2016 System Programming Assignment 2

Announce: Wed, Nov 2, 2016
Due: 23:59 Tue, Nov 22, 2016

I. Problem Description

In this assignment, you are going to practice communicating among processes. The goal of this assignment is to practice multiple-process control via pipes and FIFOs, and to understand the use of fork(). We will propose a game so that you can enjoy this. The game is described as follows.

A competition is held by a judge, and requires four players to attend. Each player tells a number in {1, 3, 5} to the judge without knowing others'. After every player is done, the judge will announce that the player who says the number that is only said by him will get points equal to the number he says. For example, four players, A, B, C, and D attend this game. They tell the judge 5, 5, 3, and 1, separately. The judge will announce that C gets 3, and D gets 1 point while A and B get 0. While if A, B, C, D say 5, 5, 3, 3, separately, they all get 0 point, for each of them is not the only one that says his number.

One announcement is called a round. In each round (except the first round) the judge will tell each player the numbers said by all players in the previous round, which helps the player decide what number he is going to choose. A competition should hold 20 rounds, and each player will end up summing up his points.

There is a big_judge who has a list of players, and he has to let every four players compete. The big_judge will schedule all the competitions to judges. That is, if there are N players in total, then the big_judge has to arrange C(N, 4) competitions and each assign a judge. Note that there will be only limited judges, and a judge is only able to preside a competition at a time. That is, the big_judge can only assign a competition to an available judge, and not until the competition ends does that judge can take over another competition.

After a competition is completed, its judge should return the rankings of that competition to the big_judge. Different places refer to different scores. The big_judge should add the scores to the players' accumulative scores. The big_judge must wait until all competitions completed, calculate accumulative scores of all players, and rank all players according to their accumulative scores.

The story described above will be achieved via pipes and FIFOs. At the very beginning, the big_judge is executed. The big_judge should fork and execute each judge (the number of judges is limited), and talk to them via pipes. Each judge must create 5 FIFOs: a well-known FIFO to read, and 4 FIFOs to write to 4 players. Once a judge is assigned a competition, it forks four children as 4 players. It is your job to realize them.

II. Format for inputs and outputs

Three programs are required: "big_judge.c", "judge.c", and "player.c".

- big_judge.c (./big_judge [judge_num] [player_num])

It requires two arguments, the number of judges (1<=judge_num<=12) and the number of players (4<=player_num<=20).

At first, the big_judge should fork and execute the number of judges specified by the argument, with IDs from1 to judge_num. The big_judge must build pipes to communicate with each of them before executing them. The message coming from the judge would be the competition result presided by that judge, which will be described later in "judge.c" part.

After big_judge executes judges, the big_judge then has to distribute every 4 players to an available judge via pipe. The players are numbered from 1to player_num., so the message sending to the judges are of the format shown in the following

[p1_id] [p2_id] [p3_id] [p4_id]

If there is no available judge, then the big_judge waits until one of the judges returns the competition result, so that it can assign another competition to that judge. There will be C(player_num,4) competitions needed to be assigned. Ideally, the big_judge should make full use of available judges but not let any available judge idle.

The big_judge has to keep accumulative scores of all players. The accumulative scores of all players are initally set to 0. When the judge returns the result back to the big_judge, where the result is the rankings of the four players, the big_judge should add scores to the players' accumulative scores according to the rankings of them. The player in the first, second, third, and fourth place gets 3, 2, 1, and 0 addictive accumulative scores, accordingly.

After all competitions are done,the big_judge should send the string "-1 -1 -1 -1\n" to all judges, indicating that all competitions are done and judges can exit. Then, the big_judge outputs all players' ID sorted by their scores, from the highest to the lowest, separated by spaces. If two players have the same score, output the one with smaller ID first. For example, if there are 5 players and have accumulative scores 10, 20, 10, 40, and 20, the big_judge should output

```
4 40
2 20
5 20
1 10
3 10
```

- judge.c (./judge [judge_id])

It requires an argument, the ID of the judge. The judge should create a FIFO named judge[judge_id].FIFO, such as judge1.FIFO, to read responses from the players, and create four FIFOs named judge[judge_id]_A.FIFO, judge[judge_id]_B.FIFO, judge[judge_id]_C.FIFO, judge[judge_id]_D.FIFO, to write messages to the players in the competition held by this judge.

