Inter-Process Communication - Pipes

Objective

- Introduction to process communication
- Using pipes as a tool for IPC (Inter-Process Communication)
- Redirecting standard input/output

Description

Many processes can live their lives without having to interact with other processes (independent processes). For many reasons including convenience, computation speedup and information sharing, some processes choose to interact with each other. Some will use/consume data that is produced by other processes (cooperating processes). A programmer who expects her/his program to communicate with other programs can build their own communication tools. They should provide a medium of communication and provide the tools to guarantee proper access to the data:

- Buffer must be created with controlled access (read, write or both)
- Data must be protected from errors caused by difference in the speed between the producer and the consumer processes:
  - Data protected from being overwritten before it's read
  - Data protected from being read multiple times

Modern operating systems provide the programmer with tools to facilitate correct communication between processes. These tools abstract a buffer and provide the proper methods to access the data. Some of these tools are designed to work with communicating processes across computer systems and others are intended to work on the same machine. Communication between processes on a single machine is known as Inter-Process Communication (IPC) and some of the tools used in IPC are: pipes, shared memory and message queues.

A pipe is a Linux kernel object that can be used to establish unidirectional communication between processes. A pipe has a limited buffer and is described/accessed through 2 file descriptors: one refers to the read end (known as the downstream) of the pipe and the other to the write end (known as the upstream) of the pipe. Data written to the upstream end of the pipe is buffered by the kernel until it is read from the downstream end of the pipe. The pipe's default behavior is to block a writer process if the buffer is full and to block a reader process if the buffer is empty. This behavior can be overwritten when a pipe is created using a pipe2 with a flag of O_NONBLOCK. Pipes are commonly used to redirect input and output from the default stdin (keyboard) and stdout (screen) to other outlets described with file descriptors. This helps direct the output of any command to a file or a device. It also helps system admins perform interesting tasks by chaining multiple commands where the output from one command is directed as input to the next one.

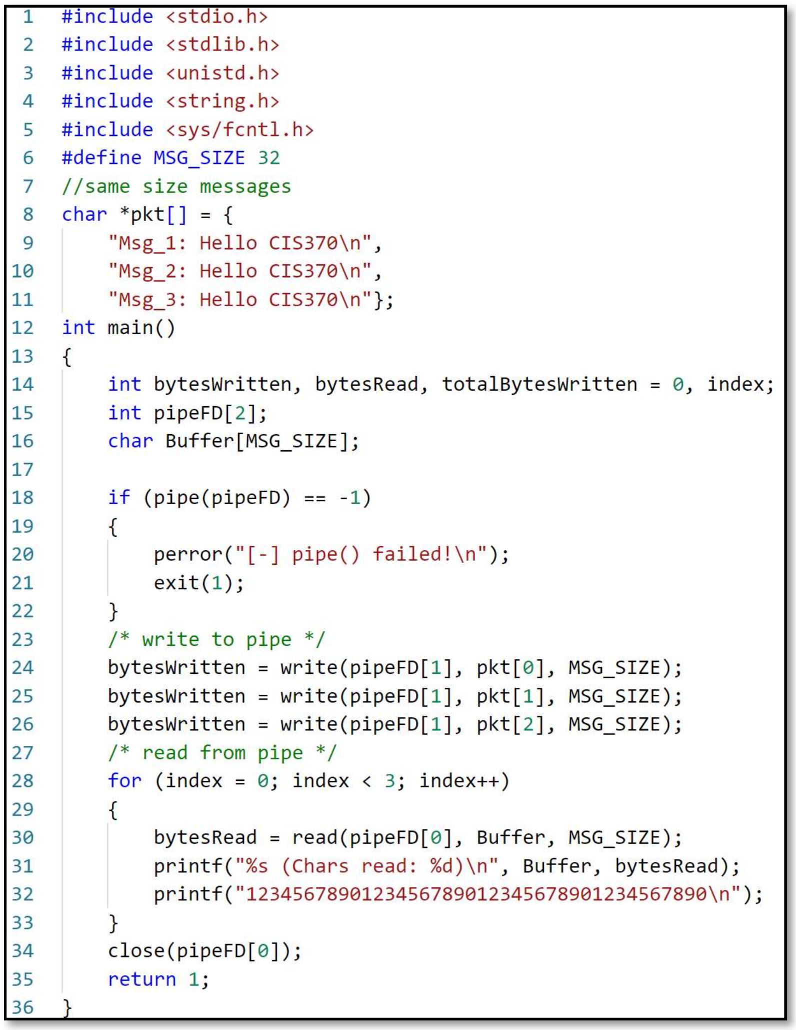
http://man7.org/linux/man-pages/man2/pipe.2.html
- `int pipe(int pipefd[2]);`
- `int pipe2(int pipefd[2], int flags);`

http://man7.org/linux/man-pages/man2/dup.2.html
- `int dup(int oldfd);`
- `int dup2(int oldfd, int newfd);`
- `int dup3(int oldfd, int newfd, int flags);`

