**East West University**

**Department of Computer Science and Engineering**

**CSE 325: LAB 04 (Handout) [Assessed Lab]**

# **Message Passing using PIPE and Understanding THREADS**

## **Lab Objective**

Familiarize students with PIPE used for message passing and THREADS for executing tasks.

## **Lab Outcome**

After completing this lab successfully, students will be able to:

1. **Understand and Implement PIPE, one of the message passing IPC resource widely used on UNIX based platforms.**

## **Lab Activities**

*1. Understanding Pipe*

A pipe has a read end and a write end. Data written to the write end of the pipe can be read from the read end.

#include<unistd.h>

int pipe(int pipedes[2]);

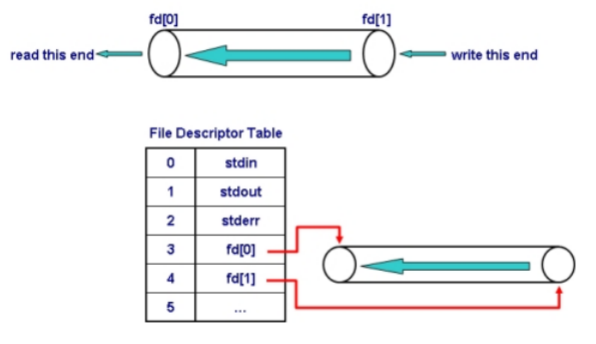


Figure 1: Pipe for IPC

*2. Creating a pipe*

A pipe can be created using a pipe api, which creates a new pipe and returns two file descriptors, one referring to the read end of the pipe, the other referring to the write end.

int pipe (int pipefd[2]);

*3. Execute the following code*

#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<string.h>  
#include<sys/types.h>  
  
int main(){  
    int pipe1fd[2];  
    int pid;  
    char msg[1024];  
    char buf;  
        // creating a pipe and checking error  
    if(pipe(pipe1fd)==-1){  
        printf("Error creating a pipe\n");  
        exit(0);  
    }  
    // creating a child process  
    pid = fork();  
    if(pid==0){ // inside a child process  
        // closing the end not used by child  
        close(pipe1fd[0]);  
        // writing to the appropriate end  
        printf("[%d] Enter message sent to the parent: \n",getpid());  
        fgets(msg,1024,stdin);  
        write(pipe1fd[1], msg, strlen(msg));  
        printf("[%d] Message sent. Child terminating\n",getpid());  
        exit(0);  
    }  
    else{ // inside the parent process  
        // closing the end not used by parent  
        close(pipe1fd[1]);  
        // waiting for the child to terminate  
        int status = -1;  
        waitpid(pid, &status, WIFEXITED(status));  
        // reading from the appropriate end (char-by-char)  
        printf("[%d] Message Received\n", getpid());  
        while(read(pipe1fd[0],&buf,1)){  
            printf("%c", buf);  
        }  
        printf("Parent terminating\n");  
          
    }  
    return 0;  
}

***Lab04\_Problem01:***

**Write a C program that creates a child process. Then child sends an integer to the parent process. Parent reads the value, and then determines whether the integer is an odd or an even number. The result must be sent to the child again who eventually reads the value (odd or even). You may use atoi() to convert a string into an integer. The following shows a sample interaction of the program.**

Inside child process [child-process-id]

Enter an integer: 11

Sending to the parent process …

Inside parent process [parent-process-id]

Received an integer: 11

Sending result to the child …

Inside child process [child-process-id]

Received result: 11 is an odd number

Child terminated

Parent terminated

*4. Understanding Threads*

A thread is a flow of execution accomplishing a given task within a process. Thread is the unit of CPU utilization. All modern CPUs are multi-threaded. Creating threads are lightweight compared to creating processes. We will use POSIX standard Pthread library to create and manipulate threads.

* Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
* A thread does not maintain a list of created threads, nor does it know the thread that created it.
* All threads within a process share the same address space.

*5. Creating and Executing Threads*

#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
  
void \*runner( void \*ptr );  
  
main()  
{  
     pthread\_t thread1, thread2;  
     char \*message1 = "Thread 1";  
     char \*message2 = "Thread 2";  
     int  iret1, iret2;  
  
    /\* Create independent threads each of which will execute function \*/  
  
     iret1 = pthread\_create( &thread1, NULL, runner, (void\*) message1);  
     iret2 = pthread\_create( &thread2, NULL, runner, (void\*) message2);  
  
     /\* Wait till threads are complete before main continues. Unless we  \*/  
     /\* wait we run the risk of executing an exit which will terminate   \*/  
     /\* the process and all threads before the threads have completed.   \*/  
  
     pthread\_join(thread1, NULL);  
     pthread\_join(thread2, NULL);   
  
     printf("Thread 1 returns: %d\n",iret1);  
     printf("Thread 2 returns: %d\n",iret2);

     exit(0);  
}  
  
void \*runner( void \*ptr )  
{  
     printf("Thread number: %lu\n", pthread\_self());  
     char \*message;  
     message = (char \*) ptr;  
     printf("%s \n", message);  
}

***Compilation: gcc thread.c –pthread -o thread***

***Lab04\_Problem02:***

**Write a C program that creates multiple threads with NULL as parameter sent to the thread execution function, runner. Each thread inside the runner function must reads an integer value, n, and then produce sum of values from 1 to n(inclusive) and prints the sum. Ensure that the main() must wait for the termination of these two threads.**

***Lab04\_Problem03:***

**This is an extension of the previous problem. The two threads now only calculate the sum but don’t print it. Instead the main() must print the sum of these two sums produced by these two threads. Modify the solution of the previous problem accordingly to solve this one.**

More information:

<http://blog.techveda.org/process-communication-using-pipes/>

<https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>

<https://computing.llnl.gov/tutorials/pthreads/>