CST8234 – C Programming W19 (Assgn2)

Programming Exercise

The purpose of this assignment is to demonstrate that you can understand and can implement

- Multiple processes, with bidirectional communication
- Blocking and non-blocking I/O
- signals

As with any program, there are several ways to implement a set of requirements, e.g., you could, in theory, implement most (but not all) of the requirements of this assignment in a simple loop in a single process. ***However if you choose to not implement your solution using multiple processes, blocking and non-blocking I/O, and signals, you will receive ZERO on the assignment… with no partial marks!***

Statement of Problem

You are going to implement a "guess the number" game. I.e., the program will randomly generate a number. The user will be prompted to guess the number, and will be told if their guess is too low, too high, or correct. The user will then be re-prompted for their next guess.

The game ends when either the user guesses the correct number, or if the game times out (see "Timers" below).

Processes

You will fork your program to create two processes. The parent process will implement the user interface (i.e., it is responsible for getting input from the user, and providing feedback such as "your guess was too low"). The parent will send the user's guess to the child process through a pipe.

The "child" process is the brains of the game… it is responsible for randomly generating the number that the user has to guess, assessing whether the users' guesses are too high or too low, and whether the game has timed out. After each guess from the parent, the child will send its assessment (e.g., +1=too high or -1=too low or 0=correct) back to the parent process through a pipe.

I/O

The parent can use blocking reads, but the child must implement NON-blocking reads. This means that in the child, you must set up the file descriptor to use non-blocking I/O before the first 'read' (see the lecture notes).

This means that the child will immediately return from a 'read', so you can now implement an idle task and sleep for a fixed interval before the next read (important for implementing the Sanity timer, see below).

Sanity Timer

The child process will implement a "sanity" (a.k.a., "watchdog") timer, i.e., it will have a counter variable (initially set to 10), and once per second will decrement that counter (hint: do this in the idle task!)

The user has a few seconds (e.g., 10) to make a guess. If they fail to make a guess in that time (i.e., when the sanity counter decrements down to zero), the child process will use the 'kill' function to send a SIGUSR1 signal to the parent process. In the parent process's SIGUSR1 handler (registered with the 'signal' function), it will print out a message informing the user that they lost the game, and immediately exit.

Additionally, when the timer gets down to half the allotted time (e.g., 5 seconds), the child process will use the 'kill' function to send a SIGUSR2 signal to the parent process. In the parent's SIGUSR2 handler, it will print out a message informing the user that time is running out.

Whenever the child process receives a guess from the parent process, it will reset the sanity counter back to 10. I.e., the user has 10 seconds *per guess*.

Options

There are two optional command line arguments. One for the max numbers to guess (e.g., providing a value of 100 would require the player guess from 0-99), and the other is for the maximum number of seconds per guess (e.g., 10). The default values are 100 and 10 if the command line argument(s) are missing. Both POSIX and GNU long options must be supported, and a helpful usage message (requires that you support -h or --help). Whether you have the parent or child parse the options is your choice.

Sample Output

```
$ ./ass2.exe
Enter a number between 0-99 (you have 10 seconds to make a guess):
Are you still there?  Time is running out!
Sorry, you ran out of time!

$ ./ass2.exe --max=20 --timeout=5
Enter a number between 0-19 (you have 5 seconds to make a guess): 10
Your guess was too high

Enter a number between 0-19 (you have 5 seconds to make a guess): 5
Your guess was too high

Enter a number between 0-19 (you have 5 seconds to make a guess):
Are you still there?  Time is running out!
3
Your guess was too low

Enter a number between 0-19 (you have 5 seconds to make a guess): 4
Congratulations, you guessed the number (4) correctly!
```

User Feedback

All gameplay feedback is generated by the parent process. I.e., there should be no printf instructions in the child process. The only output that the child is allowed to generate are error messages in the case that the file descriptor couldn't be set to non-blocking, or if there was an error reading the P2C pipe.

If you opt to process the command line arguments in the child process, you'll definitely want to implement the command line parsing as a separate file (e.g., parse.c) so you don't violate the general rule about no printf's in the child process.

Global variables

No global variables are to be used in this assignment.

Miscellaneous Hints

Compiler Warnings

Note that a signal handler takes an integer parameter, which you won't need to use. This will cause compiler warnings with the compiler flags that we've been using to-date. Similarly, if you're developing on Cygwin, you may find that you have an "implicit declaration" warning relating to the 'kill' function.

You can suppress those warnings by appending "`-D_POSIX_SOURCE -Wno-unused-parameter`" to the CFLAGS line of your makefile.

Binary vs Text Messages

The C functions 'read' and 'write' actually just transfer raw bytes. You just tell 'write' where your data lives (i.e., its address), and how many bytes to transfer. Similarly, 'read' needs to be told where the data is to be put (i.e., its address), and the maximum number of bytes to expect.

But whether those bytes are ASCII characters, shorts, longs, or entire data structures is your choice!

E.g., to send text, one process could implement the following

```
char buf1[BUF_MAX];
stcpy( buf1, "Hello" );                  // assume that BUF_MAX is large enough
write( fdWrite, buf1, strlen(buf1)+1 ); // also need to send the null terminator
```

whereas the other process would have to implement the following to read the data as a string

```
char buf2[BUF_MAX];
read( fdRead, buf2, BUF_MAX );
printf( "I received '%s'\n", buf2 );
```

Alternatively, we could send binary data by having one process implement the following:

```
short  numStudents = 25;
write( fdWrite, &numStudents, sizeof(numStudents) );
```

whereas the other process would have implement the following to read the data as a short

```
int nValue;
read( fdRead, &nValue, sizeof(nValue) );
printf( "I received '%d'\n", nValue );
```

Of course, you have to make sure that your sizes match in both the 'read' and 'write' operations. E.g., writing a 'long' but only reading a 'short' will leave two unread bytes in the pipe.

Whether your implementation transfers strings or integers between processes is your choice.

Files

I implemented my 'main' function in main.c, which sets up the pipes and forks the processes and then calls either 'main_parent' or 'main_child' which are implemented in files called parent.c and child.c, respectively. The main code also correctly closes the pipes if/when 'main_parent' and/or 'main_child' return.

The function 'main_parent' implements a loop in which the user is repeatedly prompted for guesses, and the function 'main_child' implements the loop in which the data from the parent is processed.

All additional functions pertaining to the child functionality (e.g., idle task) can then be put in child.c, and likewise the parent-specific functionality (e.g., signal handlers) can be put in parent.c.

You may want to put the command line argument parsing in a separate file.

You are free to come up with other logical arrangements, but keep in mind that putting all your functionality in one file will get you very low marks for style and efficiency. At this point in the term, you should be able to come up with a sensible distribution of functionality across your files.

Submission

When you are complete submit your programs to Brightspace.

But… before you do, please check that you've satisfied the following submission requirements

1. Did you confirm that your zipped submission is a ".zip" file?
2. Did you zip up your submission such that it **contains a folder** with your partner's *user names* (e.g., "smit9112_sing0003/")? I.e., if you open up your own zipped submission, you should see a *folder* called (for example) "smit9112_jone0001". If you just see *file(s)*, you've done it wrong, and you'll need to go to the *parent* folder, and try zipping your lab folder.
3. Did you remove all the unnecessary files from the folder contained in your zipped submission? I.e., you open up your own zipped submission, and click on the folder called (for example) "smit9112_sing0003", you should see *just your makefile and your source files, and include files.*
4. Is your makefile actually called 'makefile' or 'Makefile' (without any filename extensions?)