Submission:

- *Submit the answers to the questions below in pdf format*
- *One compressed file that contains all C and pdf files*
- *Make sure to do the following:*
    - *Check the success of every system call you use*
    - *Make sure a parent process waits for all its children*
    - *When using pipes, make sure to close the pipe-side the process will not use*

1. **Test Communication Using Pipes:**
   Download the skeleton code (P1_Code.c), write its missing details based on the code below and run it, then answer the following questions:

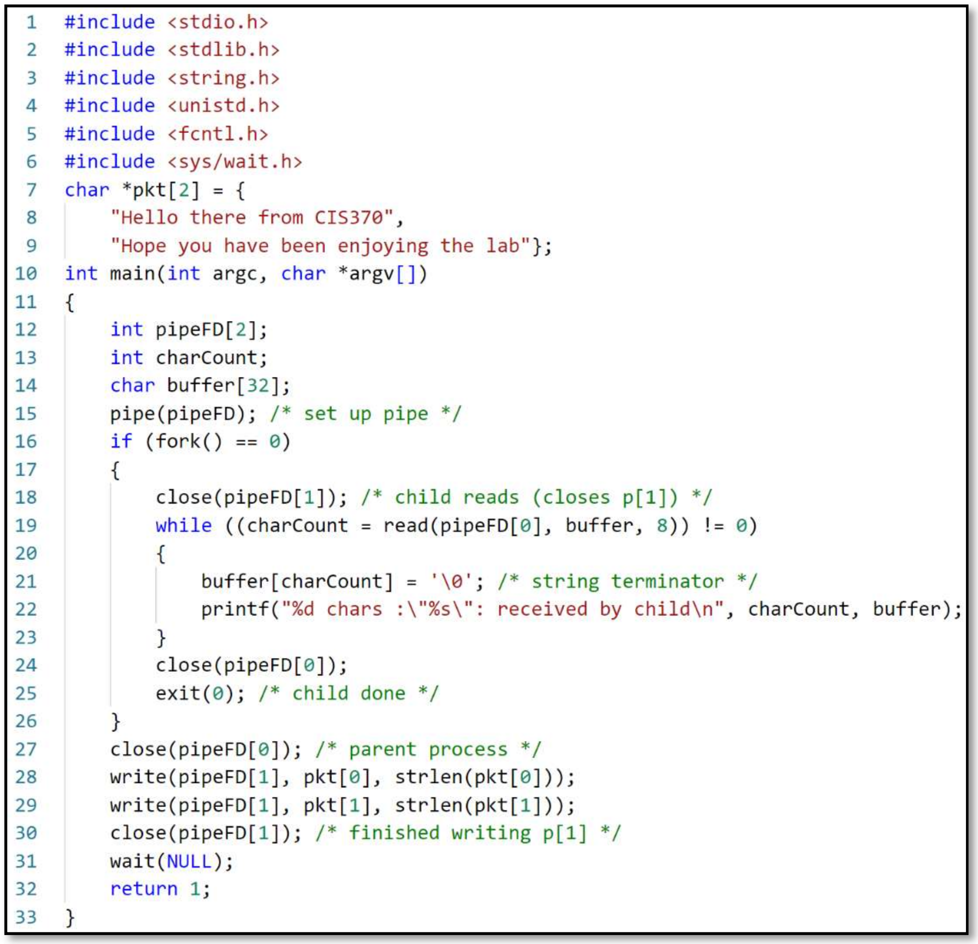


```c
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <unistd.h>
 4  #include <string.h>
 5  #include <sys/fcntl.h>
 6  #define MSG_SIZE 32
 7  //same size messages
 8  char *pkt[] = {
 9      "Msg_1: Hello CIS370\n",
10      "Msg_2: Hello CIS370\n",
11      "Msg_3: Hello CIS370\n"};
12  int main()
13  {
14      int bytesWritten, bytesRead, totalBytesWritten = 0, index;
15      int pipeFD[2];
16      char Buffer[MSG_SIZE];
17
18      if (pipe(pipeFD) == -1)
19      {
20          perror("[-] pipe() failed!\n");
21          exit(1);
22      }
23      /* write to pipe */
24      bytesWritten = write(pipeFD[1], pkt[0], MSG_SIZE);
25      bytesWritten = write(pipeFD[1], pkt[1], MSG_SIZE);
26      bytesWritten = write(pipeFD[1], pkt[2], MSG_SIZE);
27      /* read from pipe */
28      for (index = 0; index < 3; index++)
29      {
30          bytesRead = read(pipeFD[0], Buffer, MSG_SIZE);
31          printf("%s (Chars read: %d)\n", Buffer, bytesRead);
32          printf("1234567890123456789012345678901234567890\n");
33      }
34      close(pipeFD[0]);
35      return 1;
36  }
```

