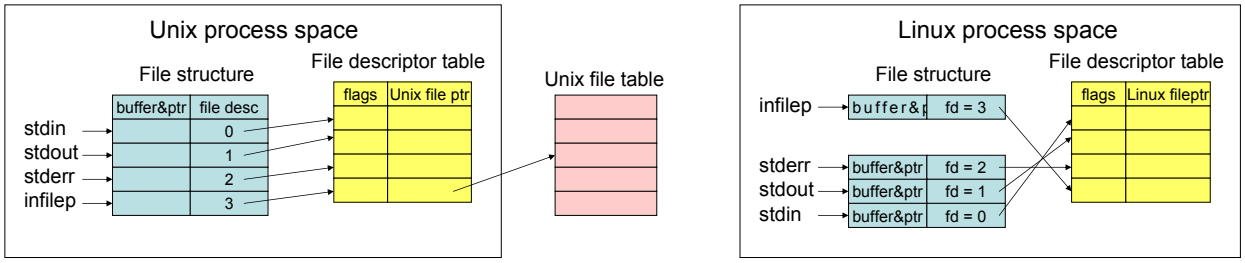
Inter-Process Communication with Pipes	**LABORATORY**	**SIX**
OBJECTIVES		
1. Files and Streams 2. Communication via Pipes		

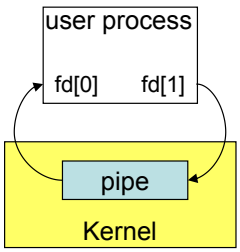
Files and Streams

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?

