Named Pipes (Optional)	LABORATORY	SEVEN
OBJECTIVES		

1. Alternative Communication in Unix/Linux
2. Named Pipes
3. Sharing Named Pipes

Alternative Communication in Unix/Linux

Processes can simply communicate via a *temporary file*: a process writes to the file and another process reads from the file. Using I/O redirection, one can do this as follows, using a temporary file.

```
parent > temp.txt; temp.txt < child
```

The problem with this approach is that the parent process must completely finish its data writing before the child process can start running. Could we relax this by allowing the parent and child to run *concurrently* as follows?

```
parent > temp.txt &
temp.txt < child &
```

Normally, when we open a file for writing, we would not expect that the same file is being read by another process at the same time. For our particular application here, this needs to be allowed. Assume that we are allowed to do this, i.e., concurrent access to the same file (with someone writing and someone reading). Now, just imagine what would happen if the child process is reading faster than the parent process? We do need a form of *blocking receive*, *non-blocking send* mechanism, in the context of message passing.

One simple way to ensure that the child process always waits for the parent process to write data when all current data have been used up by the child is by means of a ***pipe*** in the *command line*. Automatically, the child process will wait for the parent if there is no more data at any moment to read. A *single unidirectional link* exists between a pair of processes, with a bounded buffer implemented by Linux. There is no name for this pipe.

```
parent | child
```

This is an improved form of "temporary" file communication adopted in Unix/Linux and is encapsulated under pipe communication mechanism. If the child is passing data to the parent, you could write in a similar way like this.

```
child | parent
```

With this arrangement, it is *impossible* to have parent sending data to child and child sending data to parent at the same time. As a result, we need to adopt the `pipe()` mechanism, by establishing *two pipes* inside the program for parent and child processes to communicate. There is *no name* for the pipe. Once the pipe is created, the child will be able to "*inherit*" the pipe information (the pair of *file descriptors*) from the parent via the `fork()` mechanism, since a Unix/Linux child is an *exact copy* of the parent. We call this an ***unnamed pipe***.

There is a limitation that the unnamed pipe must be created *before* the child process is created. Suppose that after two processes start running, they decide that they need to *talk*. How can this be done? In Unix/Linux, there is a mechanism that shares the characteristics of a pipe, as well as a temporary file. This is called the ***named pipe***. A *named pipe* allows the communication to be established between two *existing processes*. For example, when a communication pipe is needed *on demand* some time *after* the child is created by the parent, a *named pipe* can be created.

Named Pipes

A ***named pipe*** is also known as a ***FIFO***. It is a combination of a *temporary file* and a *pipe*, i.e., it is a temporary file used in a special manner, or a pipe with a name to be referred to. It is used to establish a *one-way flow* of data like an unnamed pipe and is identified by its access point, which is a *file* on the file system. Thus, a named pipe has the *pathname* of a file associated with it. Two *unrelated* processes can *open* the file associated with the named pipe and begin communication. Note that unnamed pipes only exist when the processes exist; they will be removed when the processes terminate. On the other hand, named pipes exist even after the processes terminate, unless they are explicitly deleted by the processes.

A named pipe can be created in two ways. It can be created at the *command line* or from within a *program*. Type the *command* **`mkfifo`** ***mypipe*** at the shell and you will create a named pipe called ***mypipe***. However, for the current file system setting in the department, you would not be able to create a named pipe on Unix/Linux under your own directory, though it would probably work in Unix/Linux systems outside. So you should create the named pipe at ***/tmp***, i.e., **`cd`** ***/tmp*** and even better create a subdirectory for yourself under ***/tmp***, and then create your pipe inside that subdirectory ***/tmp/12345678d***. You can use **`ls -l`** to check your pipe and you will see the type **`p`** for a pipe in the directory information (compare this with the type **`d`** for a directory and a **`-`** for a normal file).

The second way is to create a named pipe inside a program. To do this, use the *system call* **`mkfifo()`**, with signature **`int mkfifo(const char *path, mode_t access_mode)`**, similar to the use of the system call **`open()`** and the parameter **`access_mode`** is similar to the one used in **`open()`** (for creating a new file in **Lab 5** with **`O_CREAT`** flag) or **`creat()`**. To remove a named pipe at the shell, use the *command* **`rm`** ***mypipe***. To remove a named piped pipe inside a program, use the *system call* **`unlink()`**, with signature **`int unlink(const char *path)`**.

