which process will get what is not known? For predictable behaviour, 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.

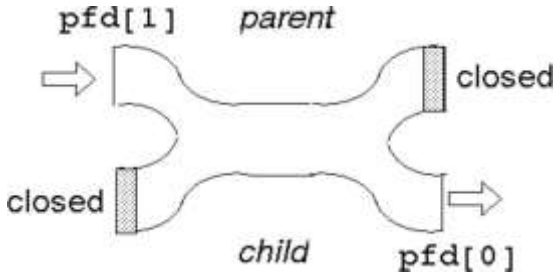

Figure 2: Closing ends of pipe

8E XAMPLES OF PIPE

EXAMPLE 01

```
# include<iostream >
#include< s t d l i b . h>
# include<unistd . h>
#include< s t r i n g . h>
# include<s t di o . h>
int main ( ) {
        intn ;
        int fd [ 2 ] ;
        char buf [ 1 0 2 5 ] ;
        char const * data = " Hello this is written to pipe " ;
        pipe ( fd ) ;
        write ( fd [ 1 ] , data , s t r l e n ( data ) ) ;
        if ( ( n=read ( fd [ 0 ] , buf , 1 0 2 4 ) ) > = 0 ) {
                buf [ n ] = 0 ;
                p r i n t f ( " Read %d bytes from pipe %s \n\n" , n , buf ) ;
```

```
        }
        else {
                perror("read");
                exit(0);
        }
        return 0;
}
```

EXAMPLE 02

```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<iostream>
#include<sys/wait.h>
using namespacestd;
int main(){
        int fd[2];
        pid_t childpid;
        charstring[] = "Hello world\n";
        charreadbuffer[80];
        intresult = pipe(fd);
        if(result<0){
                cout<<"Error while creating file";
                exit(1);
        }

        childpid = fork();
        if(childpid ==-1){
                cout<<"Error in fork "<<endl;
                exit(1);
        }

        if(childpid == 0){
                close(fd[0]);
                cout<<"Child writing to the pipe "<<endl;
                write(fd[1], string, sizeof(string));
                cout<<"Written to a file "<<endl;
                exit(0);
        }else{
                close(fd[1]);
                wait(NULL);
                cout<<"Parent reading from the pipe "<<endl;
                read(fd[0], readbuffer,sizeof(readbuffer));
                cout<<"Received string: "<<readbuffer<<endl;
                exit(0);
        }
return 0;
}
```

TASK

Task: Implement a c program that reads input from a file and counts the occurrence of each word, and prints the word count to the console in descending order of

frequency.

To accomplish this task, you can use pipes to connect several processes together:

1. The first process reads input from the file and outputs it to a pipe.
2. The second process reads input from the pipe, splits it into words, and outputs each word to a second pipe along with a count of 1.
3. The third process reads input from the second pipe, aggregates the word counts, and outputs the final word count to the console.