Inter-process Communication using Pipe

Inter-Process Communication

- Inter-Process Communication(IPC) is the generic term describing how two processes may exchange information with each other.
- In general, the two processes may be running on the same machine or on different machines
- This communication may be an exchange of data for which two or more processes are cooperatively processing the data or synchronization information to help two independent, but related, processes schedule work so that they do not destructively overlap.

Inter-Process Communication Methods

- Local → Processes running on the same machine
  - Pipe
  - Signal
  - MultiProcessing (MP) in multi-core/multi-processor architecture
- Distributed → Processes running on different machines
  - Message Passing Interface (MPI)

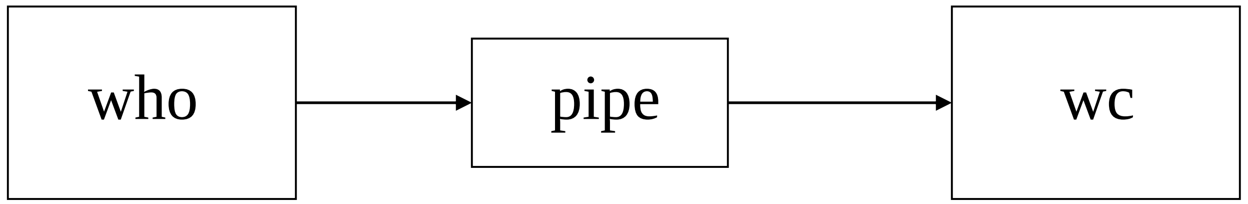
Pipes

- Pipes are an inter-process communication mechanism that allow two or more processes to send information to each other.
  - commonly used from within shells to connect the standard output of one utility to the standard input of another.
  - For example, here’s a simple shell command that determines how many users there are on the system:

    ```
    $ who | wc -l
    ```

  - The who utility generates one line of output per user. This output is then “piped” into the wc utility, which, when invoked with the “-l” option, outputs the total number of lines in its input.

Pipes



Bytes from "who" flow through the pipe to "wc"

A simple pipe

Pipes

- It’s important to realize that both the writer process and the reader process of a pipeline execute concurrently;
  - a pipe automatically buffers the output of the writer and suspends the writer if the pipe gets too full.
  - Similarly, if a pipe empties, the reader is suspended until some more output becomes available.
- All versions of UNIX support unnamed pipes, which are the kind of pipes that shells use.
- System V also supports a more powerful kind of pipe called a named pipe.

Unnamed Pipes: pipe() System Call

- An unnamed pipe is a unidirectional communications link that automatically buffers its input (the maximum size of the input varies with different versions of UNIX, but is approximately 5K) and may be created using the pipe() system call.
- Each end of a pipe has an associated file descriptor:
  - The “write” end of the pipe may be written to using write()
  - The “read” end may be read from using read()
- When a process has finished with a pipe’s file descriptor. it should close it using close()
- Note read(), write and close() are unbuffered I/O System Calls that we have studied earlier

I/O syscalls

- **`int open(char *filename, int flags, ...)`**
  - Open the file whose name is **`filename`**
    - **`flags`** often is **`O_RDONLY`**
  - Implementation (assuming **`O_RDONLY`**):
    - Find existing file on disk
    - Create file table entry
    - Set first unused file descriptor to point to file table entry
    - Return file descriptor used, -1 upon failure

I/O syscalls

- **`int close(int fd)`**
  - Close the file **`fd`**
  - Implementation:
    - Destroy file table entry referenced by element **`fd`** of file descriptor table
      - As long as no other process is pointing to it!
    - Set element **`fd`** of file descriptor table to **`NULL`**

I/O syscalls

- **`int read(int fd, void *buf, int count)`**
  - Read into **`buf`** up to **`count`** bytes from file **`fd`**
  - Return the number of bytes read; 0 indicates end-of-file

- **`int write(int fd, void *buf, int count)`**
  - Writes up to **`count`** bytes from **`buf`** to file **`fd`**
  - Return the number of bytes written; -1 indicates error

Unnamed Pipes: pipe() System Call

int pipe(int fd[2])

- pipe() creates an unnamed pipe and returns two file descriptors:
  - The descriptor associated with the “read” end of the pipe is stored in fd[0],
  - The descriptor associated with the “write” end of the pipe is stored in fd[1].
- If a process reads from a pipe whose “write” end has been closed, the “read()” call returns a value of zero, indicating the end of input.
- If a process reads from an empty pipe whose “write” end is still open, it sleeps until some input becomes available.

