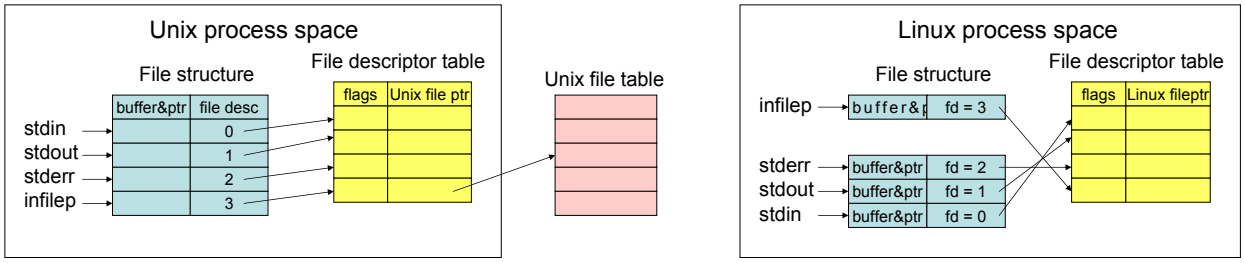
Inter-Process Communication with Pipes	**LABORATORY**	**SIX**
OBJECTIVES		
1. Files and Streams 2. Communication via Pipes		

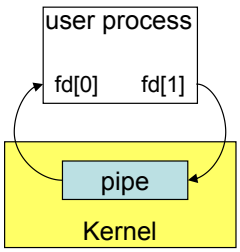
Files and Streams

In Unix/Linux, virtually everything is considered to be a ***file***, including *devices*, *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, 4 and 5). 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 6A
#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 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 before it terminates.

```
// lab 6B
#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
```

