East West University Bangladesh
Computing Science and Engineering Department

CSE-325: Operating System, Lab 5, Inter Process Communication (IPC)

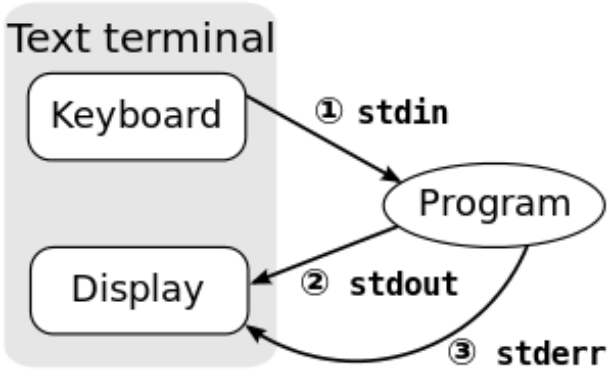
Objectives

- Familiar with Redirection and Linux Pipe.
- Work with named pipe (FIFO)

Every program we run on the command line automatically has three data streams connected to it.

- STDIN (0) - Standard input (data fed into the program)
- STDOUT (1) - Standard output (data printed by the program, defaults to the terminal)
- STDERR (2) - Standard error (for error messages, also defaults to the terminal)

Piping and redirection is the means by which we may connect these streams between programs and files to direct data in interesting and useful ways.
This lab will demonstrate piping and redirection with several examples.



Source: Wiki
http://en.wikipedia.org/wiki/Redirection_%28computing%29

What is Redirection?

Redirection is a function common to most command-line interpreters, including the various UNIX shells that can redirect standard streams to user-specified locations.
Redirection is usually implemented by placing certain characters between commands. Typically, the syntax of these characters is as follows, using **<** to redirect input, and **>** to redirect output.

Example1:
The **cat** command concatenates files and puts them all together to the standard output. By redirecting this output to a file, this file name will be created - or overwritten if it already exists.

$ **cat test1**
some words

$ **cat test2**
some other words

$ **cat test1 test2 > test3**

$ **cat test3**
some words
some other words

Example2:
Redirecting "nothing" to an existing file is equal to emptying the file:

$ **ls -l list**
-rw-rw-r-- 1 nancy nancy 117 Apr 2 18:09 list

$ **> list**

$ **ls -l list**
-rw-rw-r-- 1 nancy nancy 0 Apr 4 12:01 list

This process is called ***truncating*.**

The same redirection to an nonexistent file will create a new empty file with the given name:

$ **ls -l newlist**
ls: newlist: No such file or directory
$ **> newlist**

```
$ ls -l newlist
-rw-rw-r--  1 nancy   nancy          0 Apr  4 12:05 newlist
```

We can get the new data to be appended to the existing file by using the double greater than operator (>>).
$ ls >> myoutput
$ cat myoutput

If we use the less than operator (<) then we can send data the other way. We will read data from the file and feed it into the program via it's STDIN stream. [wc is for word count]
$ wc -l < myoutput

We may easily combine the two forms of redirection we have seen so far into a single command as seen in the example below.
$ wc -l < barry.txt > myoutput
$ cat myoutput

Here's a very simple filter program, **upper.c,** that reads input and converts it to uppercase. [If conio.h is not found then use scanf and printf function]

```
1.  #include <stdio.h>
2.  #include <conio.h>

3.  int main()
4.  {
5.      int ch;
6.     while((ch = getchar()) != EOF)
7.      {
8.            putchar(toupper(ch));
9.      }
10.     exit(0);
11. }
```

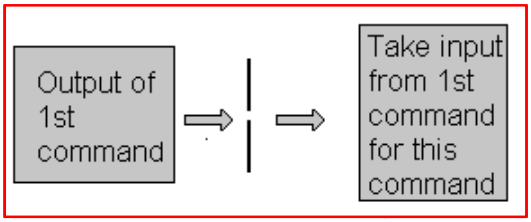
$ ./upper
hello THERE
HELLO THERE
^D
$

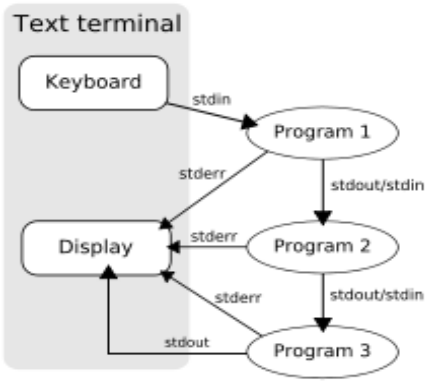
Task1:

- Use the above program (upper.c) and convert a file (myfile.txt-holds mixed case characters) to uppercase by using the shell redirection.
- Create another program (lower.c) and convert myfile.txt (holds mixed case characters) to lowercase and put the output into another file named myfile2.txt by combining the two forms of redirection.

What is Pipe?

A pipe is a way to connect the output of one program to the input of another program. We can do so using pipe (|) operator.





A pipeline of three programs run on a text terminal

Pipe Defined as:"A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands (Multiple commands) from same command line."