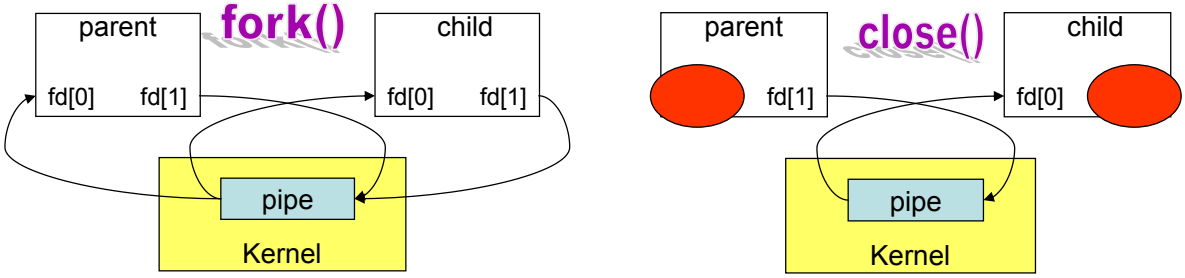
Do you discover that the *last odd line*, if any, is not received in the program? It is a good practice to *clean up* the pipe before finishing with the program. You could try to modify the program so that the last odd line will be received before it terminates.

```
// 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, childid;

  if (pipe(fd) < 0) {
     printf("Pipe creation error\n");
     exit(1);
  }
  childid = fork();
  if (childid < 0) {
     printf("Fork failed\n");
     exit(1);
  } else if (childid == 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 (childid == 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* child processes from a parent and to distribute simulation tasks to those child processes 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 child processes* ***after*** they are created, similar to **lab6C.c**, which makes use of the **pipe**. More importantly, child processes often need to *convey results back* to the parent, since child processes 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 simple version of the **Hearts** 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 child processes to terminate at the end of the game. Effectively, a star communication topology is adopted and the parent is playing the key role.

You would create **4** child processes to play the **Hearts** 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 {**A**, **K**, **Q**, **J**, **T**, **9**, **8**, **7**, **6**, **5**, **4**, **3, 2**} in descending order, where **T** means **10**. You can assume that there is no error in the input lines. Parent reads in the cards until *EOF*. The cards are dealt to the child processes in a *round-robin manner* and the game will start.

In the **Hearts** game, the *start player* plays *a card*. This is called the *lead card*. Each player will take turn to play another card. Here are the game rules. **Rule 1**: each player ***must play*** a card of the *same suit* as the lead card, unless the player has no card of the same suit. In that case, he/she is allowed to play *any card*. We call such a case a "***discard***" of a card in a different suit. **Rule 2**: the player playing the *highest card* of the *same suit* as the lead card will "***win***" all 4 cards played in the round. A card of a different suit than the lead card always ***loses***, even if it is an **A**. For example, if the lead card is ♦**2**, the card ♠**A** will still lose to it. **Rule 3**: the winner of a round will be the start player in the next round.

The scoring rule in **Hearts** is *unconventional*. A player winning a card of ♥ will be given one point. A player "winning" the very special card of ♠**Q** will be given 13 points. So after a hand is played, a total of 26 points would normally be given for all players. There is one single and very important ***exception***. If a particular player wins all 13 ♥ cards and also the ♠**Q**, then he/she will be the ***big winner*** and all the other three players will be given 26 points instead (so there is a total of 78 points given out). The player with the ***least number of points*** will be the champion after the game is played the pre-determined number of times.

A strategy to play is to *avoid* winning a round and attempt not to win the ♠**Q** by accident. So the simple approach is to play low cards whenever possible. Note that if there is already a high card played, a player could try to play a high card that is *just smaller* than the highest played card to reduce the risk of winning in the upcoming tricks (***S1***). If a player is the last player and will win the cards even with the lowest card, he/she will play the highest card to win instead, so as to keep the low cards in hand for future use (***S2***). However, we would not implement these strategies in order to simplify the program logic. In other words, you simply play the lowest card in the required suit of the lead card. In the future, you may consider to introduce artificial intelligence into your program in order to make it play like a competent player.

We assume that each child process is quite naïve in playing the game. The start player will always select the lowest card in his/her hand to play. Since no one wants to get points, each player will try to avoid winning ♠**Q** and ♥ cards. Thus, one will always play the lowest card in the suit of the lead card to avoid winning, or at least to reduce the chance of winning. If there is a chance of "discard" (i.e., no card with the same suit as the lead card) during playing, ♠**Q** will be chosen for discard. If there is no ♠**Q**, then the highest ♥ card will be discarded. If there is no ♥ card, the highest card in the remaining hand will be discarded. If there are more than one highest cards in different suits for discard, just discard ♠ before ♣ before ♦ (a simple tie-breaking rule).

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(s) have been played. A child will play a card by sending its card via a *pipe* to the parent, who will then relay the cards played in the round to the next child via a *pipe* for its consideration. The parent actually serves as the *overall controller* of the whole program, prompting the child processes for card and informing cards played to them.

Here is a simple arrangement: each child always tries to read from the pipe about *request* from parent. Parent starts the game by "writing" `tolead` to the first child and then "reads" from that child. Child "reads" from parent and "writes" its card to be played with `play`. Parent then "writes" `played` with all cards played in this round to the next child. When the game is finished, parent will compute and print the score for each child, since it knows all cards being played and the "winner" for each round. It then "writes" `done` to all child processes and concludes the game. Note that the sequence of messages printed by child processes at the beginning may occur in different orders and you do not need to try to control their order.

Do not forget to ***wait*** for all the child processes 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

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

```
hearts < card1.txt
```

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

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

```
Parent  pid 12345: child players are 12346 12347 12348 12349
Child 2 pid 12347: received DQ CJ SK ST C4 C3 H9 H4 HK D6 D9 CA S9
Child 1 pid 12346: received DK H3 HA HQ HJ H8 D8 S5 C5 S6 C9 S3 CK
Child 2 pid 12347: arranged SK ST S9 HK H9 H4 CA CJ C4 C3 DQ D9 D6
Child 3 pid 12348: received S4 C2 S2 H7 C8 CQ D5 D2 H6 SA DA D4 DT
Child 1 pid 12346: arranged S6 S5 S3 HA HQ HJ H8 H3 CK C9 C5 DK D8
Child 4 pid 12349: received S8 D3 CT HT D7 C6 DJ S7 H2 SQ H5 C7 SJ
Child 4 pid 12349: arranged SQ SJ S8 S7 HT H5 H2 CT C7 C6 DJ D7 D3
Child 3 pid 12348: arranged SA S4 S2 H7 H6 CQ C8 C2 DA DT D5 D4 D2
Parent  pid 12345: round 1 child 1 to lead
Child 1 pid 12346: play H3
Parent  pid 12345: child 1 plays H3
Child 2 pid 12347: play H4
Parent  pid 12345: child 2 plays H4
Child 3 pid 12348: play H6
Parent  pid 12345: child 3 plays H6
Child 4 pid 12349: play H2
Parent  pid 12345: child 4 plays H2
Parent  pid 12345: child 3 wins the trick
```