The judge should read from standard input, waiting for the big_judge to assign four players in. After knowing the players, the judge forks four child processes, run exec() to execute the player programs, and start 20 game rounds. The executable file of a player named player, and placed in the same directory containing big_judge and judge.

In each round, except for the first round, the judge tells every player the result of the previous round via specific FIFO to help them decide what number to tell in the next round. After giving out the message, the judge has to collect numbers coming from the four players. The format of these messages would be specified later in "player.c" part.

The judge can ask the players in turn. If a player does not respond for more than 3 seconds, the judge just assumes that the player doesn't respond any number **in this round and the following rounds**. That means, the judge should skip him and let him get 0 point in this round and the following rounds until the competition ends.

The judge accumulates each player's points. After 20 rounds, the judge should output the following to standard output:

```
[p1_id] [p1_rank]
[p2_id] [p2_rank]
[p3_id] [p3_rank]
[p4_id] [p4_rank]
```

where the ranks are ordered from 1 to 4. The player who gets the most points ranks 1, and the player who gets least points ranks 4. If multiple players get the same points in a competition, all of them occupy the **lowest** rank they would have.

For example, if four players (player 1 ~ player 4) get 20, 30, 20, and 0 points in a competition, then obviously, player 2 ranks 1 (and thus gets 3 accumulative score in big_judge), while player 1 and player 3 rank 3 (and gets 1 accumulative score in big_judge), and player 4 ranks 4 so gets 0 accumulative score.

After sending out the competition result, the judge shall wait until the big_judge assigns another competition, and do what is described above again. But when judge got "-1 -1 -1 -1\n"from the big_judge, it indicates that all competitions are done, so the judge should exit.

- player.c (./player [judge_id] [player_index] [random_key])

It requires three arguments. judge_id is the judge ID that holds the competition. player_index would be a character in {'A', 'B', 'C', 'D'}, the index this player has in this competition. random_key would be an integer in range [0, 65536), used in this player in this competition, and should be randomly generated unique for four players in the same competition. It is used to verify if a response really comes from that player.

Notice that the player index here is **NOT** the same as the player ID in judge/ big_judge. It just means which index the player has in this competition.

The player should open a FIFO named judge[judge_id]_[player_index].FIFO, which should be already created by the judge. The player reads messages from judge[judge_id]_[player_index].FIFO, such as judge1_A.FIFO, and writes responses to judge[judge_id].FIFO, such as judge1.FIFO.

In the first round, the player should first send its response to judge in the following format:
[player_index] [random_key] [number_choose]
indicating the index of this player, the random key given in the argument, and the number the player had chosen. After the first round, the message from judge would be in the following format:
[p_A_number] [p_B_number] [p_C_number] [p_D_number]
indicating the responses of all players in the previous round. Each [pi_number] would be in {0,1,3,5}. If the number is 0, it means that the player didn't make a response (because it has disconnected or his time has run out).

The above process will be repeated. The player must guarantee that it correctly gives out 20 responses, or the judge will punish it (by assuming the player not responds at all in the following rounds). The player should exit after it gives out 20 responses. The player would be executed again when competing in another competition.

III. Sample execution

```
$ ./big_judge 1 4
```

This will run 1 judge and 4 players. The big judge will fork and execute:

```
$ ./judge 1
```

The judge will create:

```
judge1.FIFO
judge1_A.FIFO
judge1_B.FIFO
judge1_C.FIFO
judge1_D.FIFO
```

The big_judge sends judge 1 (judge 1 reads from standard input):

```
1 2 3 4
```

The judge executes:

```
$ ./player 1 A 9
$ ./player 1 B 2013
$ ./player 1 C 10000
$ ./player 1 D 65535
```

Round 1, player 1 sends judge 1 through judge1.FIFO:

```
A 9 3
```

Round 2, judge 1 sends player 1 through judge1_A.FIFO:

```
3 5 5 1
```

Round 2, player 1 sends judge 1 again through judge1.FIFO:

```
A 9 5
```

The above process runs 20 rounds.
Judge 1 writes the result to standard output (sending to big_judge):

```
1 4
2 2
3 1
```

```
4 3
```
The big_judge reads the result, and does the calculation:

player 1's accumulative score + 0 = 0

player 2's accumulative score + 2 = 2

player 3's accumulative score + 3 = 3

player 4's accumulative score + 1 = 1

All competitions are over.