   a. How many bytes are written to the upstream end of the pipe?
   b. How many bytes are read from the downstream end of the pipe?
   c. Change the code in line 30 to read 10 bytes instead of `MSG_SIZE`. Show the output of the code and explain what is being printed.
   d. Change the code in line 30 to read 50 bytes instead of `MSG_SIZE`. Show the output of the code and explain the behavior of the program.

Note: All labs are due before next week's lab starts

2. **More Tests of Pipes:**
   Download the skeleton code (P2_Code.c), write its missing details based on the code below and run it, then answer the following questions:



```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <unistd.h>
5   #include <fcntl.h>
6   #include <sys/wait.h>
7   char *pkt[2] = {
8       "Hello there from CIS370",
9       "Hope you have been enjoying the lab"};
10  int main(int argc, char *argv[])
11  {
12      int pipeFD[2];
13      int charCount;
14      char buffer[32];
15      pipe(pipeFD); /* set up pipe */
16      if (fork() == 0)
17      {
18          close(pipeFD[1]); /* child reads (closes p[1]) */
19          while ((charCount = read(pipeFD[0], buffer, 8)) != 0)
20          {
21              buffer[charCount] = '\0'; /* string terminator */
22              printf("%d chars :\"%s\": received by child\n", charCount, buffer);
23          }
24          close(pipeFD[0]);
25          exit(0); /* child done */
26      }
27      close(pipeFD[0]); /* parent process */
28      write(pipeFD[1], pkt[0], strlen(pkt[0]));
29      write(pipeFD[1], pkt[1], strlen(pkt[1]));
30      close(pipeFD[1]); /* finished writing p[1] */
31      wait(NULL);
32      return 1;
33  }
```

a. Comment line 30 and run the code again. How does the code behave? Explain the behavior?
b. Uncomment line 30 and move line 31 after line 27 then run the code again. How does the code behave? Explain the behavior?

3. **Copy a File Using Pipes:**
   Implement a command (`lastnameCP`) that copies one file (source) to another (destination) using pipes.
   Hint: A possible solution:
   - Parent process creates a pipe then forks a child process:
     - The parent process opens and reads the input file, that was passed as an argument, and writes the data to the upstream end of a pipe.
     - The child process opens the output file, reads data from the pipe and writes it to the output file.
   - When done, the parent process closes the upstream and the child closes downstream.

Note: All labs are due before next week's lab starts

4. Switching *`stdout`* between the 'screen' and output file:

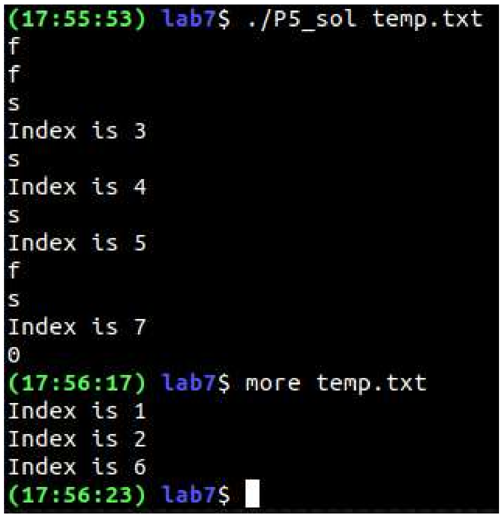
Write a C program that keeps on printing a default message either on the screen or into a file. Your program can only have a single write statement:

```
write(1, outBuffer, strlen(outBuffer)); //where char outbuffer[128];
```

Your program should read a character from the user and alter where stdout points to (the screen or the output file). The program should respond to the user input as follows:

- '0' (zero): terminate the program and exit.
- 'f': print the output to an output file specified by the user (when program was run).
- 's': print the output to the screen.

The figure below shows a run of the program where the data is written either to the screen or to 'temp.txt'



```
(17:55:53) lab7$ ./P5_sol temp.txt
f
f
s
Index is 3
s
Index is 4
s
Index is 5
f
s
Index is 7
0
(17:56:17) lab7$ more temp.txt
Index is 1
Index is 2
Index is 6
(17:56:23) lab7$
```

Hint:

- Open the file entered by the user.
- Use `dup(…)` to capture the what stdout is pointing at.
- Then use `dup2(…)` to switch between the screen and the file.