```
Parent  pid 12345: round 2 child 3 to lead
Child 3 pid 12348: play D2
Parent  pid 12345: child 3 plays D2
Child 4 pid 12349: play D3
Parent  pid 12345: child 4 plays D3
Child 1 pid 12346: play D8
Parent  pid 12345: child 1 plays D8
Child 2 pid 12347: play D6
Parent  pid 12345: child 2 plays D6
Parent  pid 12345: child 1 wins the trick
Parent  pid 12345: round 3 child 1 to lead
Child 1 pid 12346: play S3
Parent  pid 12345: child 1 plays S3
Child 2 pid 12347: play S9
Parent  pid 12345: child 2 plays S9
Child 3 pid 12348: play S2
Parent  pid 12345: child 3 plays S2
Child 4 pid 12349: play S7
Parent  pid 12345: child 4 plays S7
Parent  pid 12345: child 2 wins the trick
Parent  pid 12345: round 4 child 2 to lead
Child 2 pid 12347: play C3
Parent  pid 12345: child 2 plays C3
Child 3 pid 12348: play C2
Parent  pid 12345: child 3 plays C2
Child 4 pid 12349: play C6
Parent  pid 12345: child 4 plays C6
Child 1 pid 12346: play C5
Parent  pid 12345: child 1 plays C5
Parent  pid 12345: child 4 wins the trick
Parent  pid 12345: round 5 child 4 to lead
Child 4 pid 12349: play H5
Parent  pid 12345: child 4 plays H5
Child 1 pid 12346: play H8
Parent  pid 12345: child 1 plays H8
Child 2 pid 12347: play H9
Parent  pid 12345: child 2 plays H9
Child 3 pid 12348: play H7
Parent  pid 12345: child 3 plays H7
Parent  pid 12345: child 2 wins the trick
Parent  pid 12345: round 6 child 2 to lead
Child 2 pid 12347: play C4
Parent  pid 12345: child 2 plays C4
Child 3 pid 12348: play C8
Parent  pid 12345: child 3 plays C8
Child 4 pid 12349: play C7
Parent  pid 12345: child 4 plays C7
Child 1 pid 12346: play C9
Parent  pid 12345: child 1 plays C9
Parent  pid 12345: child 1 wins the trick
Parent  pid 12345: round 7 child 1 to lead
Child 1 pid 12346: play S5
Parent  pid 12345: child 1 plays S5
Child 2 pid 12347: play ST
Parent  pid 12345: child 2 plays ST
Child 3 pid 12348: play S4
Parent  pid 12345: child 3 plays S4
Child 4 pid 12349: play S8
Parent  pid 12345: child 4 plays S8
Parent  pid 12345: child 2 wins the trick
Parent  pid 12345: round 8 child 2 to lead
Child 2 pid 12347: play D9
Parent  pid 12345: child 2 plays D9
Child 3 pid 12348: play D4
Parent  pid 12345: child 3 plays D4
Child 4 pid 12349: play D7
Parent  pid 12345: child 4 plays D7
Child 1 pid 12346: play DK
Parent  pid 12345: child 1 plays DK
Parent  pid 12345: child 1 wins the trick
Parent  pid 12345: round 9 child 1 to lead
Child 1 pid 12346: play S6
Parent  pid 12345: child 1 plays S6
Child 2 pid 12347: play SK
Parent  pid 12345: child 2 plays SK
Child 3 pid 12348: play SA
Parent  pid 12345: child 3 plays SA
```

