Operating Systems

CS 550

The game *Paper, Scissors, Rock*

Concurrent Programming Project

Introduction

Multitasking and interprocess communication are two of the nice features of UNIX. It's nice to be able to say that you've actually programmed these features and that's what this project is all about!

Your concurrent programming project is to write a suite of programs that will run in parallel and interact to play the "*Paper, Scissors, Rock*" game. In this game, two players secretly choose either paper, scissors, or rock. They then reveal their choice. A referee decides who wins as follows:

1. **Rock beats scissors (by bluntening it)**
2. **Paper beats rock (by covering it)**
3. **Scissors beats paper (by cutting it)**
4. **Matching choices draw**

The winning player gets a point. In a draw, no points are awarded. Your program will simulate such a game, allowing the user to choose how many iterations are performed, observe the game, and see the final score.

You will create four files of **C** code, each of which plays a role (see next page):

1. **play.c - contains the code to start the game**
2. **referee.c - contains the referee's code**
3. **player.c - contains the code for each player**
4. **readstr.c - reads a string from a socket connection**

You will also create a "*makefile*", which will enable you to compile this suite of programs with ease.

To run the program, you will type:

```
% play turns
```

The play program will then execute the referee and two Player programs in parallel and wait until they're done.

Example

Here is a sample game:

```
% ./play 3
          Written by: Your Name
          Paper, Scissors, Rock: 3 iterations
                    Player 1: Ready
                    Player 2: Ready
          Go Players [1]
                    Player 1: Scissors
                    Player 2: Rock
                    Player 2 Wins
          Go Players [2]
                    Player 1: Paper
                    Player 2: Rock
                    Player 1 Wins
          Go Players [3]
                    Player 1: Paper
                    Player 2: Paper
                    Players Draw
          Final Score:
                    Player 1: 1
                    Player 2: 1
          Players Draw
     %
```

I want your output to look exactly like mine.

Deliverable:

1. Your program should print :"Written by YOUR NAME"
2. Your program is due by the end of the day, on due date
3. There is 10% penalty per day it is past due. It will not be accepted if it is more than 5 days late

Implementation:

Play

The *play* program is the coordinator and as such ***forks*** and ***execs*** the one *referee* and the two *players*. It then waits until all three are done. It must accept and check a command line parameter that is equal to the number of ***turns*** to run for and then pass it to the *referee* program.

Referee

The *referee* must prepare a *socket* that will listen for the connect request from both players. A socket is a communication line (like your telephone). Once you made connection to the socket (you dialed a number and the other party answered) you can exchange any information you want, as long as neither one of you disconnect. In our case it is like a conference call between the 2 players and the referee, except that players can't talk to each other directly.

Once a player successfully connects to the socket, sends a **"READY"** message to the referee. When both **"READY"** messages are received, referee sends each *player* a **"GO"** message, telling them to make a choice (**Paper**, **Scissors**, or **Rock**). Choices are read, by the referee, and scored. This Process is repeated until all of the iterations have been performed at which time the players are told to die (**"STOP"** message) and the *socket* is disposed of.

Player

Each *player* opens a *socket* and tells the referee that they're ready to begin the game by sending a **"READY"** message. Players keep on receiving messages from the referee. they close their socket and terminate when they receive the **"STOP"** message. They make a choice and send their choice to the referee when they receive the **"GO"** message.

Readstr

This utility routine is linked into the player and referee programs and used by the referee and both players. You can use other read and write routines if you prefer. It contains a function called *readstr()* that takes two parameters - a file descriptor and a pointer to a string. It reads one character at a time from the *socket* referred to by the file descriptor ***until a newline ('\n') character is received***. It places each character into the string except the newline, which is replaces with a NULL.

Include Files

In order to use sockets, you must include the following files at the top of each file that uses them:

```
#include <sys/types.h

#include <sys/socket.h
```

Socket Messages:

It's really tempting to make the socket messages one byte numbers rather than passing a complete string but I'm not going to let you do that! Here are the messages that I want your programs to use:

From Referee to Players:

"***GO***" - tells a player to make a choice
"***STOP***" - tells a player that the game is over

From Player to referee:

"***READY***" - informs the referee that a socket connection has been made
"***SCISSORS***" - Player chooses scissors
"***ROCK***" - Player chooses rock
"***PAPER***" - Player chooses paper

Useful System Calls:

The following system calls will be useful to you (please Use the LINUX "*man*" command to get more information on these commands):

```
int fork ()
int execv (char *path, char *argv [])
```

```
int execl (char *path, char *arg0, char *argl, …, 0)
int socket (int domain, int type, int protocol)
int bind (int fd, struct sockaddr *name, int len)
int listen (int fd, int max_connections)
int accept (int fd, struct sockaddr *fromname, int *len)
int connect (int fd, struct sockaddr *toname, int len)
int close (int fd)
int read (int fd, char *str, int len)
int write (int fd, char *str, int len)
perror (char *str)
int rand ()
srand (int seed)
int getpid ()
int unlink (char *path)
```

Separate Compilation:

Note that the *player* and *referee* programs must both use the routine called *readstr()*. It wouldn't make any sense to write it twice, one in each program, so the solution is to write the *readstr()* routine into its own file, called *readstr.c*, compile it separately, and then link it into each program that needs it. The **-c** option to the **C** compiler will compile a program without trying to make it into a complete executable. It can then be linked at a later stage with other o files:

```
% cc -c readstr.c (produces a file called readstr.o)

% cc -c player.c (produces a file called player.o)

% cc player.o readstr.o -o player (produces a complete program)
```

You won't have to perform this laborious process each time, as I'm going to tell you how to automate it using the *make* utility

Make:

There's a nice utility called *make* that I want you to read about in your LINUX book and ***man*** page. It makes separate compilation much easier to use. You must use it in your Project. Before you even start coding the **C**, produce a file called "*makefile*" in your project directory that looks like this:

Makefile:

Copy the following into a file called Makefile in your directory. Then, to recompile the suite of programs whenever you change a file, just type in the command:

% make

All the necessary recompilations and relinks are performed automatically!

Use tabs to indent the lines rather than spaces, or make will complain (which is really dumb in my opinion). *logfile* is a dummy file that ensures that all the programs are kept up-to-date. I recommend that you read about *make* to understand how it works, as it's used in industry a lot. Please note that your system may be different from mine so your makefile may be slightly different from mine.

Makefile:

```
#
# makefile for PSR
#
logfile : play referee player
      touch logfile
readstr.o : readstr.c
      gcc -c readstr.c
readstr : readstr.o
      gcc readstr.o -o readstr
play.o : play.c
      gcc -c play.c
play : play.o
      gcc play.o -o play
referee.o : referee.c
      gcc -c referee.c
referee : referee.o readstr.o
      gcc referee.o readstr.o -o referee
player.o : player.c
      gcc -c player.c
player : player.o readstr.o
      gcc player.o readstr.o -o player
```

Useful Hints:

- Socket Example
- Socket Man Page, just type “man socket” on your Linux machine