Syntax:
command1 | command2

Example (pipe):

$ ls-l | more

Output of ls-l command displayed one screen at a time.

Example (Redirection-temporal file):

$ ls -l > temp
$ more temp
$ rm temp

Output of the ls –l command displayed one screen at a time

Filter:

When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output, it is referred to as a filter.

$ls -l | grep "lab06"

Task2:

- Write a file named **fruit_list** with vi editor. List some fruit names (with mango and put some fruit names more than once) in your file.
- Use the following commands and write their tasks in details.

$sort fruit_list|uniq>sorted_fruit_list
$sort fruit_list|uniq|grep "man"

Process Pipes

Perhaps the simplest way of passing data between two programs is with the popen and pclose functions. These have the prototypes:

#include <stdio.h>

FILE *popen(const char *command, const char *open_mode);

The popen function allows a program to invoke another program as a new process and either pass data to or receive data from it. The command string is the name of the program to run, together with any parameters. open_mode must be either "r" or "w".

int pclose(FILE *stream_to_close);

When the process started with popen has finished, we can close the file stream associated with it using pclose. The pclose call will only return once the process started with popen finishes. If it's still running when pclose is called, the pclose call will wait for the process to finish.

Reading Output from an External Program

```
1.  #include <unistd.h>
2.  #include <stdlib.h>
3.  #include <stdio.h>
4.  #include <string.h>
5.
6.  /* #define BUFSIZ 10 define done inside stdio.h by macro, gcc uses 1024 */
7.
8.  int main()
9.  {
10.         FILE *read_fp;
11.         char buffer[BUFSIZ + 1];
12.         int chars_read;
13.
14.        memset(buffer, '\0', sizeof(buffer));
15.        read_fp = popen("uname −a", "r");
16.        if (read_fp != NULL) {
17.                chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
18.                if (chars_read > 0) {
19.                        printf("Output was:−\n%s\n", buffer);
20.                }
21.               pclose(read_fp);
22.               exit(EXIT_SUCCESS);
23.        }
24.       exit(EXIT_FAILURE);
25. }
```

How It Works:
The program uses the popen call to invoke the uname command with the −a parameter. It then uses the returned file stream to read data up to BUFSIZ characters (this being a #define from stdio.h) and then prints it out so it appears on the screen. Since we've captured the output of uname inside a program, it's available for processing.

Sending Output to an External Program

```
1.  #include <unistd.h>
2.  #include <stdlib.h>
3.  #include <stdio.h>
4.  #include <string.h>
5.
6.  int main()
7.  {
8.          FILE *write_fp;
9.          char buffer[BUFSIZ + 1];
10.         sprintf(buffer, "Once upon a time, there was...\n");
11.         write_fp = popen("od −c", "w");
12.         if (write_fp != NULL) {
13.                 fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
14.                 pclose(write_fp);
15.                 exit(EXIT_SUCCESS);
16.          }
17.         exit(EXIT_FAILURE);
18. }
```

How It Works:
The program uses popen with the parameter w to start the od −c command, so that it can send data to it. It then sends a string which the od −c command receives, processes and prints the result of its processing on its standard output. From the command line, we can get the same output with the command:
$ echo "Once upon a time, there was..." | od –c

The Pipe Call

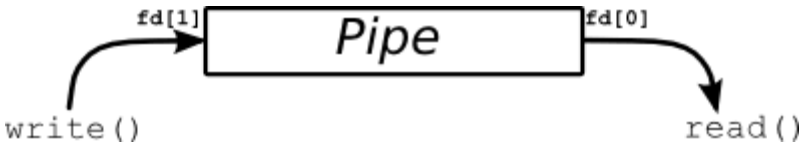
We've seen the high−level popen function, but we'll now move on to look at the lower−level pipe function. This provides a means of passing data between two programs, **without the overhead of invoking a shell** to interpret the requested command. It also gives us more control over the reading and writing of data.

The pipe function has the prototype:

#include <unistd.h>
int pipe(int file_descriptor[2]);

pipe is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns −1 and sets errno to indicate the reason for failure. Errors defined in the Linux man pages are: **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.

The two file descriptors returned are connected in a special way. Any data written to file_descriptor[1] can be read back from file_descriptor[0]. The data is processed in a first in, first out basis, usually abbreviated to FIFO. This means that if you write the bytes 1, 2, 3 to file_descriptor[1], reading from file_descriptor[0] will produce 1, 2, 3. **Important:** It's important to realize that these are file descriptors, not file streams, so we must use the lower−level **read** and **write** calls to access the data, rather than fread and fwrite.



How a pipe is organized.

For example, *stdin* is file descriptor "0", *stdout* is "1", and *stderr* is "2". Likewise, any files you open using **fopen()** get their own file descriptor, although this detail is hidden from you. (This file descriptor can be retrived from the FILE* by using the **fileno()** macro from *stdio.h*.)

Basically, a call to the **pipe()** function returns a pair of file descriptors. One of these descriptors is connected to the **write** end of the pipe, and the other is connected to the **read** end. Anything can be written to the pipe, and read from the other end in the order it came in. On many systems, pipes will fill up after you write about 10K to them without reading anything out.