The big_judge sends judge 1:
```
-1 -1 -1 -1↵
```
The judge 1 terminates.

The big_judge outputs the result:
```
3 3
2 2
4 1
1 0
```

IV. Tasks and scoring

There are 6 subtasks in this assignment. By finishing all subtasks you earn the full 7 points.

1. If judge works fine (2 points)

 Read 4 player IDs from standard input, run the competition, and output to standard output correctly.

2. If judge detects time delay (1 point)

 Detects if a player disconnects or does not respond more than 3 seconds, and correctly gives him a penalty.

3. If player works fine (1 point)

 Reads from the specific FIFO the previous round result, and writes to the judge's FIFO the response.

4. If big_judge works fine (1 point)

 Correctly communicates with TA's judges and players, completes all required competitions, and gets right results. Tries to keep every judge busy.

5. If the processes correctly stops (1 point)

 Stops the executed processes, including big_judge and judge, player.

6. Completeness (1 point)

 Can run with TA's players and get right results.

V. Bonus

Your player will compete with others' players under TA's judges and big_judge. TA would randomly partition all student's player into 6 groups, and randomly assign player id to each player.

Bonus (0.5 point) would be given to top 2 in each group. TA would then get all top 2 players (total 12) and compete them with TA's player. Extra bonus (0.5 point) would be given to the overall top 2.

You don't need to implement the partition and multiple group part. This is just a reminder on how the bonus would work.

Please don't do things evil in your player, like opening others FIFO. We'll detect this and it would results in an automatically lose for your player.

VI. Notes

- Remember that every time you writes a message to a pipe or a FIFO, you should use fflush() to ensure the message being correctly passed out.
- For the judge, remember to clear the FIFO before a new competition begins, in case any player died in last competition and didn't read all message.

VII. Submission

Your assignment should be submitted to the course website by the due. Or you will receive penalty. At least five files should be included:

1. big_judge.c
2. judge.c
3. player.c
4. Makefile (as well as other *.c)
5. REAMDE.txt

Since we will directly run your Makefile, therefore you can modify the names for .c files, but Makefile should **compile your source into three executable files named big_judge, judge,and player**. Put your student ID (lowercase) and name in a comment at the first line of your source code, with the following format

/* STUDENT_ID NAME */

In README.txt, please briefly state how to compile your program, and anything you have done besides the basic functionality. These files should be put inside **a folder named with your student ID** and you should compress the folder into a .zip submission. Please do not use .rar or other file type.

The commands below will do the trick. Suppose your student ID is b02902000, and your name is 林聆凌:

```
$ mkdir b02902000
$ cp Makefile README.txt *.c b02902000
```

And finally, you should zip your folder and upload to ceiba.

The first line of big_judge.c, judge, player.c would be /*b02902000林聆凌 */

Please do **NOT** add executable files to the compressed file. Error in the submission file (such as files not in a directory named with your student ID, compiled binary not named big_judge, judge, and player, or so on) may cause deduction of your credits. (Really expensive, isn’t it? Please follow them carefully!)

VIII. Punishments

- Plagiarism

  Plagiarism is strictly prohibited.

- Late punishment

  Your credits of this assignment will be deducted 20% for each day of delay submission. (That is, you will lose all your credits on this assignment after 5 day delaying!)

  Even though your credits will be deducted for delaying, a late submission will still be much better than absence. Just remember one thing: there are System Programming courses to attend next morning.

If you have any question, refer to the slide to see how to contact us.