```
Child 4 pid 12349: play SJ
Parent  pid 12345: child 4 plays SJ
Parent  pid 12345: child 3 wins the trick
Parent  pid 12345: round 10 child 3 to lead
Child 3 pid 12348: play D5
Parent  pid 12345: child 3 plays D5
Child 4 pid 12349: play DJ
Parent  pid 12345: child 4 plays DJ
Child 1 pid 12346: play HA
Parent  pid 12345: child 1 plays HA                  # note that this is the first discard
Child 2 pid 12347: play DQ
Parent  pid 12345: child 2 plays DQ
Parent  pid 12345: child 2 wins the trick
Parent  pid 12345: round 11 child 2 to lead
Child 2 pid 12347: play CJ
Parent  pid 12345: child 2 plays CJ
Child 3 pid 12348: play CQ
Parent  pid 12345: child 3 plays CQ
Child 4 pid 12349: play CT
Parent  pid 12345: child 4 plays CT
Child 1 pid 12346: play CK
Parent  pid 12345: child 1 plays CK
Parent  pid 12345: child 1 wins the trick
Parent  pid 12345: round 12 child 1 to lead
Child 1 pid 12346: play HJ
Parent  pid 12345: child 1 plays HJ
Child 2 pid 12347: play HK
Parent  pid 12345: child 2 plays HK
Child 3 pid 12348: play DA
Parent  pid 12345: child 3 plays DA
Child 4 pid 12349: play HT
Parent  pid 12345: child 4 plays HT
Parent  pid 12345: child 2 wins the trick
Parent  pid 12345: round 13 child 2 to lead
Child 2 pid 12347: play CA
Parent  pid 12345: child 2 plays CA
Child 3 pid 12348: play DT
Parent  pid 12345: child 3 plays DT
Child 4 pid 12349: play SQ
Parent  pid 12345: child 4 plays SQ
Child 1 pid 12346: play HQ
Parent  pid 12345: child 1 plays HQ
Parent  pid 12345: child 2 wins the trick
Parent  pid 12345: game completed
Parent  pid 12345: score = <0 22 4 0>
```

hearts < card2.txt

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

Sample output 2:

```
Parent  pid 12355: child players are 12356 12357 12358 12359
Child 1 pid 12356: received H3 HA HQ HJ H8 D8 S5 C5 S6 C9 S3 CK DK
Child 1 pid 12356: arranged S6 S5 S3 HA HQ HJ H8 H3 CK C9 C5 DK D8
Child 2 pid 12357: received CJ SK ST C4 C3 H9 H4 HK D6 D9 CA S9 DJ
Child 2 pid 12357: arranged SK ST S9 HK H9 H4 CA CJ C4 C3 DJ D9 D6
Child 3 pid 12358: received C2 S2 H7 C8 CQ D5 D2 H6 SA DA D4 DT S4
Child 3 pid 12358: arranged SA S4 S2 H7 H6 CQ C8 C2 DA DT D5 D4 D2
Child 4 pid 12359: received D3 CT HT D7 C6 DQ S7 H2 SQ H5 C7 SJ S8
Child 4 pid 12359: arranged SQ SJ S8 S7 HT H5 H2 CT C7 C6 DQ D7 D3
Parent  pid 12355: round 1 child 1 to lead
Child 1 pid 12356: play H3
Parent  pid 12355: child 1 plays H3
Child 2 pid 12357: play H4
Parent  pid 12355: child 2 plays H4
Child 3 pid 12358: play H6
Parent  pid 12355: child 3 plays H6
Child 4 pid 12359: play H2
Parent  pid 12355: child 4 plays H2
Parent  pid 12355: child 3 wins the trick
Parent  pid 12355: round 2 child 3 to lead
Child 3 pid 12358: play D2
Parent  pid 12355: child 3 plays D2
Child 4 pid 12359: play D3
Parent  pid 12355: child 4 plays D3
Child 1 pid 12356: play D8
```

```
Parent  pid 12355: child 1 plays D8
Child 2 pid 12357: play D6
Parent  pid 12355: child 2 plays D6
Parent  pid 12355: child 1 wins the trick
Parent  pid 12355: round 3 child 1 to lead
Child 1 pid 12356: play S3
. . .
Parent  pid 12355: child 2 wins the trick
Parent  pid 12355: round 13 child 2 to lead
Child 2 pid 12357: play CA
Parent  pid 12355: child 2 plays CA
Child 3 pid 12358: play DT
Parent  pid 12355: child 3 plays DT
Child 4 pid 12359: play SQ
Parent  pid 12355: child 4 plays SQ
Child 1 pid 12356: play HQ
Parent  pid 12355: child 1 plays HQ
Parent  pid 12355: child 2 wins the trick
Parent  pid 12355: game completed
Parent  pid 12355: score = <0 21 4 1>
```

hearts < card3.txt

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

Sample output 3:

```
Parent  pid 12432: child players are 12433 12434 12435 12436
Child 1 pid 12433: received DK H3 HA H8 HJ H4 D8 S5 C5 S6 C9 S3 CK
Child 1 pid 12433: arranged S6 S5 S3 HA HJ H8 H4 H3 CK C9 C5 DK D8
Child 2 pid 12434: received DQ CJ SK ST C4 C3 H9 HQ HK D6 D9 CA S9
Child 2 pid 12434: arranged SK ST S9 HK HQ H9 CA CJ C4 C3 DQ D9 D6
Child 3 pid 12435: received S4 C2 S2 H7 C8 CQ D5 D2 H6 SA DA D4 DT
Child 3 pid 12435: arranged SA S4 S2 H7 H6 CQ C8 C2 DA DT D5 D4 D2
Child 4 pid 12436: received S8 D3 CT HT D7 C6 DJ S7 H2 SQ H5 C7 SJ
Child 4 pid 12436: arranged SQ SJ S8 S7 HT H5 H2 CT C7 C6 DJ D7 D3
Parent  pid 12432: round 1 child 1 to lead
Child 1 pid 12433: play H3
Parent  pid 12432: child 1 plays H3
Child 2 pid 12434: play H9
Parent  pid 12432: child 2 plays H9
Child 3 pid 12435: play H6
Parent  pid 12432: child 3 plays H6
Child 4 pid 12436: play H2
Parent  pid 12432: child 4 plays H2
Parent  pid 12432: child 2 wins the trick
Parent  pid 12432: round 2 child 2 to lead
Child 2 pid 12434: play C3
Parent  pid 12432: child 2 plays C3
Child 3 pid 12435: play C2
Parent  pid 12432: child 3 plays C2
Child 4 pid 12436: play C6
Parent  pid 12432: child 4 plays C6
Child 1 pid 12433: play C5
Parent  pid 12432: child 1 plays C5
Parent  pid 12432: child 4 wins the trick
Parent  pid 12432: round 3 child 4 to lead
Child 4 pid 12436: play D3
. . .
Parent  pid 12432: child 2 wins the trick
Parent  pid 12432: round 13 child 2 to lead
Child 2 pid 12434: play CA
Parent  pid 12432: child 2 plays CA
Child 3 pid 12435: play DT
Parent  pid 12432: child 3 plays DT
Child 4 pid 12436: play SQ
Parent  pid 12432: child 4 plays SQ
Child 1 pid 12433: play HJ
Parent  pid 12432: child 1 plays HJ
Parent  pid 12432: child 2 wins the trick
Parent  pid 12432: game completed
Parent  pid 12432: score = <26 0 26 26>              # big winner child 2 for all hearts and SQ
```

hearts < card4.txt

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

Sample output 4:

```
Parent  pid 12456: child players are 12457 12458 12459 12460
Child 1 pid 12457: received C4 D4 CJ D7 C8 H5 DK SQ HK DA CQ DJ S5
Child 1 pid 12457: arranged SQ S5 HK H5 CQ CJ C8 C4 DA DK DJ D7 D4
Child 2 pid 12458: received HT ST HA D5 SJ H2 C2 S7 S2 C7 C3 H7 S9
Child 2 pid 12458: arranged SJ ST S9 S7 S2 HA HT H7 H2 C7 C3 C2 D5
Child 3 pid 12459: received H9 DT D9 D2 S6 S4 D8 S3 CA H8 D3 C5 C6
Child 3 pid 12459: arranged S6 S4 S3 H9 H8 CA C6 C5 DT D9 D8 D3 D2
Child 4 pid 12460: received HJ CK C9 SA SK DQ H6 CT H4 HQ S8 D6 H3
Child 4 pid 12460: arranged SA SK S8 HQ HJ H6 H4 H3 CK CT C9 DQ D6
Parent  pid 12456: round 1 child 1 to lead
Child 1 pid 12457: play D4
Parent  pid 12456: child 1 plays D4
Child 2 pid 12458: play D5
Parent  pid 12456: child 2 plays D5
Child 3 pid 12459: play D2
Parent  pid 12456: child 3 plays D2
Child 4 pid 12460: play D6
Parent  pid 12456: child 4 plays D6
Parent  pid 12456: child 4 wins the trick
Parent  pid 12456: round 2 child 4 to lead
Child 4 pid 12460: play H3
Parent  pid 12456: child 4 plays H3
Child 1 pid 12457: play H5
Parent  pid 12456: child 1 plays H5
Child 2 pid 12458: play H2
Parent  pid 12456: child 2 plays H2
Child 3 pid 12459: play H8
Parent  pid 12456: child 3 plays H8
Parent  pid 12456: child 3 wins the trick
Parent  pid 12456: round 3 child 3 to lead
Child 3 pid 12459: play D3
Parent  pid 12456: child 3 plays D3
Child 4 pid 12460: play DQ
Parent  pid 12456: child 4 plays DQ
Child 1 pid 12457: play D7
Parent  pid 12456: child 1 plays D7
Child 2 pid 12458: play HA
Parent  pid 12456: child 2 plays HA                  # a discard
Parent  pid 12456: child 4 wins the trick
Parent  pid 12456: round 4 child 4 to lead
Child 4 pid 12460: play H4
Parent  pid 12456: child 4 plays H4
Child 1 pid 12457: play HK
Parent  pid 12456: child 1 plays HK
Child 2 pid 12458: play H7
Parent  pid 12456: child 2 plays H7
Child 3 pid 12459: play H9
Parent  pid 12456: child 3 plays H9
Parent  pid 12456: child 1 wins the trick
Parent  pid 12456: round 5 child 1 to lead
Child 1 pid 12457: play C4
Parent  pid 12456: child 1 plays C4
Child 2 pid 12458: play C2
Parent  pid 12456: child 2 plays C2
Child 3 pid 12459: play C5
Parent  pid 12456: child 3 plays C5
Child 4 pid 12460: play C9
Parent  pid 12456: child 4 plays C9
Parent  pid 12456: child 4 wins the trick
Parent  pid 12456: round 6 child 4 to lead
Child 4 pid 12460: play H6
Parent  pid 12456: child 4 plays H6
Child 1 pid 12457: play SQ                           # discard SQ at earliest possible chance
Parent  pid 12456: child 1 plays SQ
Child 2 pid 12458: play HT
Parent  pid 12456: child 2 plays HT
Child 3 pid 12459: play CA
Parent  pid 12456: child 3 plays CA
Parent  pid 12456: child 2 wins the trick
Parent  pid 12456: round 7 child 2 to lead
Child 2 pid 12458: play S2
. . .
Parent  pid 12456: child 4 wins the trick
```

```
Parent  pid 12456: round 13 child 4 to lead
Child 4 pid 12460: play SA
Parent  pid 12456: child 4 plays SA
Child 1 pid 12457: play DJ
Parent  pid 12456: child 1 plays DJ
Child 2 pid 12458: play S9
Parent  pid 12456: child 2 plays S9
Child 3 pid 12459: play S6
Parent  pid 12456: child 3 plays S6
Parent  pid 12456: child 4 wins the trick
Parent  pid 12456: game completed
Parent  pid 12456: score = <4 15 4 3>
```

hearts < card5.txt

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

Sample output 5:

```
Parent  pid 12578: child players are 12579 12580 12581 12582
Child 1 pid 12579: received CA C3 CJ S3 DA D7 HQ C4 HT C6 H8 D8 S9
Child 1 pid 12579: arranged S9 S3 HQ HT H8 CA CJ C6 C4 C3 DA D8 D7
Child 2 pid 12580: received CQ S2 H2 S5 C9 S8 D9 S6 C8 H6 DT HA CK
Child 2 pid 12580: arranged S8 S6 S5 S2 HA H6 H2 CK CQ C9 C8 DT D9
Child 3 pid 12581: received D3 S4 DQ C2 ST D6 H9 H7 HK H3 HJ H5 S7
Child 3 pid 12581: arranged ST S7 S4 HK HJ H9 H7 H5 H3 C2 DQ D6 D3
Child 4 pid 12582: received SK DK SQ D4 H4 SJ CT C5 D5 C7 D2 SA DJ
Child 4 pid 12582: arranged SA SK SQ SJ H4 CT C7 C5 DK DJ D5 D4 D2
Parent  pid 12578: round 1 child 1 to lead
Child 1 pid 12579: play C3
Parent  pid 12578: child 1 plays C3
Child 2 pid 12580: play C8
Parent  pid 12578: child 2 plays C8
Child 3 pid 12581: play C2
Parent  pid 12578: child 3 plays C2
Child 4 pid 12582: play C5
Parent  pid 12578: child 4 plays C5
Parent  pid 12578: child 2 wins the trick
Parent  pid 12578: round 2 child 2 to lead
Child 2 pid 12580: play H2
Parent  pid 12578: child 2 plays H2
Child 3 pid 12581: play H3
Parent  pid 12578: child 3 plays H3
Child 4 pid 12582: play H4
Parent  pid 12578: child 4 plays H4
Child 1 pid 12579: play H8
Parent  pid 12578: child 1 plays H8
Parent  pid 12578: child 1 wins the trick
Parent  pid 12578: round 3 child 1 to lead
Child 1 pid 12579: play S3
Parent  pid 12578: child 1 plays S3
Child 2 pid 12580: play S2
Parent  pid 12578: child 2 plays S2
Child 3 pid 12581: play S4
Parent  pid 12578: child 3 plays S4
Child 4 pid 12582: play SJ
Parent  pid 12578: child 4 plays SJ
Parent  pid 12578: child 4 wins the trick
Parent  pid 12578: round 4 child 4 to lead
Child 4 pid 12582: play D2
Parent  pid 12578: child 4 plays D2
Child 1 pid 12579: play D7
Parent  pid 12578: child 1 plays D7
Child 2 pid 12580: play D9
Parent  pid 12578: child 2 plays D9
Child 3 pid 12581: play D3
Parent  pid 12578: child 3 plays D3
Parent  pid 12578: child 2 wins the trick
Parent  pid 12578: round 5 child 2 to lead
Child 2 pid 12580: play S5
Parent  pid 12578: child 2 plays S5
Child 3 pid 12581: play S7
Parent  pid 12578: child 3 plays S7
Child 4 pid 12582: play SQ
Parent  pid 12578: child 4 plays SQ
Child 1 pid 12579: play S9
```

```
Parent  pid 12578: child 1 plays S9
Parent  pid 12578: child 4 wins the trick
Parent  pid 12578: round 6 child 4 to lead
Child 4 pid 12582: play D4
. . .
Parent  pid 12578: child 2 wins the trick
Parent  pid 12578: round 13 child 2 to lead
Child 2 pid 12580: play CK
Parent  pid 12578: child 2 plays CK
Child 3 pid 12581: play H7
Parent  pid 12578: child 3 plays H7
Child 4 pid 12582: play DJ
Parent  pid 12578: child 4 plays DJ
Child 1 pid 12579: play CJ
Parent  pid 12578: child 1 plays CJ
Parent  pid 12578: child 2 wins the trick
Parent  pid 12578: game completed
Parent  pid 12578: score = <8 4 0 14>
```

Level 1 requirement: 4 child processes are created and capable of printing out the cards received ***and*** I/O redirection is used (i.e. with your C program reading from **`stdin`** or keyboard). Under this simple requirement, the child processes can just pick out their own cards from the inputs based on their own positions *without having to use the pipe*.
Level 2 requirement: the child processes can receive its cards from the parent and communicate with parent to play using the pipe, and can play the first round correctly in the absence of a discard.
Level 3 requirement: the game is played correctly in the absence of a discard.
Level 4 requirement: the game is played correctly under all situations, with the correct scores computed.

Bonus level: the two better strategies (**S1** and **S2** above) are implemented for child processes so as to play the highest card just lower than the highest played card to reduce risk of winning in future and to win with the highest card if forced to win as the final player. Alternatively, you could implement another cleverer strategy **S3** and explain how it works with examples.

Name your program **hearts.c** and **submit it via BlackBoard on or before 25 March, 2024**.