Inter-Process Communication with Pipes	**LABORATORY**	**SIX**
OBJECTIVES		
1. More Unix Job and Process Management 2. Files and Streams 3. Communication via Pipes		

More Unix Job and Process Management

Recall that in Unix/Linux, one or more processes will be created when executing programs using the shell and you could view the processes with **`ps`**. You may monitor the list of processes currently executing in the machine, by the command **`top`**. You could attempt to terminate a process by the **`kill`** command.

You could execute multiple processes or jobs when you have only one terminal or connection to Unix/Linux by running them in the background. To put a job to background, use the "**`&`**" operator. The job that is running at the shell and not in background is called the *foreground* job. Only the foreground job can read input from the keyboard. You may first run a job in foreground (normal execution) and then put it to background, by hitting **<Ctrl-Z>** (this is called the *suspend character*) and then type **`bg`**. You could see the list of jobs running in background by typing **`jobs`**. The first job is called **[1]**, the second one is called **[2]** and so on. You could *terminate* a foreground job by hitting **<Ctrl-C>** (this is called the *interrupt character*). If you want to *terminate* a background job, you have to use the **`kill`** command. The following terminates the second job **[2]**.

```
kill %2
```

You may use **`bg`** to put a job to background, so you may use **`fg`** to put a background job to foreground. To put **[2]** to foreground, type **`fg %2`**. When a background job wants to read input from the keyboard, it will print something like **`[1] + Suspended (tty input) a.out`** to inform you that program **a.out** in first job **[1]** wants to get input. You could put it to foreground by **`fg %1`** and then enter the input. You may then put it back to background by <**Ctrl-Z**> and **`bg`**. By carefully *switching* different jobs between foreground and background, you can do quite a lot of work even when you have only one connection to Unix/Linux (e.g. connecting from home). This is also what a strong system administrator is able to do.

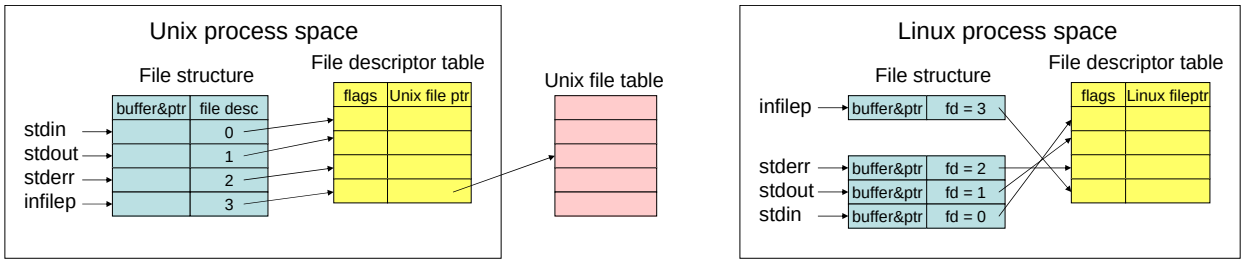
The **`kill`** command actually just sends a *signal* to the specified process, instead of "killing" the process. Recall that in Unix/Linux, a signal is just an interrupt. Normally, we could terminate a process by sending it the proper signal to "kill" it. By default, the signal sent by **`kill`** in C-shell to the process will be the signal **TERM** (i.e., 15). This is meant to *terminate* that process. Try to write a simple C program in which it sleeps for 60 seconds. You may write a C program with an infinite loop like **`while (1) { }`**. (*If you are not an experienced and mindful user, please DO NOT do this, since if you put your infinite looping program in background without killing it, you will consume much CPU time and make other people suffer. This had already happened when some of you are developing the program for lab 3, leaving many child processes running and adopted by process 1, when the parent process terminates*). You could use another terminal to check for its process ID (via **`ps`**). You can now *terminate* it by the **`kill`** command. After you kill a process using normal **`kill`**, you will see a line **Terminated** printed out by C-shell, meaning that **TERM** signal is received.

