Assignment #3: Process IPC

ECE 650 – Fall 2016

Due by 11:59pm eastern time on Tuesday, March 22

General Instructions

1. You will work individually on this assignment.
2. The code for this assignment should be developed and tested in a UNIX-based environment.
3. You must follow this assignment spec carefully, and turn in everything that is asked (and in the proper formats, as described). Due to the large class size, this is required to make grading more efficient. Particularly for this assignment, much of the testing will be automated. If you do not follow exact instructions for your submission materials (file names, program output, etc.) points will be deducted.

Overview

In this assignment you will develop a pair of programs that model a simple game described below. The game is simple, but the assignment will give you hands-on practice with creating a multi-process application, processing command line arguments, setting up IPC channels between the processes (using UNIX fifos), and reading / writing information between processes across the IPC channels.

The game that will be modeled is called *hot potato*, in which there are some number of players who quickly toss a potato from one player to another until, at a random point, the game ends and the player holding the potato is "it". The object is to not be the one holding the potato at the end of the game. In this assignment, you will create a ring of "player" processes that will pass the potato around. Thus each player process has a left neighbor and a right neighbor. Also, there will be a "ringmaster" process that will start each game, report the results, and shut down the game.

To begin, the ringmaster creates a "potato" with some number of hops and sends the potato to a randomly selected player. Each time a player receives the potato, it will decrement the number of hops and append its identity to the potato. If the remaining number of hops is greater than zero, the player will randomly select a neighbor and send the potato to that neighbor. The game ends when the hop counter reaches zero. The player holding the potato sends it to the ringmaster, indicating the end of the game. The ringmaster prints a trace of the game to the screen (from the player identities that are appended to the potato), and shuts the game down (by sending a message to each player to indicate they may shut down as the game is over).

Each player will create several uni-directional IPC channels: to/from the player to the left, to/from the player to the right, and to/from the ringmaster. The potato can arrive on any of the three incoming channels. Commands and important information may also be received from the ringmaster. The ringmaster will create *N* IPC channels in each direction (a pair per player). At the end of the game, the ringmaster will receive the potato from the player who is "it".

The assignment is to create one ringmaster process and some number of player processes, then play a game and terminate all the processes gracefully. You may explicitly create each process from an interactive shell; however, the player processes must exit themselves in response to commands from the ringmaster.

IPC Mechanism

In this assignment, you will use UNIX fifos as the IPC channel for communication between the ringmaster and player processes. Your programs must use exactly the format described here. The ringmaster program is invoked as shown below:

```
ringmaster <number_of_players> <number_of_hops>
```

where `number_of_players` is greater than 1 and `number_of_hops` is greater than or equal to zero and less than or equal to 512 (make sure to validate your command line arguments!).

The player program is invoked as:

```
player <player_id>
```

where `player_id` is the ID of each player in the game. If there are 4 players, each player will have an ID of 0, 1, 2, or 3. The players are connected in the ring such that the left neighbor of player i is player i-1 and the right neighbor is player i+1. Player 0 is the right neighbor of player n).

Zero is a valid number of hops. In this case, the program must create the ring of processes. After the ring is created, the ringmaster shuts down the game.

Recall that UNIX fifos are based on files created within the file system. This allows fifos to be used for IPC between unrelated (i.e. no parent / child / sibling relationship) processes, unlike UNIX pipes. As such, fifos are often referred to as "named pipes". Your UNIX fifos used in this assignment should be created in the /tmp directory of your UNIX-based machine. The various fifos should be named as follows:

master_p<id> (e.g. master_p0): For communication from the master to each player
p<id>_master (e.g. p0_master): For communication from each player to the master

p<id>_p<id+1> (e.g. p0_p1): For communication from player 0 to player 1
p<id+1>_p<id> (e.g. p1_p0): For communication from player 1 to player 0

As UNIX fifos are uni-directional, we need a set of 2 fifos between each pair of communicating processes.

Resources:

Refer to the end of our IPC lecture for notes on the UNIX fifo IPC mechanism (e.g. mkfifo is used to create a UNIX fifo, and then two processes will "open" the file name used to create the fifo, just as opening a regular file. Typically one process will open the file with read permission O_RDONLY and one will open with write permission, O_WRONLY).

You may wish to use the unlink(filename) command to make sure that the UNIX fifo files are removed from the system at the end of the program (or at the start of the program, before mkfifo(…) calls are made.

You will also find that you will need to use the "select" call over a set of file descriptors from both the ringmaster and player processes in order to know when some information has been written to one of a set of the UNIX fifos.

Finally, you will need to create random numbers (e.g. between 0 to N). To do so, you may use the rand() call. Your code should first seed the random number generator:

```
srand( (unsigned int) time(NULL) );
```

Then you may generate a random number between 0 and N-1 using:

```
int random = rand() % N;
```

Output:

The programs you create must follow the description below precisely. If you deviate from what is expected, it will impact your grade.

The following describes **all** of the output of the `ringmaster` program. Do not have any other output.

Initially:

```
Potato Ringmaster
Players = <number>
Hops = <number>
```

Upon connection with a player (i.e. each player should send some initial message to the ringmaster to indicate that it is ready):

```
Player <number> is ready to play
```

When launching the potato to the first randomly chosen player:

```
All players present, sending potato to player <number>
```

When it gets the potato back (at the end of the game). The trace is a comma separated list of player numbers. No spaces or newlines in the list.

```
Trace of potato:
<n>,<n>, …
```

The following describes **all** of the output of the `player` program. Do not have any other output.

After receiving an initial message from the ringmaster to tell the player the total number of players in the game:

```
Connected as player <number> out of <number> total players
```

When forwarding the potato to another player:

```
Sending potato to <number>
```

When number of hops is reached:

```
I'm it
```

Detailed Submission Instructions

Your submission will include source code files and a Makefile. Your source code files should contain at least the following (skeleton or starter versions of these files are provided for you to get started):

1. **ringmaster.c** – The source code for your ringmaster program as described above.
2. **player.c** – The source code for your player program as described above.
3. **Makefile** – A makefile that will compile ringmaster.c and player.c into executable programs named ringmaster and player, respectively
4. **potato.h** – A simple file containing a sample potato structure that can be sent across UNIX fifos between players and between players and the ringmaster.

You will submit a single zip file named "hw3.zip" to your sakai dropbox location, e.g.:

```
zip hw3.zip ringmaster.c player.c potato.h Makefile
```