Breakout / Lab 12

REPL: Read–Eval–Print Loop

CSCI 1730 – Spring 2017

Problem / Exercise

A read–eval–print loop (REPL), also known as an interactive top-level or shell, is a simple, interactive computer programming environment that takes single user inputs (i.e. single expressions), evaluates them, and returns the result to the user. For this breakout lab, you need to design a REPL that can print out information about a job pipeline that a user has described as input. You are **NOT** allowed to use `execve(2)` (or the `exec(3)` family of functions), `system(3)`, `popen(3)`, and `pclose(3)` in your implementation. Whenever possible, use low-level system calls and unbuffered I/O.

Here are some examples (first line is a "`repl$ `" prompt that your program will literally print to standard output plus user input that your program will read in from the user; subsequent lines are output):

- `repl$ cat file.txt`

```
Job STDIN  = STDIN_FILENO
Job STDOUT = STDOUT_FILENO
Job STDERR = STDERR_FILENO

0 pipe(s)
1 process(es)

Process 0 argv:
0: cat
1: file.txt
```

- `repl$ cat file.txt | less`

```
Job STDIN  = STDIN_FILENO
Job STDOUT = STDOUT_FILENO
Job STDERR = STDERR_FILENO

1 pipe(s)
2 process(es)

Process 0 argv:
0: cat
1: file.txt

Process 1 argv:
0: less
```

- `repl$ cat file.txt | grep // > out.txt`

```
Job STDIN  = STDIN_FILENO
Job STDOUT = out.txt (truncate)
Job STDERR = STDERR_FILENO

1 pipe(s)
2 process(es)

Process 0 argv:
0: cat
1: file.txt

Process 1 argv:
0: grep
1: //
```

- `repl$ cat | grep // | less < in.txt >> out.txt`

```
Job STDIN  = in.txt
Job STDOUT = out.txt (append)
Job STDERR = STDERR_FILENO

2 pipe(s)
3 process(es)

Process 0 argv:
0: cat

Process 1 argv:
0: grep
1: //

Process 2 argv:
0: less
```

- `repl$ cat f1.txt f2.txt > out.txt e>> log.txt`

```
Job STDIN  = STDIN_FILENO
Job STDOUT = out.txt (truncate)
Job STDERR = log.txt (append)

0 pipe(s)
1 process(es)

Process 0 argv:
0: cat
1: f1.txt
2: f2.txt
```

- `repl$ cat file1 file2 file3`

```
Job STDIN  = STDIN_FILENO
Job STDOUT = STDOUT_FILENO
Job STDERR = STDERR_FILENO

0 pipe(s)
1 process(es)

Process 0 argv:
0: cat
1: file1
2: file2
3: file3
```

- `repl$ echo "my \"cool\" shell" | less`

```
Job STDIN  = STDIN_FILENO
Job STDOUT = STDOUT_FILENO
Job STDERR = STDERR_FILENO

1 pipe(s)
2 process(es)

Process 0 argv:
0: echo
1: my "cool" shell

Process 1 argv:
0: less
```

1 Group Brainstorm

Breakup into groups based on your seating and brainstorm about how to solve the problem or exercise. Make sure everyone understands the problem, and sketch out potential ways to move towards a solution. Perhaps something that was discussed during lecture might be useful?

2 Submit Individual Brainstorm

Login to eLC and submit a version of your group's brainstorm, written in your own words. You may add additional information if you want. You need to write enough in order to convince the grader that you understand the problem or exercise and that you have a plan for moving forward towards a solution. Please include the last names of the other people in your group in your submission. The brainstorm submission should be available on eLC in your assignment dropbox. We prefer that you submit your individual brainstorms before the end of your breakout period, however, you generally have until 11:55 PM on Friday during the week of your breakout (as indicated on eLC) to submit them. **NOTE:** Submissions that do not include an individual brainstorm will not be graded.

3 Submission

Before your next breakout lab session, you need to submit your code. You will still be submitting your project via nike. Make sure your work is on `nike.cs.uga.edu` in a directory called `LastName-FirstName-lab12`. To submit your lab and email yourself a copy, execute the following command from within the parent directory:

```
$ submit LastName-FirstName-lab12 cs1730a
$ tar zcvf LastName-FirstName-lab12.tar.gz LastName-FirstName-lab12
$ mutt -s "lab12" -a LastName-FirstName-lab12.tar.gz -- your@email.com < /dev/null
```

4 Some Nonfunctional Requirements

Your submission needs to satisfy the following nonfunctional requirements:

- **Directory Setup:** Make sure that all of your files are in a directory called `LastName-FirstName-lab12`, where `LastName` and `FirstName` are replaced with your actual last name and first name, respectively.
- **Libraries:** You are allowed to use any of the C or C++ standard libraries. When reading or writing to a file are concerned, you need to use low-level calls to `read(2)` and `write(2)` and related functions. You are NOT allowed to use the following system calls in any of your implementations: `fork(2)`, `execve(2)`, `exec(3)`, `popen(3)`, and `system(3)` (or related functions).
- **Unbuffered Output:** Whenever possible, program output should be unbuffered. The best way to guarantee that output is unbuffered (i.e., characters at the destination as soon as possible) is to directly call `write(2)`. If you are using `printf(3)`, you should disable output buffering using `setvbuf(3)`. If you are using C++ output streams (e.g., `cout`), then you should disable output buffering using `setf` and `unitbuf` .
- **Documentation:** All classes, structs, and functions must be documented using Javadoc (or Doxygen) style comments. Use inline documentation, as needed, to explain ambiguous or tricky parts of your code.
- `Makefile` **File:** You need to include a `Makefile`. Your `Makefile` needs to compile and link separately. That is, make sure that your `Makefile` is setup so that your `.cpp` files each compile to individual `.o` files. This is very important.
- **Standards & Flags:** Make sure that when you compile, you pass the following options to `g++` in addition to the `-c` option:

  ```
  -Wall -std=c++14 -g -O0 -pedantic-errors
  ```

  Other compiler/linker options may be needed in addition to the ones mentioned above. The expectation is that the grader should be able to type `make clean` and `make` in the following to clean and compile/link your submission, respectively.
- `README` **File:** Make sure to include a `README` file that includes the following information presented in a reasonably formatted way:
  - Your Name and 810/811#
  - Instructions on how to compile and run your program.
  - Reflection

  Make sure that each line in your `README` file does not exceed 80 characters. Do not assume line-wrapping. Please manually insert a line break if a line exceeds 80 characters.
- **Compiler Warnings:** Since you should be compiling with both the `-Wall` and `-pedantic-errors` options, your code is expected to compile without `g++` issuing any warnings.
- **Memory Leaks:** Since this assignment may make use of dynamic memory allocation, you are expected to ensure that your project implementation does not result in any memory leaks. We will test for memory leaks using the `valgrind` utility.