Other common signals that are supposed to terminate a process include **INT** (2, meaning *interrupt*), **QUIT** (3), **KILL** (9). Try **`kill -3`** on a process and you will see a line **Quit** printed out by C-shell. Signal 9 is perhaps a *strongest signal* that allows processes to be really killed. You could often see that Unix users use the command **`kill -9 pid`** to kill a die-hard process that could not be terminated otherwise.

Command	Usage	Description
bg	**bg**	Move a foreground job to background.
fg	**fg %2**	Move a background job to foreground.
jobs	**jobs**	Show the list of background jobs.
kill	**kill %2** **kill 4321** **kill –QUIT 4321** **kill –9 4321**	Terminate a background job. Terminate process 4321. Quit process 4321. Kill process 4321.
time	**time a.out**	Show the total execution time of a program and its share of CPU time.
uptime	**uptime**	Show the period of time since the computer is up and running and its current workload, e.g. number of users.
top	**top**	Show the CPU and resource utilization of the computer with a list of active processes.

Files and Streams

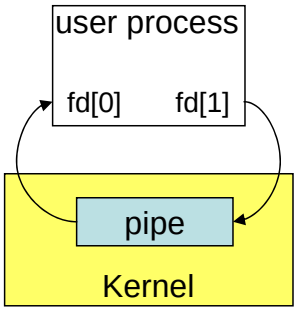
In Unix/Linux, virtually everything is considered to be a file, including *pipes* and *sockets*. Recall that after opening a file, a *file handle* of type **`FILE*`** is returned, pointing to a data structure called a *file structure* in the user area of the process. It contains a *buffer* and a *file descriptor*. A file is identified by either a *file handle* or a *file descriptor*. Pipes and sockets are used for communication between processes, and we call them *streams*. This is because unlike a file, for which a program may jump around in the file (though not always the case), a program can only take (read) input from an input stream and would put (write) output to an output stream *sequentially*. Thus, in Unix/Linux, every stream is considered as a *file*, which is identified by a *file descriptor*, indirectly via the *file handle*. Recall the structure of the file structure and file descriptor table associated with each user process, and the shared Unix/Linux file table pointing to the real storage for the file.



Communication via Pipes

In Unix/Linux, processes could communicate through a temporary file, so that a process writes to the file and another process reads from the file. Using temporary file is not a clean way of programming, since other processes and users could see the existence of the temporary file and may corrupt it. A better way is to use the *pipe* for communication. A **pipe** is the simplest form of *Inter-Process Communication* (IPC) mechanism in Unix/Linux, and is a *message-based direct communication mechanism* (refer to Lecture 3). A *file* is represented by a file descriptor and a *pipe* is represented by a ***pair** of file descriptors*; the first one is for reading and the second one is for writing. A pipe is created by the system call **`pipe`**. An *array* of integer of size 2 is passed as an argument for the system call to return the pair of file descriptors.

The **`pipe`** system call will accept an array of 2 file descriptors (integers) and create a data structure in the kernel space, as illustrated in the diagram. You could consider the pipe structure to be a queue. One can insert elements to the end of the queue by writing to the pipe, and remove elements from the front of the queue by reading from the pipe. The following program **lab6A.c** creates a pipe identified by **fd[0]** and **fd[1]**. As its name implies, **fd[0]** is for reading (**`stdin`** refers to stream **`0`**) and **fd[1]** is for writing (**`stdout`** refers to stream **`1`**). The process then writes into the pipe data it reads from the keyboard, and then reads from the pipe the information written using another variable. Keyboard input is terminated by <**Ctrl-D**>. Note that there is a *hidden bug* if you input an *empty line* in the program. Could you *debug* it?




```
/* lab 6 A */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main()
{
  int   fd[2];  /* for the pipe */
  char  buf[80], buf2[80];
  int   n;

  if (pipe(fd) < 0) {
     printf("Pipe creation error\n");
     exit(1);
  }
  /* repeat a loop to write into the pipe and read from the same pipe */
  while (1) {
       printf("Please input a line\n");
       n = read(STDIN_FILENO, buf, 80); /* read a line from stdin */
       if (n <= 0) break; /* EOF or error */
       buf[--n] = 0; /* remove newline character */
       printf("%d char in input line: [%s]\n", n, buf);

       write(fd[1], buf, n); /* write to pipe */
       printf("Input line [%s] written to pipe\n",buf);

       n = read(fd[0], buf2, 80); /* read from pipe */
       buf2[n] = 0;
       printf("%d char read from pipe: [%s]\n", n, buf2);
  }
  printf("bye bye\n");
  close(fd[0]);
  close(fd[1]);
  exit(0);
}
```