After a named pipe is created, it can be *opened* for *reading* or *writing* just like a file. It can be opened by using the **`open()`** system call. You then use **`write()`** or **`read()`** system calls to send/receive data. Since a named pipe is like a file, it can also be deleted by the program. Also because it is actually a file, it *would not* be read and written at the same time (as determined by the ***mode*** when you ***open*** this named pipe). As a result, you *do not need* to close the excessive ends of the pipe like *unnamed pipes*, since when you open a *named pipe*, you only have one access point. Note that though there is no rule to bar you from opening the named pipe using **`O_RDWR`** mode (both read and write), you will not gain any benefit from it, as it is equivalent to letting both ends of your unnamed pipe remain open, a situation that we will always try to avoid in pipe programming. It is virtually impossible to use such kind of named pipe setting (opening with **`O_RDWR`** mode) for bi-directional communication.

By default, the read operation to a named pipe is *blocking*, like an unnamed pipe. If a process reads from a named pipe and if the pipe does not contain any data, the reading process will be *blocked* (i.e., forced to wait). This resolves the problem that the reading process executes faster than the writing process. If a process writes to a named pipe with no reader, the writing process will be blocked until another process opens the named pipe for reading. If a reader exists, the writer can continue to write in a non-blocking manner. As a result, the *non-blocking send, blocking receive* mechanism in message passing is provided. Note that treating a named pipe as a file, you might use the **`fopen()`** library call to open and **`fprintf()/fputs()`** and **`fscanf()/fgets()`** library calls to write and read. However, this is not recommended for beginners, since the buffering effect of streams makes it much more difficult to debug.

The following sample program is a rewritten version of **lab6C.c** using *named pipe*. Note that the use of mode **0600** (*owner right = read / write*) when the named pipe is created indicates that only the user can access the named pipe. Processes created by other uses cannot read from it. Parent will accept input from the keyboard as in **lab6C.c** and send the encrypted strings to child. Recall that keyboard input is terminated by typing <**Ctrl-D**>. Compare this program **lab7A.c** with program **lab6C.c**.

```
// lab 7A
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
  char pipename[] = "/tmp/mypipe12345678d"; // pathname of the named pipe
  char mapl[] = "qwertyuiopasdfghjklzxcvbnm"; // for encoding letter
  char mapd[] = "1357924680"; // for encoding digit
  char  buf[80];
  int   i, n, childid, fd;

  // create the named pipe
  if (mkfifo(pipename,0600) < 0) {
      printf("Pipe creation error\n");
      exit(1);
  }

  childid = fork();
  if (childid < 0) {
      printf("Fork failed\n");
      exit(1);
  } else if (childid == 0) { // child
     // open pipe for reading
     if ((fd = open(pipename,O_RDONLY)) < 0) {
         printf("Pipe open error\n");
         exit(1);
     }
     while ((n = read(fd,buf,80)) > 0) { // read from pipe
            buf[n] = 0;
            printf("<child> message [%s] of size %d bytes received\n",buf,n);
     }
     close(fd); // close the pipe
     printf("<child> I have completed!\n");
  } else { // parent
    // open pipe for writing
     if ((fd = open(pipename,O_WRONLY)) < 0) {
         printf("Pipe open error\n");
         exit(1);
     }
     while (1) {
            printf("<parent> please enter a message\n");
            n = read(STDIN_FILENO,buf,80); // read a line
            if (n <= 0) break; // EOF or error
            buf[--n] = 0;
            printf("<parent> message [%s] is of length %d\n",buf,n);
            for (i = 0; i < n; i++) // encrypt
                 if (buf[i] >= 'a' && buf[i] <= 'z')
                     buf[i] = mapl[buf[i]-'a'];
                 else if (buf[i] >= 'A' && buf[i] <= 'Z')
                     buf[i] = mapl[buf[i]-'A']-('a'-'A');
                 else if (buf[i] >= '0' && buf[i] <= '9')
                     buf[i] = mapd[buf[i]-'0'];
            printf("<parent> sending encrypted message [%s] to child\n",buf);
            write(fd,buf,n); // send the encrypted string
     }
     close(fd); // close the pipe
     wait(NULL);
     printf("<parent> I have completed!\n");
     unlink(pipename); // remove the pipe
  }
  exit(0);
}
```

To support two independent processes communicating with the named pipe, we could make the following arrangement. Let us call the process writing data the *sender process*, and the process reading data the *receiver process*. Here, we assign the sender process with the responsibility for *creating* the named pipe. A direct consequence of this arrangement is that the sender program must be run before the receiver program. You could of course request the receiver process to create the pipe and let it wait for the sender process to become active.