Unnamed Pipes: pipe() System Call

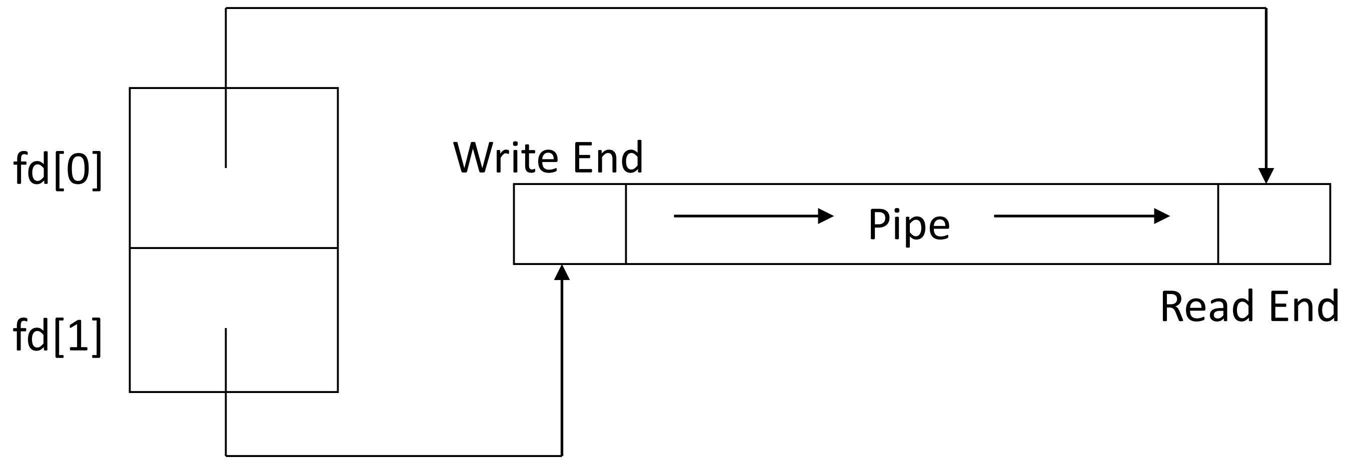
- If a process tries to read more bytes from a pipe that are present, all of the current contents are returned and read() returns the number of bytes actually read.
- if a process writes to a pipe whose “read” end has been closed, the write fails and the writer is sent a SIGPIPE signal. the default action of this signal is to terminate the receiver.
- If a process writes fewer bytes to a pipe than the pipe can hold, the write() is guaranteed to be atomic; that is, the writer process will complete its system call without being preempted by another process.
- If the kernel cannot allocate enough space for a new pipe, pipe() returns a value of -1; otherwise, it returns a value of 0.

Unnamed Pipes: pipe() System Call

- Assume that the following code was executed:

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

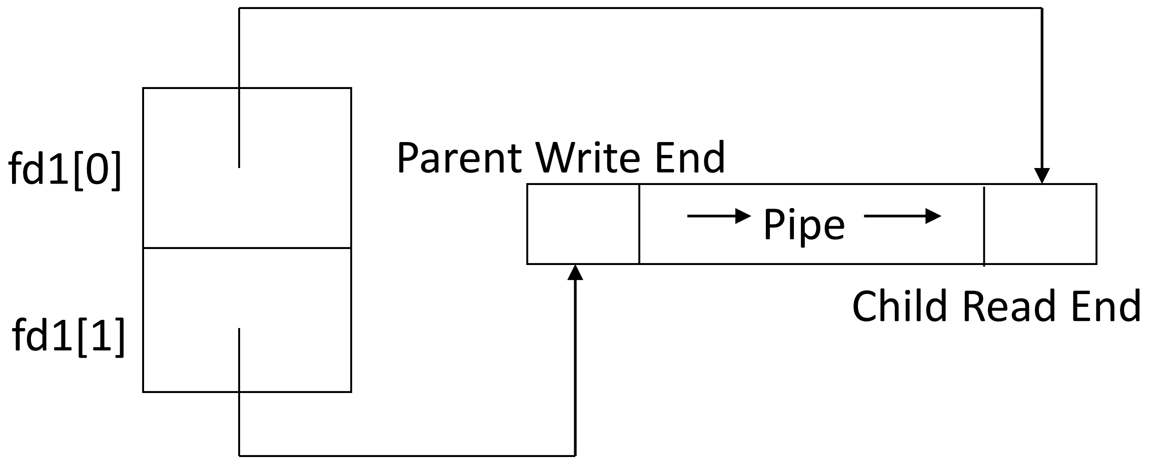
data structure as shown below will be created

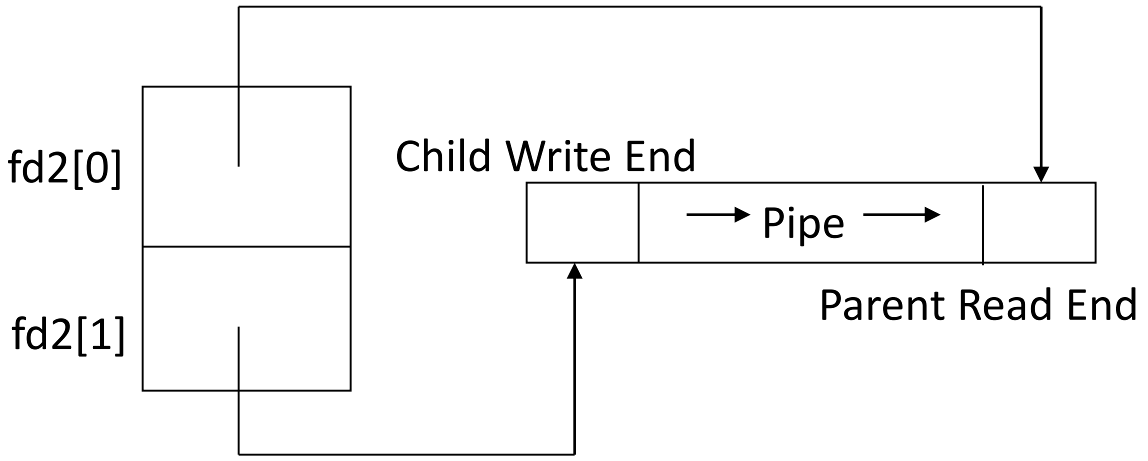


Unnamed Pipes: pipe() System Call

- Pipe are 1-way communication that means for bidirectional (2-way) communication between parent and child you need 2 pipes

```
int fd1[2], fd2[2];
pipe(fd1); pipe(fd2);
```