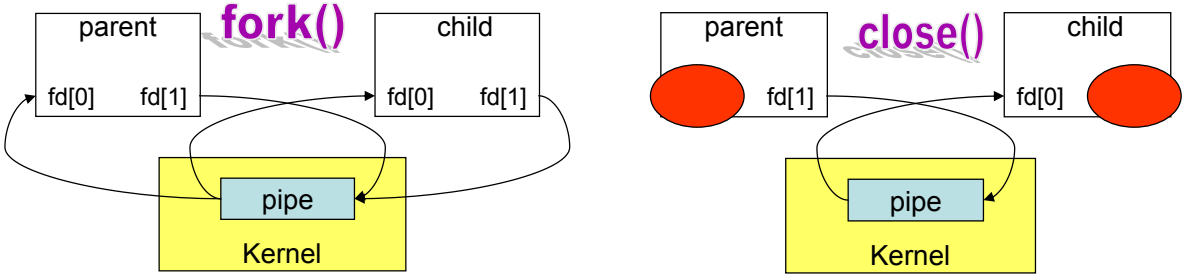
```
        if (even)
           printf("Please input an even line\n");
        else
           printf("Please input an odd line\n");
        n = read(STDIN_FILENO, buf, 80); // read a line from stdin
        if (n <= 0) break; // EOF or error
        buf[--n] = 0; // remove newline character
        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. You may also try to execute **`fork()`** before creating the pipe with **`pipe()`** in **lab6C.c**. Do you know what happens and why?



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 6C
#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 6D
. . .
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 and to distribute tasks to those children in **Lab 4**, by having the child getting the necessary data from the argument list or from variables. In real applications, the parent *may not know* the data before-hand, until provided by the user. It is also possible that the parent would need to decide what data to be passed to which child *on demand*, e.g. pass the next available task to the child which just completes its current task in the presence of many tasks to be assigned. It would be more efficient for the simulation study in **Lab 4** when there are many tasks but a limited number of child processes. 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 usually with a goal to help handling some of the workload of the parent. The use of *exit status* could only return a single value between 0 to 255, and only when a child process terminates. Passing data back to parent when the child processes are still executing is therefore needed.

You are to extend the program **lab6C.c** to play a simplified version of the **Uno** game, to allow *communication* between parent and child processes. The parent first *creates* all the necessary *pipes* (***two pipes*** for each child, one from parent to child and the other from child to parent), and then *forks* the *child processes*. Note that each pipe is associated with two file descriptors, one for read and one for write. The parent and each child will ***close*** the excessive ends of the respective pipes. This is very important to help program development. Failure to close excessive pipes could often result in an incorrect program. The processes 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. Effectively, a star communication topology is adopted and the parent is playing the key role.

You are to create *n* child processes for the players ($3 \le n \le 6$) in this game. The C program is written in such a way that it accepts inputs from the ***keyboard***, e.g., using **`scanf()`** or **`fgets()`** (**`gets()`** should ***not*** be used due to its dangerous buffer overflow problem). We will then compile this C program and ***run*** it based on the technique of *input redirection* from a file for easy testing (see the sample executions). Thus, *end-of-file testing* should be adopted for handling the inputs from the ***keyboard***.

The actual input data 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 {**A**, **K**, **Q**, **J**, **T**, **9**, **8**, **7**, **6**, **5**, **4**, **3, 2**} in that order, where **T** means **10**. There are two additional joker cards, presented as **JO** and **jo**. You can assume that there is no error in the input lines, e.g. no duplicated cards. Parent reads in the cards until *EOF*. The cards are dealt to the children in a *round-robin manner* and the game will start by turning up the first card in the **deck**, and placing it onto the **discard pile**.

The **Uno** game is also known as the **Last Card** game. In the original **Uno** game, there are 108 cards for *n* players, each being dealt 7 cards initially. In our *simplified version*, we use the standard playing card with 54 cards (inclusive of *two jokers*). We also remove those **wild** cards as well as **skip**, **draw two** cards. In our simplified **Uno** game, each player is dealt **5 cards**. The *first player* will try to play *a card* from his/her hand. A card can be played only if it matches in ***suit*** or in ***rank*** to the card on the top of the **discard pile**. The played card will then be put onto the top of the **discard pile**. If no card can be played, the player will draw the first card from the **deck**. Each player will then take turn to play/draw a card. The first player who gets rid of all his/her cards will be the *winner*.