Here's a program that uses pipe to create a pipe.

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

int main(void)
{
	int pfds[2];
	char buf[30];

	if (pipe(pfds) == -1) {
		perror("pipe");
		exit(1);
	}

	printf("writing to file descriptor #%d\n", pfds[1]);
	write(pfds[1], "test", 5);
	printf("reading from file descriptor #%d\n", pfds[0]);
	read(pfds[0], buf, 5);
	printf("read \"%s\"\n", buf);

	return 0;
}
```

`fork() and pipe()`

From the above example, it's pretty hard to see how these would even be useful. Let's put a **fork()** in the mix and see what happens. First, we'll have the parent make a pipe. Secondly, we'll **fork()**. Now, the **fork()** man page tells us that the child will receive a copy of all the parent's file descriptors, and this includes a copy of the pipe's file descriptors. Thus, the child will be able to send stuff to the write-end of the pipe, and the parent will get it off the read-end.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
        int pfds[2];
        char buf[30];

        pipe(pfds);

        if (!fork()) {
                printf(" CHILD: writing to the pipe\n");
                write(pfds[1], "test", 5);
                printf(" CHILD: exiting\n");
                exit(0);
        } else {
                printf("PARENT: reading from pipe\n");
                read(pfds[0], buf, 5);
                printf("PARENT: read \"%s\"\n", buf);
                wait(NULL);
        }
        return 0;
}
```

The search for Pipe as we know it

Here is an example of using **pipe()** in a more familiar situation. The challenge: implement "**ls | wc -l**" in C.

This requires usage of a couple more functions you may never have heard of: **exec()** and **dup()**. The **exec()** family of functions replaces the currently running process with whichever one is passed to **exec()**. This is the function that we will use to run **ls** and **wc -l**.

dup() takes an open file descriptor and makes a clone (a duplicate) of it. This is how we will connect the standard output of the **ls** to the standard input of **wc**. See, stdout of **ls** flows into the pipe, and the stdin of **wc** flows in from the pipe. The pipe fits right there in the middle!

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

int main(void)
{
        int pfds[2];

        pipe(pfds);

        if (!fork()) {
                close(1);       /* close normal stdout */
                dup(pfds[1]);   /* make stdout same as pfds[1] */
                close(pfds[0]); /* we don't need this */
                execlp("ls", "ls", NULL);
        } else {
                close(0);       /* close normal stdin */
```

```
                dup(pfds[0]);   /* make stdin same as pfds[0] */
                close(pfds[1]); /* we don't need this */
                execlp("wc", "wc", "-l", NULL);
        }

        return 0;
}
```

Tasks:

1. Modify the above program to make the child does the `wc -l`, and the parent does the `ls`
2. At the end of `pipe` manual page (`man 2 pipe`), there is an example program. Compile it, run it, understand it, and then, modify the program, let parent do read, and child do write.

Named Pipes: FIFOs

So far, we have only been able to pass data between programs that are related, i.e. programs that have been started from a common ancestor process. Often, this isn't very convenient, as we would like unrelated processes to be able to exchange data. We do this with FIFOs, often referred to as named pipes.

A named pipe is a special type of file (remember everything in UNIX is a file!) that exists as a name in the file system, but behaves like the unnamed pipes that we've met already. We can create named pipes from the command line and from within a program. Historically, the command line program for creating them was

mknod: $ mknod filename p

However, the mknod command is not in the X/Open command list, so may not be available on all UNIX systems. The preferred command line method is to use:

$ mkfifo filename

From inside a program, we can use two different calls. These are:

```
#include <sys/types.h>
#include<sys/stat.h>
int mkfifo(const char *filename, mode_t mode);
mknod("myfifo", S_IFIFO | 0644 , 0);
```

In the above example, the FIFO file will be called "*myfifo*". The second argument is the creation mode, which is used to tell mknod() to make a FIFO (the S_IFIFO part of the OR) and sets access permissions to that file (octal 644, or rw-r--r--) which can also be set by ORing together macros from *sys/stat.h*. This permission is just like the one you'd set using the **chmod** command. Finally, a device number is passed. This is ignored when creating a FIFO, so you can put anything you want in there.
(An aside: a FIFO can also be created from the command line using the Unix **mknod** command.)

```
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);
```

Like the mknod command, you can use the mknod function for making many special types of file. Using a dev_t value of 0 and ORing the file access mode with S_IFIFO is the only portable use of this function which creates a named pipe. We'll use the simpler mkfifo function in our examples.

Similar Problem like Producers and Consumers

Once the FIFO has been created, a process can start up and open it for reading or writing using the standard **open()** system call.

Two programs are given – one will send data through a FIFO named ***speak.c(lab05_4.c)***, and the other is called ***tick.c(lab05_5.c),*** as it will remove data out of the FIFO.

Tasks:

1. When you run the example programs (speak and tick), there should be a new file named **american_maid** appear in your working directory ($PWD). What will happen if you delete this FIFO file while the two programs running? Why?
2. Modify the example programs to use **mkfifo** instead of **mknod**.
3. Extend the example programs, and make it have three (3) writers.