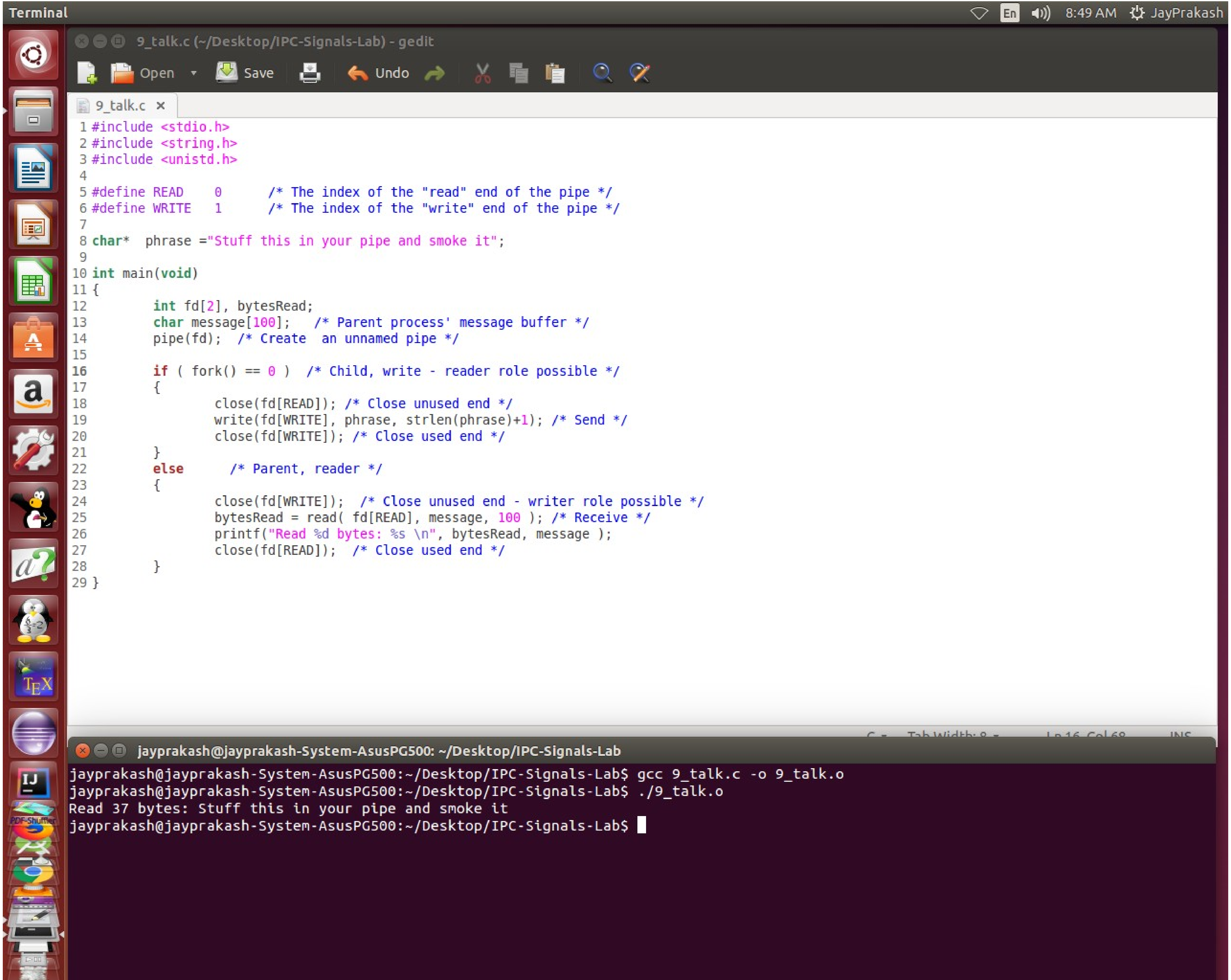


Unnamed Pipes: pipe() System Call

- Unnamed pipes are usually used for communication between a parent process and its child, with one process writing and the other process reading.
- The typical sequence of events for such a communication is as follows:
    1. The parent process creates an unnamed pipe using pipe().
    2. The parent process forks.
    3. The writer closes its “read” end of the pipe, and the designated reader closes its “write” end of the pipe.
    4. The processes communicate by using write() and read() system calls.
    5. Each process closes its active pipe descriptor when it’s finished with it.

