CSI3131 – Lab2
Inter-process Communication (IPC)

Objectives

To practice inter-process communication using **pipes**, **named-pipes**, and **Signals**. We are going to using a two player game (Tic-Tac-Toe) to demonstrate the inter-process communication. Two players will launch the game from two different shells and the applications will use IPC to update the game moves.

Part1 – PIPES

In Part1 of the attached source files, you'll find the following files:

- **pipe1.c**: is an example of using a pipe to redirect stdout to the output side of a pipe. In another process the read side of the pipe is used to store the input to a buffer, which is displayed on the screen.
- **pipe2.c**: in this example the output of "**ls -l**" is redirected to a pipe. The input side of the pipe is redirected to stdin, which is read, line-by-line using **getline** and displayed on the screen.
- **pipe3.c**: in this example the output of "**ls -l**" is redirected to a pipe. The input side of the pipe is redirected to stdin, which is used by "**sort -r**" to sort the output of "**ls -l**" in reverse order.

Checkout the Linux command **tee**.

```
tee --help
Usage: tee [OPTION]... [FILE]...
Copy standard input to each FILE, and also to standard output.

  -a, --append              append to the given FILEs, do not overwrite
  -i, --ignore-interrupts   ignore interrupt signals
  -p                        diagnose errors writing to non pipes
      --output-error[=MODE]   set behavior on write error.  See MODE below
      --help     display this help and exit
      --version  output version information and exit
```

Try **ls | tee tmp.txt**.

The output of the file list is displayed on the screen and is also saved to a file **tmp.txt**.

Your task is develop a c-program mytee.c, the program is expected to behave similar to tee, but when it displays the output to stdout it adds a line number to every line.

Hint: use pipe2.c as a template, use exec to launch tee with the same arguments as passed to mytee. Redirect stdout to the pipe and read the output line-by-line, concatenate the line number to every line and display the output on the screen.

The output of mytee using the file pipe1.c should look like the following:

```
./mytee.exe < pipe1.c
   1    #include <stdio.h>
   2    #include <unistd.h>
   3    #include <stdlib.h>
   4    #include <sys/wait.h>
   5
   6
   7    /**
   8     * pipe stdout to a buffer
   9     *
  10     */
  11
  12    int main(int argc, char **argv)
  13    {
  14      int pipefd[2];
  15      int pid;
  16      int pid2;
  17
  18      pipe(pipefd);
  19
  20      if (fork() == 0)
  21      {
  22          close(pipefd[0]);    // close reading end in the child
  23
  24          dup2(pipefd[1], 1);  // send stdout to the pipe
  25          dup2(pipefd[1], 2);  // send stderr to the pipe
  26
  27          close(pipefd[1]);    // this descriptor is no longer needed
  28
  29          // exec(...);
  30          for (char c='a'; c<='z'; c++) printf ("%c", c);
  31          printf("%c", '\0'); // end of string char
  32      }
  33      else
  34      {
  35          // parent
  36
  37          char buffer[1024];
  38
  39          close(pipefd[1]);  // close the write end of the pipe in the parent
  40
  41          while (read(pipefd[0], buffer, sizeof(buffer)) != 0)
  42          {
  43            printf("buffer %s", buffer);
  44          }
  45      }
  46    }
```

Part2 – Named Pipes

In Part2 of the attached source files, you'll find the following files:
named_pipe_r.c: demonstrate using a named pipe called my_pipe for input (reading)
named_pipe_w. c: demonstrate using a named pipe called my_pipe for output (writing)
To run the demonstration follow the following steps:

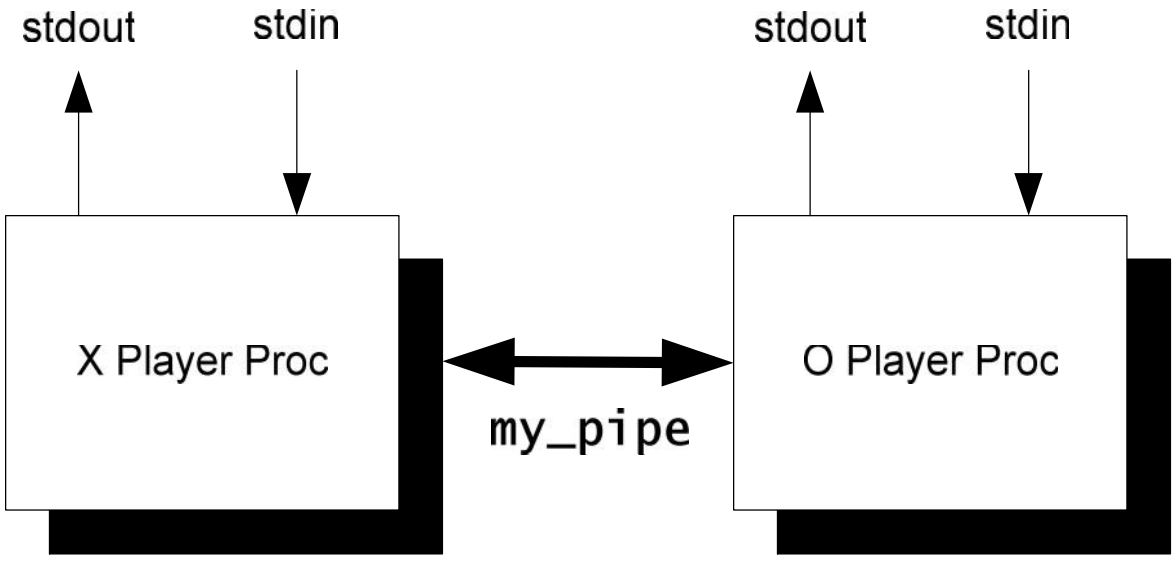
1- compile each application:
   a. **gcc named_pipe_w.c –o named_pipe_w**
   b. **gcc named_pipe_r.c –o named_pipe_r**
2- create a new named pipe call it `my_pipe`
   a. **mkfifo my_pipe**
3- open two new terminals and run each application in one terminal
4- the two applications will exchange greeting messages using the named pipes

The named pipes are type of synchronous communication. The pipe has two ends write & read. Before the pipe operates it must be open for reading & writing by two separate threads/processes. The sender will wait for the receiver and the receiver will wait for the sender. Notice that when you run either side it blocks till the other application starts before the message exchange. You can use the same pipe to exchange message in half duplex mode by alternating read/write modes. To use full duplex mode you need two pipes, one pipe for each direction.

Your task is to use named pipes to play the game tic-tac-toe between two processes, one process is for player X and the other is for player Y.

> Refer to The Game section for explanation on how the game is played and the source files associated with it.

The named pipe will be used to exchange player moves between two player processes – see figure below. The **stdin** is used to get player moves, and **stdout** is used to display the current game board.



Your task is to modify **np_tic_tac_toe.cc** source file to represent a single tic-tac-toe player [X|O]. The two player game will communicate with each other using **my_pipe** FIFO. Use the following steps as guideline:

1- Include a game instance in the application.
2- player X will start, player O is going to wait – turn value will depend on the player
3- Use the **convert2string()** function to convert the game state to a string to write in file
4- Use the **set_game_state(char *state)** to update the game state from the received message
5- if the player's turn: open the pipe in write mode, get the player move, send the move to the other player by writing the game string to the pipe, close the pipe. Check if the game has ended, otherwise increment the turn, goto 6.
6- If not the player's turn: open the pipe in read mode, read the other user's updated game state, close the pipe, set the game state accordingly, display the updated game, and check if the game has ended. If the game did not end increment the game turn and goto 5.
7- A working program will allow two players to use different terminals to play the game and communicating the game movements across applications

Compile & run your application:

1- `g++ -I tic_tac_toe tic_tac_toe/tic_tac_toe.cc part2/np_tic_tac_toe.cc -o np_tic_tac_toe`
2- `make sure the named pipe` is created, if not `mkfifo my_pipe`
3- `run the X player in a new terminal: ./np_tic_tac_toe X`
4- `run the O player in a new terminal ./np_tic_tac_toe O`
5- `run the game`
6- `Verify your solution works using the 3 provided scenarios by piping the provided files to your application. The application should run autonomously to the expected outcome.`
    a. X win scenario: `x_play_list_xwin.txt, o_play_list_xwin.txt, to run this scenario:`
        i. `./np_tic_tac_toe X < x_play_list_xwin.txt`
        ii. `./np_tic_tac_toe O < o_play_list_xwin.txt`
    b. O win scenario: `x_play_list_owin.txt, o_play_list_owin.txt, to run this scenario:`
        i. `./np_tic_tac_toe X < x_play_list_owin.txt`
        ii. `./np_tic_tac_toe O < o_play_list_owin.txt`
    c. Draw scenario: `x_play_list_draw.txt, o_play_list_draw.txt, to run this scenario:`
        i. `./np_tic_tac_toe X < x_play_list_draw.txt`
        ii. `./np_tic_tac_toe O < o_play_list_draw.txt`

Part3 – Signals

In Part3 of the attached source files, you'll find the following files: **signal.cc**:
This sample console application intercept **SIGINT** & **SIGUSR1** signals. SIGINT is generated when you press (CTRL C) on the keyboard will cause the application to end.