Here are the simplified game rules for a player. **Rule 1**: each player must play a card of the *same rank* or of the *same suit* as the top card on the **discard pile**, unless the player has no card to play. **Rule 2**: the player can play a *joker card* ***only if*** he/she has *no other card* to play; the joker card is considered as the *wild card* that can match any suit and any rank; playing a joker card will cause a ***reverse*** in the direction of card play (from clockwise to counter-clockwise or vice versa). **Rule 3**: a player will draw a card from the **deck** if no card can be played; if the drawn card is playable, it will be played now.

There is a penalty imposed on the losers and that will usually be calculated as the total number of points for cards remaining in their hands. As a result, it is better not to keep high ranked cards to reduce the penalty. We assume that each child process is following this simple guideline: in case of multiple cards

playable, the *highest card* will be played, assuming the rank of **JO > jo > ♠A > ♥A > ♣A > ♦A > … > ♣2 > ♦2** to reduce potential penalty, where **JO** and **jo** represent the big and little jokers respectively.

In this program, the parent will act like the *table* for holding the **deck** of cards for drawing, as well as the *arbitrator* to tell the next child going to play a card what card has been played, i.e. the *top card* of the **discard pile**. A child will play a card by sending its card via a *pipe* to the parent, who will then relay the top card played to the next child via a *pipe* for its consideration. If the child cannot play any card, the parent will send it the card drawn from the deck. The parent actually serves as the *overall controller* of the whole program, prompting children for cards and passing cards drawn to children.

Here is a simple arrangement: each child always tries to read from the pipe about *request* from parent. Parent starts the game by "writing" `topcard` to the first child and then "reads" from that child. Child "reads" from parent for this top card and "writes" its card to be played to parent with `play`. Parent then "writes" `topcard` played to the next child. If a child has no card to play, it "writes" to parent with `cantplay`. Parent then "writes" the card drawn to this child with `draw`. That child has to reply with either `play` (the drawn card can be played) or `cantplay2` (the drawn card cannot be played too) before the next child is invited to play. Note that the parent needs to check whether a joker card **JO** or **jo** is played, and then invite the next child in the reversed order.

When the game is finished with a winner, parent "writes" `winner` to all children and concludes the game. If a player cannot draw a card due to running out of cards in the **deck**, we *simplify* the game to conclude without any winner, i.e. no one wins and "writes" `nowinner` to all children. Each child reports its own hand at the end. In the actual **uno** game, upon the **deck** running out, the **discard pile** is *reshuffled* to become the **deck** for drawing and this is repeated, until a winner is determined. We also simplify the game to relieve a player of the burden to declare ***uno*** (or ***last card***) when there is only one card remaining. A penalty would be imposed in the original game if that going-to-win player is caught not making such a declaration.

Do not forget to ***wait*** for all the children to complete and ***close*** the pipes in the end of the game. Please provide proper comments and check your program before submission. Your program must run on **apollo** or **apollo2**. Here are some sample executions, with content of data file **`card.txt`** shown.

`playcard 3 < card.txt`