Unnamed Pipes: pipe() System Call

- Please note : Bidirectional communication is only possible by using two pipes.
- Here’s a small program that uses a pipe to allow the child to send message to parent which is read by parent and display it: talk.c


Terminal
8:49 AM
JayPrakash
9_talk.c (~/Desktop/IPC-Signals-Lab) - gedit
Open
Save
Undo
9_talk.c
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/IPC-Signals-Lab

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

#define READ     0      /* The index of the "read" end of the pipe */
#define WRITE    1      /* The index of the "write" end of the pipe */

char*  phrase ="Stuff this in your pipe and smoke it";

int main(void)
{
        int fd[2], bytesRead;
        char message[100];   /* Parent process' message buffer */
        pipe(fd);  /* Create  an unnamed pipe */

        if ( fork() == 0 )  /* Child, write - reader role possible */
        {
                close(fd[READ]); /* Close unused end */
                write(fd[WRITE], phrase, strlen(phrase)+1); /* Send */
                close(fd[WRITE]); /* Close used end */
        }
        else      /* Parent, reader */
        {
                close(fd[WRITE]);  /* Close unused end - writer role possible */
                bytesRead = read( fd[READ], message, 100 ); /* Receive */
                printf("Read %d bytes: %s \n", bytesRead, message );
                close(fd[READ]);  /* Close used end */
        }
}
```

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ gcc 9_talk.c -o 9_talk.o
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ ./9_talk.o
Read 37 bytes: Stuff this in your pipe and smoke it
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$
```

Unnamed Pipes: pipe() System Call

- The child included the phrase’s NULL terminator as part of the message so that the parent could easily display it.

write(fd[WRITE], phrase, strlen(phrase)+1); → length+1

- When a writer process sends more than one variable-length message into a pipe, it must use a protocol to indicate to the reader the location for the end of the message.
- Methods for such indication include :
  - sending the length of a message(in bytes) before sending the message itself
  - ending a message with a special character such as a new line or a NULL

Unnamed Pipes: pipe() System Call

- UNIX shells use unnamed pipes to build pipelines, connecting the standard output of the first to the standard input of the second.
- For example we can pass two arguments to connect.c, first argument generating output on STDOUT and the other taking input which will be redirected from STDOUT.