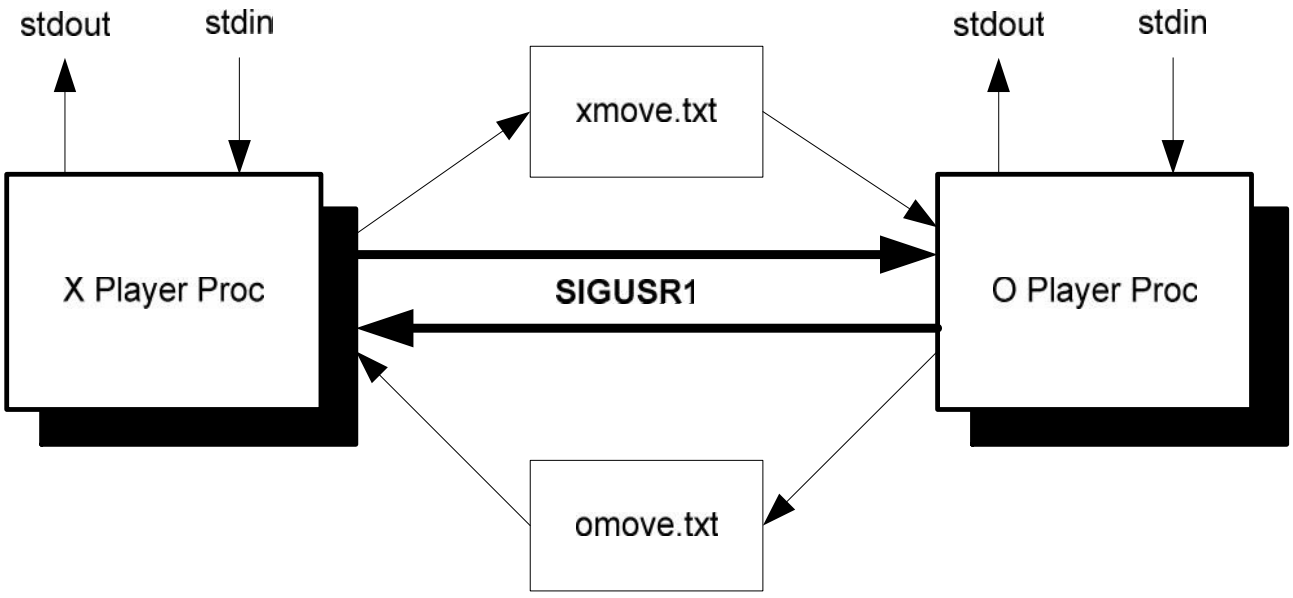
The application will first display its PID, you can use it to send it SIGUSR1 signals from another terminal. It will display a message that the signal was intercepted.

Signals can be considered as asynchronous software interrupts that will be served concurrently with the application. To compile & run the sample program:

1- gcc signal.cc –o signal
2- ./signal
3- From a different terminal try:
    a. Try sending SIGUSR1 to the signal process: **kill –s SIGUSR1 <PID>**, where PID is the signal process PID, notice it will acknowledge receipt of the signal
    b. Try sending SIGINT to the process: **kill –s SIGINT <PID>**, process should terminate. This is equivalent to using CTRL C from the process's terminal
    c. Try **kill <PID>**, process should terminate

Your task is to modify sig_tic_tac_toe.cc to allow 2 players to play the game from different terminals. The usage model is sig_tic_tac_toe X or sig_tic_tac_toe O, where [X|O] identifies the player. The application starts by writing its PID to a file **xmove.txt** (player X) or **omove.txt** (player O), then it tries to open the opponent's file to read the opponent's PID. It will be using the opponent's PID to send SIGUSR1 signals to the opponent's application notifying it that it's now its turn.
*Hint: use the c-command **kill (oponent_pid, SIGUSR1);** to trigger moves*

Refer to The Game section for explanation on how the game is played and the source files associated with it.