Note that a pipe in Unix/Linux is like a *water pipe*. It makes no difference whether you pour in two small cups of water or pour in one large cup of water, as long as the volume is the same. Try this out in **lab6B.c**. Here you will input two lines and write them into the pipe. You can then read out from the pipe the single line. You will see that both lines are merged into one single message. Now, try to input two long lines and then some short lines again. What do you observe? What conclusion could you draw?

Do you discover that the *last odd line*, if any, is not received by the child in the program? It is a good practice to *clean up* the pipe before finishing with the program. You could try to modify the program so that the last odd line will be received by the child before it terminates.

```
/* lab 6 B */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main()
{
  int   fd[2];  /* for the pipe */
  char  buf[80], buf2[80];
  int   even, n;

  if (pipe(fd) < 0) {
     printf("Pipe creation error\n");
```

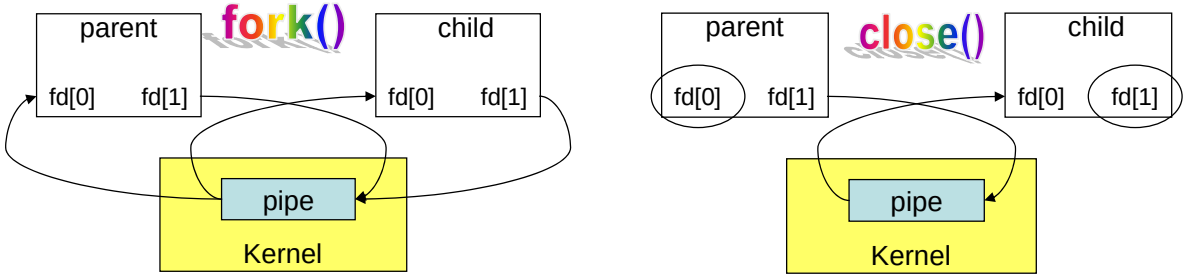
```
    exit(1);
  }
  even = 1;
  /* repeat a loop to write into the pipe and read from the same pipe */
  while (1) {
       even = 1 - even; /* toggle even variable */
       if (even)
          printf("Please input an even line\n");
       else
          printf("Please input an odd line\n");
       n = read(STDIN_FILENO, buf, 80); /* read a line from stdin */
       if (n <= 0) break; /* EOF or error */
       buf[--n] = 0; /* remove newline character */
       printf("%d char in input line: [%s]\n", n, buf);

       write(fd[1], buf, n); /* write to pipe */
       printf("Input line [%s] written to pipe\n",buf);

       if (even) { /* only read with even loop */
          n = read(fd[0], buf2, 80);
          buf2[n] = 0;
          printf("%d char read from pipe: [%s]\n", n, buf2);
       }
  }
  printf("bye bye\n");
  close(fd[0]);
  close(fd[1]);
  exit(0);
}
```

Pipes with Child Processes

A *pipe* is a *communication mechanism*. It will make no sense if a process is writing data into a pipe that it is reading. A natural application of the pipe is the use of more than one process. It is the standard way of communication in Unix/Linux that a parent creates the *pipe* and then executes a **`fork`**. Then both parent process and child process know about the pipe since they share the file descriptors to the pipe. They then **`close`** the *excessive ends* of the pipe to avoid accidental and erroneous access of the pipe. They use the *remaining ends* of the pipe to communicate, passing around data. The actual buffer for the pipe resides inside the system kernel space that a programmer should not see.