```
SJ S6 D8 SQ D6 S2 H5 D9 D4 S8 H7 D3 DT H6 D2 CQ H8 ST C7 H3 S9 HJ D5 C6 HT DA S5 C3 SA CT C8 C4
C9 CJ C5 H9 S7 C2 D7 CK
```

Sample output 1 (output lines may not be in this particular order due to concurrent process execution):

```
Parent, pid 23432: 3 children, 40 cards
Child 1, pid 23433: received cards SJ SQ H5 S8 DT
Child 3, pid 23435: received cards D8 S2 D4 D3 D2
Child 2, pid 23434: received cards S6 D6 D9 H7 H6
Parent, pid 23432: top card CQ
Child 1, pid 23433: my cards SJ SQ H5 S8 DT
Child 1, pid 23433: play card SQ
Parent, pid 23432: top card SQ
Child 2, pid 23434: my cards S6 D6 D9 H7 H6
Child 2, pid 23434: play card S6
Parent, pid 23432: top card S6
Child 3, pid 23435: my cards D8 S2 D4 D3 D2
Child 3, pid 23435: play card S2
Parent, pid 23432: top card S2
Child 1, pid 23433: my cards SJ H5 S8 DT
Child 1, pid 23433: play card SJ
Parent, pid 23432: top card SJ
Child 2, pid 23434: my cards D6 D9 H7 H6
Child 2, pid 23434: no card to play, draw card
Parent, pid 23432: H8 drawn by child 2
Child 2, pid 23434: my cards D6 D9 H7 H6 H8
Child 2, pid 23434: still no card to play
Parent, pid 23432: top card SJ
Child 3, pid 23435: my cards D8 D4 D3 D2
Child 3, pid 23435: no card to play, draw card
Parent, pid 23432: ST drawn by child 3
Child 3, pid 23435: my cards D8 D4 D3 D2 ST
Child 3, pid 23435: play card ST
Parent, pid 23432: top card ST
```

```
Child 1, pid 23433: my cards H5 S8 DT
Child 1, pid 23433: play card DT
Parent, pid 23432: top card DT
Child 2, pid 23434: my cards D6 D9 H7 H6 H8
Child 2, pid 23434: play card D9
Parent, pid 23432: top card D9
Child 3, pid 23435: my cards D8 D4 D3 D2
Child 3, pid 23435: play card D8
Parent, pid 23432: top card D8
Child 1, pid 23433: my cards H5 S8
Child 1, pid 23433: play card S8
Parent, pid 23432: top card S8
Child 2, pid 23434: my cards D6 H7 H6 H8
Child 2, pid 23434: play card H8
Parent, pid 23432: top card H8
Child 3, pid 23435: my cards D4 D3 D2
Child 3, pid 23435: no card to play, draw card
Parent, pid 23432: C7 drawn by child 3
Child 3, pid 23435: my cards D4 D3 D2 C7
Child 3, pid 23435: still no card to play
Parent, pid 23432: top card H8
Child 1, pid 23433: my cards H5
Child 1, pid 23433: play card H5
Child 1, pid 23433: end game, I am winner
Child 2, pid 23434: end game, my remaining cards D6 H7 H6
Child 3, pid 23435: end game, my remaining cards D4 D3 D2 C7
Parent, pid 23432: child 1 is winner
Parent, pid 23432: 21 cards remaining in the deck
```

playcard 6 < card.txt