Use the following steps as guideline:

1- Include a game instance in the application.
2- Use the signal notification to set a global flag (e.g. opponent_done) to true.
3- Use the **convert2string()** function to convert the game state to a string to write in file
4- Use the **set_game_state(char *state)** to update the game state from the received message
5- player X will start, player O is going to wait – turn value will depend on the player
6- If the player's turn: get the player move, update the game, and write the move to the other player in **xmove.txt** (player X) or **omove.txt** (player O), close the file. Check if the game has ended, otherwise increment the turn, goto 7.
7- If not the player's turn: wait for signal (e.g. sleep(1) while opponent not done), read the opponent's updated game state from **xmove.txt** (player O) or **omove.txt** (player X), close the file, set the game state accordingly, display the updated game, and check if the game has ended. If the game did not end increment the game turn and goto 6.
8- A working program will allow two players to use different terminals to play the game and communicating the game movements across applications

Compile & run your application:

1- `g++ -I tic_tac_toe tic_tac_toe/tic_tac_toe.cc part3/sig_tic_tac_toe.cc -o sig_tic_tac_toe`
2- `make sure the files xmove & omove are deleted, necessary to synchronize the two processes`
3- `run the X player in a new terminal ./sig_tic_tac_toe X`
4- `run the O player in a new terminal ./sig_tic_tac_toe O`
5- `run the game`
6- `Verify your solution works using the 3 provided scenarios by piping the provided files to your application. The application should run autonomously to the expected outcome.`
    a. X win scenario: `x_play_list_xwin.txt, o_play_list_xwin.txt, to run this scenario:`
        i. `./sig_tic_tac_toe X < x_play_list_xwin.txt`
        ii. `./sig_tic_tac_toe O < o_play_list_xwin.txt`
    b. O win scenario: `x_play_list_owin.txt, o_play_list_owin.txt, to run this scenario:`
        i. `./sig_tic_tac_toe X < x_play_list_owin.txt`
        ii. `./sig_tic_tac_toe O < o_play_list_owin.txt`
    c. Draw scenario: `x_play_list_draw.txt, o_play_list_draw.txt, to run this scenario:`
        i. `./sig_tic_tac_toe X < x_play_list_draw.txt`
        ii. `./sig_tic_tac_toe O < o_play_list_draw.txt`

The game

Provided files **tic_tac_toe.cc**, **tic_tac_toe .h**, and **play_ tic_tac_toe .cc** provide the required basic functionality to a play a TIC-TAC-TOE game.

RULES FOR TIC-TAC-TOE:

- The game is played on a grid that's 3 squares by 3 squares.
- You are X, your friend (or the computer in this case) is O. Players take turns putting their marks in empty squares.
- The first player to get 3 of her marks in a row (up, down, across, or diagonally) is the winner.

The console application uses standard input (stdin) and standard output (stdout) to get users input and display the game board status. The tic_tac_toe class contains the following functions:

- **int make_move (int raw, int col, char player)**: check the player's move is legitimate, perform the move if legitimate
- **char game_result()**: return the game winner or 'd' if draw. If the game did not end returns '-'
- **void get_player_move(char player)**: prompts the player to enter a move. Moves are in row column coordinates, where rows are in {A, B, C}, and columns in {1, 2, 3}, e.g. B2
- **void display_game_board()**: displays the current state of the board on the console display
- **char* convert2string()**: converts the game state to a string (use it for message passing)
- **void set_game_state(char *state)**: set the game state from string (use it to update game from remote message)

The **play_tic_tac_toe** console application demonstrate the usage of the game's class within an application. Since we're **using c++** code, we're going to use **g++ or gcc compiler**. To compile and run the application:

1. **g++ -o play_tic_tac_toe.exe tic_tac_toe/tic_tac_toe.cc tic_tac_toe/play_tic_tac_toe.cc**
2. **./play_ tic_tac_toe**

Try it:

./play_tic_tac_toe.exe < tic_tac_toe/play_list.txt

```
  1 2 3
A - - -
B - - -
C - - -
Player X enter your move:
  1 2 3
A - - -
B - X -
C - - -
  1 2 3
A - - -
B - X -
C - - -
Player O enter your move:
  1 2 3
A O - -
B - X -
C - - -
  1 2 3
A O - -
B - X -
C - - -
Player X enter your move:
  1 2 3
A O X -
B - X -
C - - -
  1 2 3
A O X -
B - X -
C - - -
Player O enter your move:
  1 2 3
A O X O
B - X -
C - - -
  1 2 3
A O X O
B - X -
C - - -
Player X enter your move:
  1 2 3
A O X O
B - X -
C - X -
Game finished, result: X
```