In the following simple encryption program, the parent creates a *pipe* and then *forks* a child. Both will *close excessive ends* of the pipe and then the parent will send data to the child using **`write`**. The child receives data from the parent using **`read`**. The return value to **`read`** is the number of bytes read from the pipe into the buffer. It is negative if there is an error. Note that all data passed between parent and child is a sequence of *consecutive bytes*, without any predefined boundary. An error will occur if a reader tries to read from a pipe when the other end is closed, or a writer tries to write to a pipe when the other end is closed. Check for *negative return value* to conclude for the *completion of using the pipe*.

```
/* lab 6 C */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main()
{
  char  mapl[] = "qwertyuiopasdfghjklzxcvbnm"; /* for encoding letter */
  char  mapd[] = "1357924680"; /* for encoding digit */
```

```
  int   fd[2];  /* for the pipe */
  char  buf[80];
  int   i, n, returnpid;

  if (pipe(fd) < 0) {
     printf("Pipe creation error\n");
     exit(1);
  }
  returnpid = fork();
  if (returnpid < 0) {
     printf("Fork failed\n");
     exit(1);
  } else if (returnpid == 0) { /* child */
     close(fd[1]); /* close child out */
     while ((n = read(fd[0],buf,80)) > 0) { /* read from pipe */
          buf[n] = 0;
          printf("<child> message [%s] of size %d bytes received\n",buf,n);
     }
     close(fd[0]);
     printf("<child> I have completed!\n");
  } else { /* parent */
     close(fd[0]); /* close parent in */
     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[1],buf,n); /* send the encrypted string */
     }
     close(fd[1]);
     wait(NULL);
     printf("<parent> I have completed!\n");
  }
  exit(0);
}
```

Note that in **lab6C.c**, only the parent can pass data to the child, since there is only *one pipe*, with *two ends* (two file descriptors as an array). If the child needs to return data back to the parent in conventional Unix, *another pipe* is needed (with another two file descriptors) and the direction of communication in this second pipe will be reversed. Such a conventional pipe is called a *non-duplex pipe*. To see why it is important to close *unused pipes*, consider **lab6D.c** in which ***both*** child and parent write into the same pipe. In other words, the child forgets to *close* its end of the outgoing pipe. Try to type in inputs repeatedly and observe the outputs. Now, the reader would read in a mess, with mixed data from both parent and child.

```
/* lab 6 D */
. . .
int main()
{
  . . .
  } else if (returnpid == 0) { /* child */
     /* close(fd[1]); child forgets to close out */
     while ((n = read(fd[0],buf,80)) > 0) { /* read from pipe */
          buf[n] = 0;
          printf("<child> message [%s] of size %d bytes received\n",buf,n);
          sleep(3); /* add this line to delay child */
          for (i = 0, j = 0; i < n; i = i+2, j++) /* skip the odd characters */
              buf2[j] = buf[i];
          write(fd[1],buf2,j); /* accidentally echo back new string via fd[1] */
     }
     close(fd[0]);
     printf("<child> I have completed!\n");
  } else { /* parent */
     . . .
}
```

Laboratory Exercise

You have created *n* children from a parent to distribute cards to the children in **Lab 4.** In real applications, the parent may not know the data until provided by the user. It is thus important that a parent is able to *pass data to children* after they are created, similar to **lab6C.c**, which makes use of the **pipe**. More importantly, children often need to *convey results back* to the parent, since children are created by a parent to help handling some of the workload.