Compile the two programs for **lab7B*.c**. Run the *sender program first* and *then run the receiver program*, preferably using another window to make it easy to see. Observe the outputs generated. Since there is a constraint that the sender program must be run ***before*** the receiver program, we can now try to run the programs in the incorrect order and see whether there is any synchronization related problem. Observe the problem occurred. In order to clean up the mess left over, you need to **delete** the named pipe created by the sender program explicitly at command prompt with **`rm`** ***/tmp/mypipe12345678d*** before you can start over again in running the programs in correct order.

```
// lab 7B sender
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
  char pipename[] = "/tmp/mypipe12345678d"; // pathname of the named pipe
  char mapl[] = "qwertyuiopasdfghjklzxcvbnm"; // for encoding letter
  char mapd[] = "1357924680"; // for encoding digit
  char buf[80];
  int  i, n, fd;

  // create the named pipe
  if (mkfifo(pipename,0600) < 0) {
      printf("Pipe creation error\n");
      exit(1);
  }
  // open pipe for writing
  if ((fd = open(pipename,O_WRONLY)) < 0) {
      printf("Pipe open error\n");
      exit(1);
  }
  while (1) {
        printf("<sender> please enter a message\n");
        n = read(STDIN_FILENO,buf,80); // read a line from keyboard
        if (n <= 0) break; // EOF or error
        buf[--n] = 0;
        printf("<sender> message [%s] is of length %d\n",buf,n);
        for (i = 0; i < n; i++) // encrypt
            if (buf[i] >= 'a' && buf[i] <= 'z')
                buf[i] = mapl[buf[i]-'a'];
            else if (buf[i] >= 'A' && buf[i] <= 'Z')
                buf[i] = mapl[buf[i]-'A']-('a'-'A');
            else if (buf[i] >= '0' && buf[i] <= '9')
                buf[i] = mapd[buf[i]-'0'];
        printf("<sender> sending encrypted message [%s] to receiver\n",buf);
        write(fd,buf,n); // send the encrypted string via pipe
  }
  close(fd); // close the pipe
  printf("<sender> I have completed!\n");
  unlink(pipename); // remove the pipe
  exit(0);
}
```

```
// lab 7B receiver
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
  char pipename[] = "/tmp/mypipe12345678d";  // pathname of the named pipe
  char buf[80];
  int  n, fd;

  // open pipe for reading
  if ((fd = open(pipename,O_RDONLY)) < 0) {
     printf("Pipe open error\n");
     exit(1);
  }
  while ((n = read(fd,buf,80)) > 0) { // read from pipe
       buf[n] = 0;
       printf("<receiver> message [%s] of size %d bytes received\n",buf,n);
  }
  close(fd); // close the pipe
  printf("<receiver> I have completed!\n");
  exit(0);
}
```

Sharing Named Pipes

Besides allowing your own programs to communicate, it is also possible for programs created by different users to communicate via named pipes. Recall that usually mode **0600** (*owner right = read / write*) is used when a named pipe is created. If you want to communicate with processes created by other users, use the mode **0666** instead (so that *everyone* can read and write). Any program knowing the pathname of this named pipe would be able to talk through this pipe to the process reading from the other end. Similarly, you could “listen” to this pipe to “hear” what the process on the other end is saying. You may use the mode **0644** if you only allow other people to listen, but not to talk to you. An interesting phenomenon of *cross-talking* is possible, when there are multiple senders and multiple receivers. In other words, the senders and receivers could greatly interfere with one another. If your program is not cleverly designed, there is a risk of data being *read* by the inappropriate parties, and even worse, resulting in *deadlocks* (more to be covered in Lecture 9).

If you are interested, you may also try this out by modifying the two programs. It is a bit of challenge for beginners to explore this (for your own curiosity). All programs must run on the *same machine* when named pipes are used.

As a final remark, the way of writing the two sender and receiver programs in **Lab 7B** would be similar to your network programming task using *sockets*. Sockets are used for communication between any two processes, either on the *same machine* like a named/unnamed pipe, or on *two different machines* over the network. A socket is created making reference to the *machine name* (domain name or IP address) and a *port number*, which is similar in nature to the pathname in the named pipe. *Client-server programming* is the most common approach for programs communicating via sockets. In our case, the sender program that creates a named pipe often plays the role of a server program and the receiver program that opens an existing pipe for communication often plays the role of a client program. In conventional client-server programming, you will need a bi-directional channel for two-way communication, since the client normally will ask the server to do something and get some results back. This bi-directional communication channel is implemented as a ***pair*** of unnamed pipes or a ***pair*** of named pipes in our exercises. Sockets are *bi-directional* in nature.

