Inter-Process Communication (IPC) using Pipes in C

Overview

- This homework will count as 8% of your final course grade.
- This homework is to be completed **individually**. Do not share your code with anyone else.
- You **must** use C for this homework assignment.
- Your program **must** successfully compile and run on Ubuntu v14.04.3 LTS.
- Your program **must** successfully compile via `gcc` with absolutely no warning messages when the `-Wall` (i.e., warn all) compiler option is used. We will also use `-Werror`.

Homework Specifications

In this second homework, you will use C to implement a fork-based calculator program. The goal is to work with pipes and processes, i.e., inter-process communication (IPC). Further, you must parallelize all processing to the extent possible.

Overall, your program will read an input file that specifies a set of calculations to be made using a Scheme/LISP-like format. More specifically, your program will call `fork()` to create child processes to perform each calculation, thus constructing a process tree that represents the fully parsed mathematical expression.

The input file is specified on the command-line as the first argument and contains exactly one expression to be evaluated. For this homework, you must support addition, subtraction, multiplication, and division, adhering to the standard mathematical order of operations as necessary. Note that each of these operations must have at least two operands, with operands to be processed from left to right.

Any line beginning with a `#` character is ignored (i.e., these lines are comments). Further, all blank lines are to be ignored, including lines containing only whitespace characters.

Example expressions are shown below.

```
# EXAMPLE 1: 7 * 8
#
(* 7 8)
```

```
# EXAMPLE 2: 5 + 6 + 7
#
(+ 5 6 7)
```

```
# EXAMPLE 3: 5 + 6 + 7 * -8
#
(+ 5 6 (* 7 -8))

# EXAMPLE 5: 18 - 14 / 3
#
(- 18 (/ 14 3))
```

```
# EXAMPLE 4: 12 - 3 - 9
#
(- 12 3 9)

# EXAMPLE 6: 99 * (88 + 77 - 66) * 55
#
(* 99 (+ 88 (- 77 66)) 55)
```

The six examples shown above have corresonding "parse trees" shown below.

```
EXAMPLE 1:    *
             / \
            7   8

EXAMPLE 2:     +
              /|\
             / | \
            5  6  7

EXAMPLE 3:     +
              /|\
             / | \
            5  6  *
                 / \
                7  -8

EXAMPLE 4:      -
               /|\
              / | \
             12  3  9
```

```
EXAMPLE 5:     -
              / \
             18   /
                 / \
                14   3

EXAMPLE 6:     *
               /|\
              / | \
             99  |  55
                 +
                / \
               /   \
              88     -
                    / \
                   /   \
                  77     66
```

For the above parse tree diagrams, your program must use `fork()` to create child processes that match these trees. Here, the edges between nodes are pipes in which each child process conveys the result of its intermediate calculation.

A simple algorithm here is to fork a child process for each operand encountered. In the `(* 7 8)` example, the parent process calls `fork()` twice, i.e., for the `7` and `8` operands.

In the `(- 18 (/ 14 3))` example, the parent process also calls `fork()` twice, this time for the `18` and `(/ 14 3)` operands. The second child process then calls `fork()` twice, i.e., for the `14` and `3` operands (yielding `4` as a result).

Note that you can assume all values given and all values calculated will be integers. Therefore, use integer division (i.e., truncate any digits after the decimal point).

Required Output

When you execute your program, each parent process must display a message when it first parses an operator (i.e., starts an operation). Each child process must display a message when it sends an intermediate result back to its parent via a pipe. Further, when a parent process completes its given calculation, it must display a message and send the intermediate result back to its parent via a pipe. Finally, the top-level parent process must display the final answer for the given expression.

For the example `(+ 5 6 (* 7 -8))` expression, your program must display the output shown below, though note that process IDs will likely be different and the order of some lines of output could be different on different runs, too. As noted above, parallelize to the extent possible.

```
PROCESS 31091: Starting "+" operation
PROCESS 31092: Sending "5" on pipe to parent
PROCESS 31093: Sending "6" on pipe to parent
PROCESS 31094: Starting "*" operation
PROCESS 31095: Sending "7" on pipe to parent
PROCESS 31096: Sending "-8" on pipe to parent
PROCESS 31094: Processed "(* 7 -8)"; sending "-56" on pipe to parent
PROCESS 31091: Processed "(+ 5 6 (* 7 -8))"; final answer is "-45"
```

Handling Errors

Your program must ensure that the correct number of command-line arguments are included. If not, display an error message and usage information as follows to `stderr`:

```
ERROR: Invalid arguments
USAGE: ./a.out <input-file>
```

If system calls fail, use `perror()` to display the appropriate error message(s) to `stderr`, then exit the program by returning `EXIT_FAILURE`.

If given an invalid expression, display an error message using the format shown below. Display this to `stdout`. For example, given `(* 3)`, display the following:

```
PROCESS 6431: Starting "*" operation
PROCESS 6431: ERROR: not enough operands
```

As another example error, given `(QRST 5 6)`, display the following:

```
PROCESS 7330: ERROR: unknown "QRST" operator
```

In both of the above cases, the problematic process should exit with no additional output.

Upon return from `wait()`, if the child process terminated due to a signal, display the following error message to `stdout`:

```
PARENT: child <child-pid> terminated abnormally
```

If the child process terminated normally, but its exit status was not `0`, display the following error message to `stdout`:

```
PARENT: child <child-pid> terminated with nonzero exit status <exit-status>
```

Submission Instructions

To be determined....