```
SJ S6 D8 SQ D6 S2 H5 D9 D4 S8 H7 D3 DT H6 D2 CQ H8 ST C7 H3 S9 HJ D5 C6 HT DA S5 C3 SA CT C8 C4
C9 CJ C5 H9 S7 C2 D7 CK
```

Sample output 2:

```
Parent, pid 23456: 6 children, 40 cards
Child 4, pid 23460: received cards SQ S8 CQ HJ C3
Child 1, pid 23457: received cards SJ H5 DT C7 HT
Child 2, pid 23458: received cards S6 D9 H6 H3 DA
Child 6, pid 23462: received cards S2 D3 ST C6 CT
Child 3, pid 23459: received cards D8 D4 D2 S9 S5
Child 5, pid 23461: received cards D6 H7 H8 D5 SA
Parent, pid 23456: top card C8
Child 1, pid 23457: my cards SJ H5 DT C7 HT
Child 1, pid 23457: play card C7
Parent, pid 23456: top card C7
Child 2, pid 23458: my cards S6 D9 H6 H3 DA
Child 2, pid 23458: no card to play, draw card
Parent, pid 23456: C4 drawn by child 2
Child 2, pid 23458: my cards S6 D9 H6 H3 DA C4
Child 2, pid 23458: play card C4
Parent, pid 23456: top card C4
. . .
Parent, pid 23456: top card H5
Child 2, pid 23458: my cards D9 H3 DA
Child 2, pid 23458: play card H3
Parent, pid 23456: top card H3
Child 3, pid 23459: my cards D8 D2 C2
Child 3, pid 23459: no card to play, draw card
Parent, pid 23456: CK drawn by child 3
Child 3, pid 23459: my cards D8 D2 C2 CK
Child 3, pid 23459: still no card to play
Parent, pid 23456: top card H3
Child 4, pid 23460: my cards S8 CQ C3
Child 4, pid 23460: play card C3
Parent, pid 23456: top card C3
Child 5, pid 23461: my cards H7 D5
Child 5, pid 23461: no card to play, no more card to draw
Child 5, pid 23461: end game, my remaining cards H7 D5
Child 2, pid 23458: end game, my remaining cards D9 DA
Child 4, pid 23460: end game, my remaining cards S8 CQ
Child 1, pid 23457: end game, my remaining cards DT
Child 3, pid 23459: end game, my remaining cards D8 D2 C2 CK
Child 6, pid 23462: end game, my remaining cards S2 D3 CT S7 D7
Parent, pid 23456: no winner
Parent, pid 23456: all cards consumed
```

playcard 3 < card.txt

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

Sample output 3:

```
Parent, pid 12321: 3 children, 54 cards
Child 2, pid 12323: received cards S6 D6 D9 H7 jo
Child 3, pid 12324: received cards D8 S2 D4 D3 H6
Child 1, pid 12322: received cards SJ SQ H5 S8 DT
Parent, pid 12321: top card D2
Child 1, pid 12322: my cards SJ SQ H5 S8 DT
Child 1, pid 12322: play card DT
Parent, pid 12321: top card DT
Child 2, pid 12323: my cards S6 D6 D9 H7 jo
Child 2, pid 12323: play card D9
Parent, pid 12321: top card D9
Child 3, pid 12324: my cards D8 S2 D4 D3 H6
Child 3, pid 12324: play card D8
Parent, pid 12321: top card D8
Child 1, pid 12322: my cards SJ SQ H5 S8
Child 1, pid 12322: play card S8
Parent, pid 12321: top card S8
Child 2, pid 12323: my cards S6 D6 H7 jo
Child 2, pid 12323: play card S6
Parent, pid 12321: top card S6
Child 3, pid 12324: my cards S2 D4 D3 H6
Child 3, pid 12324: play card H6
Parent, pid 12321: top card H6
Child 1, pid 12322: my cards SJ SQ H5
Child 1, pid 12322: play card H5
Parent, pid 12321: top card H5
Child 2, pid 12323: my cards D6 H7 jo
Child 2, pid 12323: play card H7
Parent, pid 12321: top card H7
Child 3, pid 12324: my cards S2 D4 D3
Child 3, pid 12324: no card to play, draw card
Parent, pid 12321: JO drawn by child 3
Child 3, pid 12324: my cards S2 D4 D3 JO
Child 3, pid 12324: play joker JO
Parent, pid 12321: joker played, direction reversed
Parent, pid 12321: top card H7
Child 2, pid 12323: my cards D6 jo
Child 2, pid 12323: play joker jo
Parent, pid 12321: joker played, direction reversed
Parent, pid 12321: top card H7
Child 3, pid 12324: my cards S2 D4 D3
Child 3, pid 12324: no card to play, draw card
Parent, pid 12321: CQ drawn by child 3
Child 3, pid 12324: my cards S2 D4 D3 CQ
Child 3, pid 12324: still no card to play
Parent, pid 12321: top card H7
Child 1, pid 12322: my cards SJ SQ
Child 1, pid 12322: no card to play, draw card
Parent, pid 12321: H8 drawn by child 1
Child 1, pid 12322: my cards SJ SQ H8
Child 1, pid 12322: play card H8
Parent, pid 12321: top card H8
Child 2, pid 12323: my cards D6
Child 2, pid 12323: no card to play, draw card
Parent, pid 12321: ST drawn by child 2
Child 2, pid 12323: my cards D6 ST
Child 2, pid 12323: still no card to play
Parent, pid 12321: top card H8
Child 3, pid 12324: my cards S2 D4 D3 CQ
Child 3, pid 12324: no card to play, draw card
Parent, pid 12321: C7 drawn by child 3
Child 3, pid 12324: my cards S2 D4 D3 CQ C7
Child 3, pid 12324: still no card to play
Parent, pid 12321: top card H8
Child 1, pid 12322: my cards SJ SQ
Child 1, pid 12322: no card to play, draw card
Parent, pid 12321: H3 drawn by child 1
Child 1, pid 12322: my cards SJ SQ H3
Child 1, pid 12322: play card H3
Parent, pid 12321: top card H3
Child 2, pid 12323: my cards D6 ST
```

```
Child 2, pid 12323: no card to play, draw card
Parent, pid 12321: S9 drawn by child 2
Child 2, pid 12323: my cards D6 ST S9
Child 2, pid 12323: still no card to play
Parent, pid 12321: top card H3
Child 3, pid 12324: my cards S2 D4 D3 CQ C7
Child 3, pid 12324: play card D3
Parent, pid 12321: top card D3
Child 1, pid 12322: my cards SJ SQ
Child 1, pid 12322: no card to play, draw card
Parent, pid 12321: HJ drawn by child 1
Child 1, pid 12322: my cards SJ SQ HJ
Child 1, pid 12322: still no card to play
Parent, pid 12321: top card D3
Child 2, pid 12323: my cards D6 ST S9
Child 2, pid 12323: play card D6
Parent, pid 12321: top card D6
Child 3, pid 12324: my cards S2 D4 CQ C7
Child 3, pid 12324: play card D4
Parent, pid 12321: top card D4
Child 1, pid 12322: my cards SJ SQ HJ
Child 1, pid 12322: no card to play, draw card
Parent, pid 12321: D5 drawn by child 1
Child 1, pid 12322: my cards SJ SQ HJ D5
Child 1, pid 12322: play card D5
Parent, pid 12321: top card D5
Child 2, pid 12323: my cards ST S9
Child 2, pid 12323: no card to play, draw card
Parent, pid 12321: C6 drawn by child 2
Child 2, pid 12323: my cards ST S9 C6
Child 2, pid 12323: still no card to play
Parent, pid 12321: top card D5
Child 3, pid 12324: my cards S2 CQ C7
Child 3, pid 12324: no card to play, draw card
Parent, pid 12321: HT drawn by child 3
Child 3, pid 12324: my cards S2 CQ C7 HT
Child 3, pid 12324: still no card to play
Parent, pid 12321: top card D5
Child 1, pid 12322: my cards SJ SQ HJ
Child 1, pid 12322: no card to play, draw card
Parent, pid 12321: DA drawn by child 1
Child 1, pid 12322: my cards SJ SQ HJ DA
Child 1, pid 12322: play card DA
Parent, pid 12321: top card DA
Child 2, pid 12323: my cards ST S9 C6
Child 2, pid 12323: no card to play, draw card
Parent, pid 12321: S5 drawn by child 2
Child 2, pid 12323: my cards ST S9 C6 S5
Child 2, pid 12323: still no card to play
Parent, pid 12321: top card DA
Child 3, pid 12324: my cards S2 CQ C7 HT
Child 3, pid 12324: no card to play, draw card
Parent, pid 12321: C3 drawn by child 3
Child 3, pid 12324: my cards S2 CQ C7 HT C3
Child 3, pid 12324: still no card to play
Parent, pid 12321: top card DA
Child 1, pid 12322: my cards SJ SQ HJ
Child 1, pid 12322: no card to play, draw card
Parent, pid 12321: SA drawn by child 1
Child 1, pid 12322: my cards SJ SQ HJ SA
Child 1, pid 12322: play card SA
Parent, pid 12321: top card SA
Child 2, pid 12323: my cards ST S9 C6 S5
Child 2, pid 12323: play card ST
Parent, pid 12321: top card ST
Child 3, pid 12324: my cards S2 CQ C7 HT C3
Child 3, pid 12324: play card HT
Parent, pid 12321: top card HT
Child 1, pid 12322: my cards SJ SQ HJ
Child 1, pid 12322: play card HJ
Parent, pid 12321: top card HJ
Child 2, pid 12323: my cards S9 C6 S5
Child 2, pid 12323: no card to play, draw card
Parent, pid 12321: CT drawn by child 2
Child 2, pid 12323: my cards S9 C6 S5 CT
Child 2, pid 12323: still no card to play
Parent, pid 12321: top card HJ
```