Potential Difference between Unix and Linux Pipes (for Information Only)

The generic pipe only allows one-way communication. If two-way communication is needed, *a pair of pipes* would be required. Recognizing the drawback of simplex pipes, **certain** versions of Unix implement *full-duplex pipes* (i.e., bi-directional pipes). In other words, the **pipe** created in **some** Unix systems, e.g. Solaris Unix, actually represents *a pair of communication links*. The default pipe is **fd[1] ⇒ fd[0]**, as we used before. The new pair is **fd[0] ⇒ fd[1]** in the reversed direction, and the child can write into **fd[0]** and the parent can read from **fd[1]**. This is illustrated in **lab7C.c**. However, there is a ***catch*** in this arrangement. There is now a need to detect for the **end of inputs** at the child process, or else the parent process in **lab7C.c** will wait forever, as the child will wait for data at **`read()`**, while the parent has already jumped out of the loop. A possible trick is to send the special <**Ctrl-D**> to the child to inform the child to exit the loop. If you ever run **lab7C.c** on certain Unix machine, e.g. the decommissioned Solaris Unix **rocket**, you could observe that one pipe is sufficient.

Note that this *full-duplex pipe* does ***not*** work in Linux (also not on most Unix machines and Mac). Try to run **lab7C.c** on Linux (e.g., **apollo**). If ever possible, compare the messages printed by the parent process against those from Solaris Unix. You can verify the difference by **`man 2 pipe`** on Linux and **`man -s 2 pipe`** on Solaris Unix and compare the description for the **`pipe`** system calls. TCP sockets actually are full-duplex and are similar to these bi-directional Unix pipes.

The lesson is that you should develop a portable program. *Always* create a pair of pipes to implement two-way communication between parent and child processes. ***Do not rely*** on the slim possibility of a full-duplex pipe available in your Unix. It is an error to write into **fd[0]** or read from **fd[1]** in Linux. Do ***close*** the excessive ends of the pipes before the main program logic to avoid certain bugs.

```
// lab 7C - bidirectional pipe workable on Solaris
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  . . .
  } else if (childid == 0) { // child
     while ((n = read(fd[0],buf,80)) > 0) { // read from pipe
          if (buf[0] == 4) break; // know about termination
          buf[n] = 0;
          printf("<child> message [%s] of size %d bytes received\n",buf,n);
          // reverse the string
          for (i = 0; i < n; i++) buf2[i] = buf[n-i-1];
          write(fd[0],buf2,n); // send the reversed string
     }
     close(fd[0]);
     close(fd[1]);
     printf("<child> I have completed!\n");
  } else { // parent
     while (1) {
          . . .
          printf("<parent> sending encrypted message [%s] to child\n",buf);
          write(fd[1],buf,n); // send the encrypted string
          // read processed result from child via the other end
          n = read(fd[1],buf2,80);
          buf2[n] = 0;
          printf("<parent> result [%s] of size %d bytes from child\n",buf2,n);
     }
     buf[0] = 4; // use control-D for termination
     write(fd[1],buf,1); // write control-D
     close(fd[0]);
     close(fd[1]);
     wait(NULL);
     printf("<parent> I have completed!\n");
  }
  exit(0);
}
```

Laboratory Exercise

Convert your **hearts.c** program in **Lab 6** using *unnamed pipes* to two programs using *named pipes* for more practicing. The two different programs are for *player* (child) and *arbiter* (parent) respectively. You could decide on the *handshaking protocol* between the players and the arbiter on who would initiate a request and who would be replying with what information.

To go one step further, you could design your player program to take in an argument to indicate whether the program is a *computer* player or a *human* player. A program for human player will accept input from keyboard in real-time in order to make a decision on which card to play when being asked. A computer player may make a *simple choice* as in **Lab 6**, or a *random choice* or an *intelligent choice* when there are more than one playable cards. Now, you have now created a simple *multi-player network game*, when all players run their player program on the ***same*** ***nix** machine as the arbiter. The star communication structure (see **limit.c** in **Lecture 5**) is the most appropriate arrangement to support the concept of the arbiter. To make the program more fun, you could improve the programming logic to make the computer player smarter. Finally, you could implement other common variants of the game with more types of functional or rule-altering cards.

Do this at your own leisure and you ***do not need*** to submit any program for this lab. However, you are encouraged to try this type of programming exercise to prepare for the challenges of network-based socket programming.