You are to extend exercise in **Lab 4** with **lab6C.c** to play a simple version of the ***Big 2*** game, to allow *communication* between parent and child processes. The parent first creates the necessary pipes (two pairs for each child, one from parent to child and the other from child to parent), and then forks the child processes. The parent and each child will *close* the excessive ends of the respective pipes. They will communicate via the pipes to carry out the necessary computation. Finally, everyone closes all the pipes and the parent will wait for all the children to terminate at the end of the game.

You would create *n* children to play the game. Input is received from the *keyboard*, but *input redirection* from a file is expected, so *end-of-file testing* should be adopted for the input. The input file contains a collection of cards. Each card is of the form <suit rank>, where suit is either **S** (♠ or *spade*), **H** (♥ or *heart*), **C** (♣ or *club*) or **D** (♦ or *diamond*), and rank is drawn from the set {**2, A**, **K**, **Q**, **J**, **T**, **9**, **8**, **7**, **6**, **5**, **4**, **3**} in descending order, where **T** means **10**. Thus the largest card in the deck is ♠**2** and the smallest card is ♦**3**. You can assume that there is no error in the input lines. Parent reads in the cards until *EOF*. Since people may be careless, sometimes the deck of cards may contain duplicated cards from another deck. If there is a duplicated card, the parent should detect it. It will just keep the first one and discard the duplicated card, printing a warning message accordingly. The correct cards are dealt to the children in a *round-robin manner* and the game will start.

In the simplified Big 2 game, a player can only play *a single card*. Nothing else can be played: no pair, no straight, no flush etc. Each player will take turn to play cards out of its hand, until there is no card. The first player to play all its cards is the *winner*. The last player remaining is the *loser*. There are three simple rules to play cards. **Rule 1**: the card player by the next player must be larger than the card played by the previous player. **Rule 2**: if a player has no card larger than the previous player, it will *pass* and the next player will try to play a card. **Rule 3**: if everyone else passes, the last player of the previously played largest card can select to play any card from its hand.

For fairness, the player holding the smallest card, i.e. ♦**3**, will play first and it must play that smallest card ♦**3**. All players are very simple, sometimes naïve, and do not play strategically. Since only single card will be played, each player will only play the smallest card that is just larger than the previous played card. A player will play the smallest card from its hand if all other players pass. After a player finishes playing all its cards, the next player will continue to play a larger card. If no player can play a larger card than the finishing player, the player next to the finishing player will be allowed to play the smallest card from its hand, since all players have passed.

In this program, the parent will act like the table for holding the cards, as well as the arbitrator to tell each child what card has been played. A child will play a card by sending it via a *pipe* to the parent, who will then relay the card to the next child via a *pipe* for its consideration. A child will send a "*pass*" call if it cannot play a card. A child will send a "*completion*" call if it has played the last card. Do not forget to ***wait*** for the child to complete and ***close*** the pipes in the end of game. Please provide proper comments and check your program before submission. Your program must run on **apollo** or **apollo2**.

Sample executions (input data are stored inside a text file, e.g. **`card.txt`**):

```
playGame 4 < card.txt
```

```
D4 HK CA  DA SK HQ CK DK SQ   HJ CQ DQ SJ HT CJ DJ ST H9 CT HA S9 C9 H8   D9 S7 H6 D8
C8 S2 H5 C6 C3 S6 H4   C5 D6 DT H3 C4 D5  S4 H2 H7   C7 S3 S8 C2 D3   S5 D7 SA D2
```