Unnamed Pipes: pipe() System Call

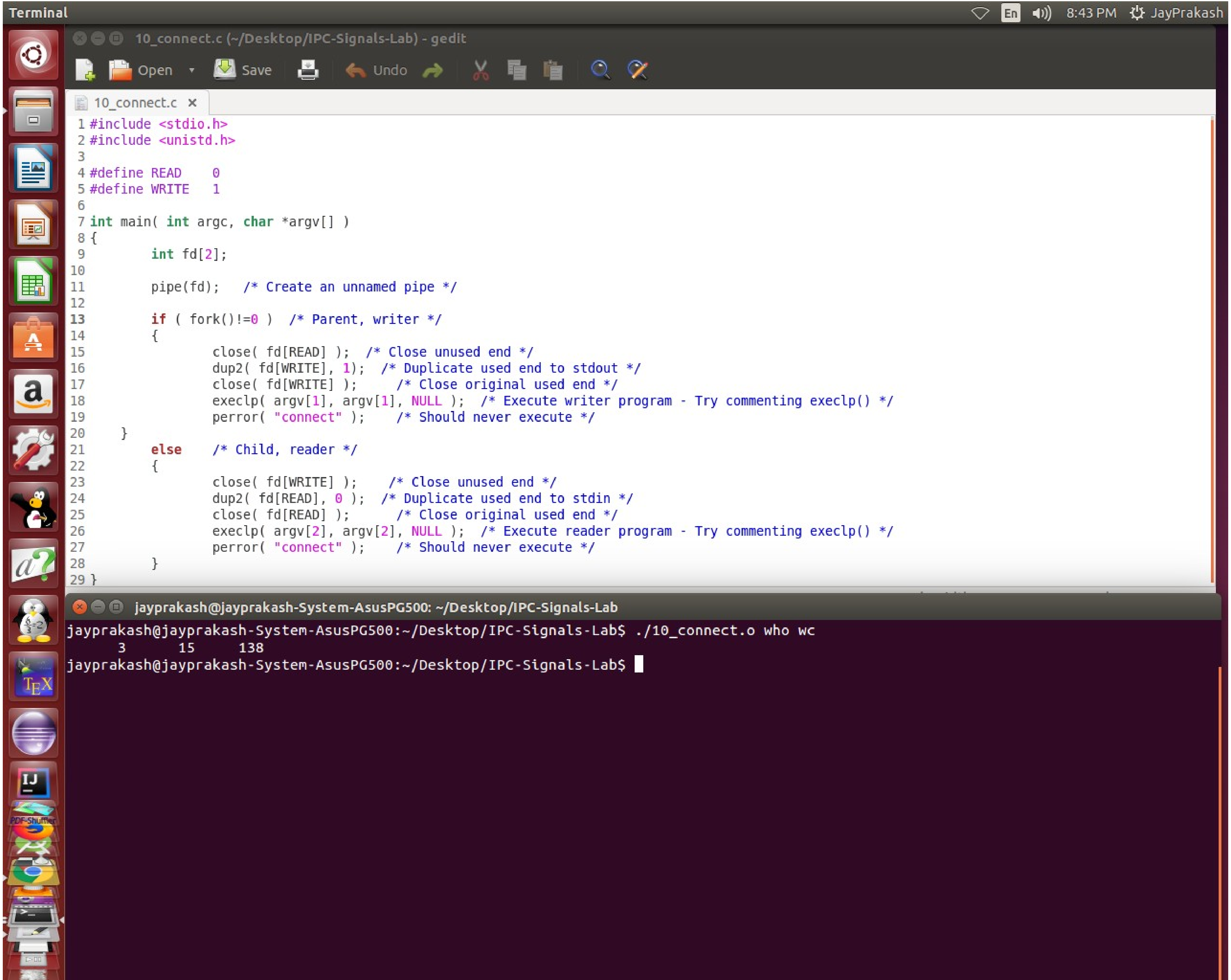
```
$ who             ---> execute “who” by itself.
gglass            ttyp0     Feb  15  18:15 (xyplex_3)

$ connect  who  wc       ---> pipe “who” through “wc”.
        1        6        57        …1 line,  6 words, 57 chars.
```

int dup2(int oldfd, int newfd)

- dup2() makes newfd be the copy of oldfd, closing newfd first if necessary, but note the following:
  - If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed.
  - If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd.

- After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably.

- They refer to the same open file description and thus share file offset and file status flags; for example, if the file offset is modified by using some another system call (eg., lseek) on one of the descriptors, the offset is also changed for the other.



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

#define READ    0
#define WRITE   1

int main( int argc, char *argv[] )
{
        int fd[2];

        pipe(fd);   /* Create an unnamed pipe */

        if ( fork()!=0 )  /* Parent, writer */
        {
                close( fd[READ] );  /* Close unused end */
                dup2( fd[WRITE], 1);  /* Duplicate used end to stdout */
                close( fd[WRITE] );     /* Close original used end */
                execlp( argv[1], argv[1], NULL );  /* Execute writer program - Try commenting execlp() */
                perror( "connect" );    /* Should never execute */
    }
        else    /* Child, reader */
        {
                close( fd[WRITE] );    /* Close unused end */
                dup2( fd[READ], 0 );  /* Duplicate used end to stdin */
                close( fd[READ] );      /* Close original used end */
                execlp( argv[2], argv[2], NULL );  /* Execute reader program - Try commenting execlp() */
                perror( "connect" );    /* Should never execute */
        }
}
```

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ ./10_connect.o who wc
      3      15     138
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$
```

Analogy

Our connect program is analogous to the pipe (|) used on our UNIX/Linux systems.

You could clearly validate the output you get from the connect program with that you get using pipes (|)

We have tested our program by passing who and wc as arguments to our connect program.

You could try comparing the output of who and wc using standard pipes (who | wc)