CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Homework 2 (document version 1.0)
Process Creation and Inter-Process Communication (IPC) in C

Overview

- This homework is due by 11:59:59 PM on Tuesday, October 3, 2017.
- 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, and your code **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`, which will treat all warnings as critical errors.
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v16.04.3 LTS. Note that the `gcc` compiler is version 5.4.0 (`Ubuntu 5.4.0-6ubuntu1~16.04.4`).

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).

A key requirement here is that 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 * 80
#
(* 7 80)
```

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

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

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

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

```
# 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   80

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, each node represents a process, while each edge represents a separate pipe by which each child process conveys the result of its intermediate calculation back to its parent.

The algorithm to use here is to fork a child process for each operand encountered. In the `(* 7 80)` example, the parent process calls `fork()` twice, i.e., for the `7` and `80` 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). Then, each child process must display a message when it sends an intermediate result back to its parent via its 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 very likely be different and the order of some lines of output could be different on different runs, too. As noted above, you must parallelize all of this to the extent possible.

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

Note that in the example above, interleaving of output may occur in child processes handling each operand.

Handling Command-Line Argument Errors

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

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

If a system call fails, use `perror()` to display the appropriate error message on `stderr`, then exit the program by returning `EXIT_FAILURE`.

Handling Invalid Expressions

If given an invalid expression in which there is only one operand, display an error message using the format shown below. Note that the first operand would still cause a child process to be created. Display this error on `stdout`. For example, given `(* 3)`, display the following:

```
PID 6431: My expression is "(* 3)"
PID 6431: Starting "*" operation
PID 6431: ERROR: not enough operands; exiting
PID 6432: My expression is "3"
PID 6432: Sending "3" on pipe to parent
```

Further, if you encounter an invalid operator, no child processes are created. As an example, given `(QRST 5 6)`, display the following on `stdout`:

```
PID 7330: My expression is "(QRST 5 6)"
PID 7330: ERROR: unknown "QRST" operator; exiting
```

Division by zero is not allowed, thus if `0` appears as anything but the first operand to division, report an error. More specifically, display an error on `stdout`. For example, given `(/ 33 0)`, display the following:

```
PID 8188: My expression is "(/ 33 0)"
PID 8188: Starting "/" operation
PID 8188: ERROR: division by zero is not allowed; exiting
PID 8189: My expression is "33"
PID 8189: Sending "33" on pipe to parent
```

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

Handling Process Termination Errors

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

```
PID 7220: child <child-pid> terminated abnormally
```

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

```
PID 11923: child <child-pid> terminated with nonzero exit status <exit-status>
```

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server. The specific URL is on the course website.

Note that this assignment will be available on Submitty a few days before the due date. Please do not ask on Piazza when Submitty will be available, as you should perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, as discussed in class (on 9/7), output to standard output (`stdout`) is buffered. To ensure buffered output is properly flushed to a file for grading on Submitty, use `fflush()` after every set of `printf()` statements, as follows:

```
printf( ... );     /* print something out to stdout */
fflush( stdout );  /* make sure that the output is sent to a */
                   /*  redirected output file, if specified  */
```

Second, also discussed in class (on 8/31), use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!" );
    fflush( stdout );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE homework2.c
```

Finally, be sure you use `fork()`, `wait()/waitpid()`, and `pipe()` system calls to implement this homework assignment.