```
Child 3, pid 12324: my cards S2 CQ C7 C3
Child 3, pid 12324: no card to play, draw card
Parent, pid 12321: C8 drawn by child 3
Child 3, pid 12324: my cards S2 CQ C7 C3 C8
Child 3, pid 12324: still no card to play
Parent, pid 12321: top card HJ
Child 1, pid 12322: my cards SJ SQ
Child 1, pid 12322: play card SJ
Parent, pid 12321: top card SJ
Child 2, pid 12323: my cards S9 C6 S5 CT
Child 2, pid 12323: play card S9
Parent, pid 12321: top card S9
Child 3, pid 12324: my cards S2 CQ C7 C3 C8
Child 3, pid 12324: play card S2
Parent, pid 12321: top card S2
Child 1, pid 12322: my cards SQ
Child 1, pid 12322: play card SQ
Child 1, pid 12322: end game, I am winner
Child 2, pid 12323: end game, my remaining cards C6 S5 CT
Child 3, pid 12324: end game, my remaining cards CQ C7 C3 C8
Parent, pid 12321: child 1 is winner
Parent, pid 12321: 21 cards remaining in the deck
```

playcard 4 < card.txt

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

Sample output 4:

```
Parent, pid 12345: 4 children, 54 cards
Child 1, pid 12346: received cards SJ D6 D4 DT JO
Child 2, pid 12347: received cards S6 S2 S8 jo CQ
Child 4, pid 12349: received cards SQ D9 D3 D2 ST
Child 3, pid 12348: received cards D8 H5 H7 H6 H8
Parent, pid 12345: top card C7
Child 1, pid 12346: my cards SJ D6 D4 DT JO
Child 1, pid 12346: play joker JO
Parent, pid 12345: joker played, direction reversed
Parent, pid 12345: top card C7
Child 4, pid 12349: my cards SQ D9 D3 D2 ST
Child 4, pid 12349: no card to play, draw card
Parent, pid 12345: H3 drawn by child 4
Child 4, pid 12349: my cards SQ D9 D3 D2 ST H3
Child 4, pid 12349: still no card to play
Parent, pid 12345: top card C7
Child 3, pid 12348: my cards D8 H5 H7 H6 H8
Child 3, pid 12348: play card H7
Parent, pid 12345: top card H7
Child 2, pid 12347: my cards S6 S2 S8 jo CQ
Child 2, pid 12347: play joker jo
Parent, pid 12345: joker played, direction reversed
Parent, pid 12345: top card H7
Child 3, pid 12348: my cards D8 H5 H6 H8
Child 3, pid 12348: play card H8
Parent, pid 12345: top card H8
Child 4, pid 12349: my cards SQ D9 D3 D2 ST H3
Child 4, pid 12349: play card H3
Parent, pid 12345: top card H3
. . .
Parent, pid 12345: top card SK
Child 4, pid 12349: my cards HT H9
Child 4, pid 12349: no card to play, draw card
Parent, pid 12345: DK drawn by child 4
Child 4, pid 12349: my cards HT H9 DK
Child 4, pid 12349: play card DK
Parent, pid 12345: top card DK
Child 1, pid 12346: my cards S9 S7 CA
Child 1, pid 12346: no card to play, draw card
Parent, pid 12345: HA drawn by child 1
Child 1, pid 12346: my cards S9 S7 CA HA
Child 1, pid 12346: still no card to play
Parent, pid 12345: top card DK
Child 2, pid 12347: my cards S2
Child 2, pid 12347: no card to play, draw card
Parent, pid 12345: DJ drawn by child 2
Child 2, pid 12347: my cards S2 DJ
Child 2, pid 12347: play card DJ
```

```
Parent, pid 12345: top card DJ
Child 3, pid 12348: my cards S5 S3 S4
Child 3, pid 12348: no card to play, draw card
Parent, pid 12345: H4 drawn by child 3
Child 3, pid 12348: my cards S5 S3 S4 H4
Child 3, pid 12348: still no card to play
Parent, pid 12345: top card DJ
Child 4, pid 12349: my cards HT H9
Child 4, pid 12349: no card to play, no more card to draw
Child 1, pid 12346: end game, my remaining cards S9 S7 CA HA
Child 4, pid 12349: end game, my remaining cards HT H9
Child 3, pid 12348: end game, my remaining cards S5 S3 S4 H4
Child 2, pid 12347: end game, my remaining cards S2
Parent, pid 12345: no winner
Parent, pid 12345: all cards consumed
```

playcard 5 < card.txt

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

Sample output 5:

```
Parent, pid 12380: 5 children, 54 cards
Child 4, pid 12384: received cards SQ D4 jo H8 HJ
Child 2, pid 12382: received cards S6 H5 D3 JO H3
Child 1, pid 12381: received cards SJ S2 H7 D2 C7
Child 3, pid 12383: received cards D8 D9 DT CQ S9
Child 5, pid 12385: received cards D6 S8 H6 ST D5
Parent, pid 12380: top card C6
Child 1, pid 12381: my cards SJ S2 H7 D2 C7
Child 1, pid 12381: play card C7
Parent, pid 12380: top card C7
Child 2, pid 12382: my cards S6 H5 D3 JO H3
Child 2, pid 12382: play joker JO
Parent, pid 12380: joker played, direction reversed
Parent, pid 12380: top card C7
Child 1, pid 12381: my cards SJ S2 H7 D2
Child 1, pid 12381: play card H7
Parent, pid 12380: top card H7
Child 5, pid 12385: my cards D6 S8 H6 ST D5
Child 5, pid 12385: play card H6
Parent, pid 12380: top card H6
Child 4, pid 12384: my cards SQ D4 jo H8 HJ
Child 4, pid 12384: play card HJ
Parent, pid 12380: top card HJ
. . .
Parent, pid 12380: top card H9
Child 1, pid 12381: my cards S2
Child 1, pid 12381: no card to play, draw card
Parent, pid 12380: C2 drawn by child 1
Child 1, pid 12381: my cards S2 C2
Child 1, pid 12381: still no card to play
Parent, pid 12380: top card H9
Child 2, pid 12382: my cards H3
Child 2, pid 12382: play card H3
Child 2, pid 12382: end game, I am winner
Child 1, pid 12381: end game, my remaining cards S2 C2
Child 5, pid 12385: end game, my remaining cards S5 S7
Child 3, pid 12383: end game, my remaining cards D8
Child 4, pid 12384: end game, my remaining cards H8
Parent, pid 12380: child 2 is winner
Parent, pid 12380: 14 cards remaining in the deck
```

playcard 6 < card.txt

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

Sample output 6:

```
Parent, pid 12468: 6 children, 54 cards
Child 1, pid 12469: received cards SJ H5 DT H8 D5
Child 2, pid 12470: received cards S6 D9 jo ST C6
Child 5, pid 12473: received cards D6 H7 JO S9 S5
Child 3, pid 12471: received cards D8 D4 H6 C7 HT
Child 4, pid 12472: received cards SQ S8 D2 H3 DA
Child 6, pid 12474: received cards S2 D3 CQ HJ C3
Parent, pid 12468: top card SA
```

```
Child 1, pid 12469: my cards SJ H5 DT H8 D5
Child 1, pid 12469: play card SJ
Parent, pid 12468: top card SJ
Child 2, pid 12470: my cards S6 D9 jo ST C6
Child 2, pid 12470: play card ST
Parent, pid 12468: top card ST
Child 3, pid 12471: my cards D8 D4 H6 C7 HT
Child 3, pid 12471: play card HT
Parent, pid 12468: top card HT
Child 4, pid 12472: my cards SQ S8 D2 H3 DA
Child 4, pid 12472: play card H3
Parent, pid 12468: top card H3
Child 5, pid 12473: my cards D6 H7 JO S9 S5
Child 5, pid 12473: play card H7
Parent, pid 12468: top card H7
Parent, pid 12468: top card H7
Child 6, pid 12474: my cards S2 D3 CQ HJ C3
Child 6, pid 12474: play card HJ
Parent, pid 12468: top card HJ
Child 1, pid 12469: my cards H5 DT H8 D5
Child 1, pid 12469: play card H8
Parent, pid 12468: top card H8
Child 2, pid 12470: my cards S6 D9 jo C6
Child 2, pid 12470: play joker jo
Parent, pid 12468: joker played, direction reversed
Parent, pid 12468: top card H8
Child 1, pid 12469: my cards H5 DT D5
Child 1, pid 12469: play card H5
Parent, pid 12468: top card H5
. . .
Parent, pid 12468: top card CQ
Child 5, pid 12473: my cards D6 JO
Child 5, pid 12473: play joker JO
Parent, pid 12468: joker played, direction reversed
Parent, pid 12468: top card CQ
Child 6, pid 12474: my cards D3 C3 CT
Child 6, pid 12474: play card CT
Parent, pid 12468: top card CT
Child 1, pid 12469: my cards DT D5
Child 1, pid 12469: play card DT
Parent, pid 12468: top card DT
Child 2, pid 12470: my cards D9
Child 2, pid 12470: play card D9
Child 2, pid 12470: end game, I am winner
Child 1, pid 12469: end game, my remaining cards D5
Child 4, pid 12472: end game, my remaining cards D2 DA
Child 6, pid 12474: end game, my remaining cards D3 C3
Child 5, pid 12473: end game, my remaining cards D6
Child 3, pid 12471: end game, my remaining cards D8 D4 H6 C7
Parent, pid 12468: child 2 is winner
Parent, pid 12468: 20 cards remaining in the deck
```

Level 1 requirement: *n* child processes are created and capable of printing out the cards received (similar to level 1 in **Lab 4 exercise**), ***and*** I/O redirection is used (i.e. C program reading from **`stdin`**).
Level 2 requirement: the child processes can communicate with the parent and the game plays correctly for the first round when there is no joker card. It would be easier to perform your program testing with fewer cards.
Level 3 requirement: the game is played correctly when there is no joker card.
Level 4 requirement: the game is played correctly.

Bonus level: continue the game by *reshuffling* cards on the discard pile to form the deck again, upon running out of cards in the deck; ***or*** make the jokers carry *combined effects*, e.g. playing the little joker could cause a direction reversal plus a penalty to draw one card for the next player and playing the big joker could cause a direction reversal plus a penalty to draw one card for both neighboring players, or other sensible alternatives. Explain your enhancement with appropriate test data files to demonstrate your program.

Name your program **playcard.c** and **submit it via BlackBoard on or before 28 March, 2022**.