Sample output 1:

```
Parent: the child players are 23456 23457 23458 23459
Parent: duplicated card HK is discarded
Child 1, pid 23456: I have 13 cards
Child 2, pid 23457: I have 13 cards
Child 4, pid 23459: I have 13 cards
Child 3, pid 23458: I have 13 cards
Child 1, pid 23456: D4 SK SQ SJ ST S9 S7 S2 S6 DT S4 S3 S5
Child 4, pid 23459: DA DK DQ DJ HA D9 C8 C3 D6 D5 C7 D3 D2
Child 2, pid 23457: HK HQ HJ HT H9 C9 H6 H5 H4 H3 H2 S8 D7
Child 3, pid 23458: CA CK CQ CJ CT H8 D8 C6 C5 C4 H7 C2 SA
Child 4: play D3
Parent: child 4 plays D3
Child 1: play S3
Parent: child 1 plays S3
Child 2: play H4
Parent: child 2 plays H4
Child 3: play C5
Parent: child 3 plays C5
Child 4: play D6
Parent: child 4 plays D6
Child 1: play S6
Parent: child 1 plays S6
. . .
Child 1: play S2
Parent: child 1 plays S2
Child 2: pass
Parent: child 2 passes
Child 3: pass
Parent: child 3 passes
Child 4: pass
Parent: child 4 passes
Child 1: play D4
Parent: child 1 plays D4
Child 2: play H5
Parent: child 2 plays H5
. . .
Child 4: play DA
Parent: child 4 plays DA
Child 1: pass
Parent: child 1 passes
Child 2: play H2
Parent: child 2 plays H2
Child 3: pass
Parent: child 3 passes
Child 4: pass
Parent: child 4 passes
Child 1: pass
Parent: child 1 passes
Child 2: play H3
Parent: child 2 plays H3
. . .
Child 3: play H8
Parent: child 3 plays H8
Child 3: I complete
Parent: child 3 is winner
Child 4: pass
Parent: child 4 passes
Child 1: play ST
Parent: child 1 plays ST
. . .
Child 1: play S4
Parent: child 1 plays S4
Child 1: I complete
Parent: child 1 completes
Child 2: play C9
Parent: child 2 plays C9
Child 4: pass
```

```
Parent: child 4 passes
Child 2: play H9
Parent: child 2 plays H9
Child 2: I complete
Parent: child 2 completes
Parent: child 4 is loser
```

playGame 3 < card.txt

```
D3 HK CA DA SK HQ CK DK SQ HJ CQ DQ SJ HT HK CJ CA DJ ST H9 CT CA HA
```

Sample output 2:

```
Parent: the child players are 13586 13588 13589
Parent: duplicated HK discarded
Parent: duplicated CA discarded
Parent: duplicated CA discarded
Child 1, pid 13586: I have 7 cards
Child 1, pid 13586: D4 DA CK HJ SJ DJ CT
Child 2, pid 13588: I have 7 cards
Child 3, pid 13589: I have 6 cards
Child 3, pid 13589: CA HQ SQ DQ CJ H9
Child 2, pid 13588: HK SK DK CQ HT ST HA
Child 1: play D3
Parent: child 1 plays D3
Child 2: play HT
Parent: child 2 plays HT
Child 3: play CJ
Parent: child 3 plays CJ
Child 1: play HJ
Parent: child 1 plays HJ
. . .
Child 2: play SK
Parent: child 2 plays SK
Child 2: I complete
Parent: child 2 is winner
Child 3: pass
Parent: child 3 passes
Child 1: pass
Parent: child 1 passes
Child 3: play H9
Parent: child 3 plays H9
Child 1: play DJ
Parent: child 1 plays DJ
Child 3: play SQ
Parent: child 3 plays SQ
Child 3: I complete
Parent: child 3 completes
Parent: child 1 is loser
```

playGame 5 < card.txt

```
S7 H6 D8 C8 S2 H5 C6 C3 S6 H4 C5 D6 DT H3 C4 D5 S4 H2 H7 C7 S3 S8 C2 D3 S5 D7
SA D3 D2
```

Sample output 3:

```
. . .
Child 4: play D3
Parent: child 4 plays D3
Child 5: play C4
Parent: child 5 plays C4
Child 1: play D5
Parent: child 1 plays D5
. . .
Parent: child 3 plays H2
Child 3: I complete
Parent: child 3 is winner
. . .
Parent: child 2 is loser
```

Name your program **playGame.c** and ***submit it via BlackBoard on or before 23 March***.