CS 252: Systems Programming
Fall 2024
Project 3: Shell Interpreter

Prof. Turkstra

Part 1: Monday, September 23 11:59pm
Part 2: Wednesday, October 2 11:59pm
Part3: Friday, October 11 11:59pm

Goals

The goal of this project is to build a shell interpreter which combines behavior from common shells, including Bash and csh. Some skeleton code has been provided, so you will not be starting from scratch.

Deadlines

The project consists of three parts.

- The deadline for part 1 is **Monday, September 23 11:59pm**.
- The deadline for part 2 is **Wednesday, October 2 11:59pm**.
- The deadline for part 3 is **Friday, October 11 11:59pm**.

All extra credit parts are due with the final submission.

Testing

Much of your shell will be graded using automatic testing, so make sure that your shell passes the provided tests. Your grade for this project will partially depend on the number of tests that pass.

See `proj3/test-shell/README` for an explanation of how to run the tests. All tests can be run from the `test-shell` directory. The tests will also give you an **estimated** grade. This grade is just an approximation. Other tests which are not provided will be used as well during official grading.

To run the tests only for a specific part, you can type:

```
./testall partN
```

Where N = 1, 2, or 3.

`./testall` with no arguments runs all of the tests.

1 The Assignment - Part 1

Login to a CS department machine (a lab machine or `data.cs.purdue.edu`), navigate to your preferred directory, and run

```
git clone ~cs252/repos/$USER/proj3.git
cd proj3
```

Build the shell by typing `make`, and start it by typing `./shell`. Type in some commands, for example:

```
ls -al
ls -al aaa bbb > out
```

At this point, the shell does not have much implemented; notice what happens if you try to use some shell features that you used in Project 1. For example, try redirecting input or editing a typo in a command.

1.1 Parsing and Executing Commands

To begin, you will write a scanner and parser for your shell using Lex and Yacc. Look through the skeleton code and understand how it works. Lex will produce tokens by matching regular expressions; Yacc will then match these tokens to grammatical expressions and execute the corresponding code.

The file `command.h` implements a data structure that represents a shell command. A shell command can consist of multiple parts; the struct `single_command` implements an argument list for a single command (i.e. a command of the form `mycmd arg1 arg2 arg3`). Then, when pipes are used, a command will be composed of multiple `single_command`'s. The struct `command` represents such a list of simple commands (which can also be just one `single_command`). Additionally, `command` has fields which allow the user to specify the files to use for input, output, and error redirection.

Look through the `Makefile` to understand how the program is built. In particular, note how C source code files are generated from your `shell.l` and `shell.y` files, which are then compiled with the rest of your source code.

1.1.1 Accepting more complex commands

You will use Lex and Yacc to implement the grammar of your shell. See Lex and Yacc Primer for more details, and here and here for tutorials on Lex and Yacc.

The skeleton shell initially implements only a very limited grammar:

```
cmd [arg]* [> filename]
```

The first objective for Part 1 is to modify `shell.l` and `shell.y` to support a more complex grammar:

```
cmd [arg]* [ | cmd [arg]* ]* [ [> filename] [< filename] [2> filename]
[ >& filename] [>> filename] [>>& filename] ]* [&]
```

Fill in the necessary code in `shell.l` and `shell.y` to fill in the `command` struct. Make sure that the `command` struct is printed correctly.

Some example commands to test with are included below:

```
ls
ls -al
ls -al aaa bbb cc
ls -al aaa bbb cc > outfile
ls | cat | grep
ls | cat | grep > out < inp
ls aaaa | grep cccc | grep jjjj ssss dfdffdf
ls aaaa | grep cccc | grep jjjj ssss dfdffdf >& out < in
httpd &
ls aaaa | grep cccc | grep jjjj ssss dfdffdf >>& out < in
```

1.2 Executing Commands

Now you will implement the execution of single commands, IO redirection, piping, and backgrounding of processes.

1.2.1 Single command process creation and execution

For each single command, create a new process using `fork()` and call `execvp()` to execute the corresponding executable.

If the `command` is not set to execute in the background, then your shell will have to wait for the last single command to finish using `waitpid()`. Refer to the `man` pages of these functions for information on their arguments and return values.

Additionally, we have provided the file `ls_grep.c` as an example of a program that creates processes and performs redirection. This is an unrolled and program-specific example of the loop your shell should implement.

After you have completed this part, you should be able to execute commands such as:

```
ls -al ls -al /etc &
```

1.2.2 File redirection

If the `command` specifies files for IO redirection (of input, output, or error), then create those files as necessary. To change the file descriptors to point to the specified files, you will need to use `dup2()`. Note that file descriptors 0, 1, and 2 correspond to standard input, output, and error respectively. See the example redirection in`ls_grep.c`.

After you have completed this part, you should be able to execute commands such as:

```
ls -al > out
cat -q cat 2> dog
ls
cat out
ls /tttt >& err
cat err
cat < out
cat < out > out2
cat out2
```

```
ls /tt >>& out2
```

Note:

- `2>` redirects stderr to the specified file
- `>&` redirects both stdout and stderr to the specified file
- `>>` appends stdout to the specified file
- `>>&` appends both stdout and stderr to the specified file

1.2.3 Pipes

Pipes are an interface that allow for inter-process communication. They have two ends, one for reading and one for writing. Data which is written into the write end of the pipe is buffered until it is read from the read end by another process.

Use `pipe()` to create a pipe that will redirect the output of a single command to the input of the next single command. You will need to use `dup2()` again to handle the redirection. See the example piping in `ls_grep.c`.

After you have completed this part, you should be able to execute commands such as:

```
ls -al | grep command
ls -al | grep command | grep command.o
ls -al | grep command
ls -al | grep command | grep command.o > out
cat out
```

1.2.4 isatty()

When your shell uses a file as standard input your shell should not print a prompt. Use the function `isatty()` to find out if the input comes from a file or from a terminal.

Note: Due to how the automated tests are built, you will need to complete this portion of part 1 before your shell will pass any of the automated tests.

1.2.5 Exit

Implement a special command called `exit` which will exit the shell when run. Note that `exit` should not cause a new process to be created; it should be picked up by your shell during parsing and cause your shell to `exit`.

```
myshell> exit
bash$
```

1.3 Submission

The deadline for Part 1 is **Monday, September 23 11:59pm**.

To submit your program for grading:

1. Login to a CS department machine
2. Navigate to your `proj3` directory
3. Run `make clean`
4. Run `make` to check that your shell builds correctly
5. Run `make submit_part1`

These Standard Rules Apply

- Follow the coding standards as specified on the course website.
- Your shell must compile and run on the lab Linux machines.
- Do not look at anyone else's source code. Do not work with any other students.

2 Adding Features - Part 2

In this part, you will add features that make your shell more useful.

2.1 ctrl-C

In `bash`, and other common shells, you can press ctrl-C to interrupt, and usually terminate, a running command. This can be especially helpful if a command you are running takes longer than expected to finish or if you are running a buggy program that falls into an infinite loop. This is accomplished by detecting the key combination and generating a `SIGINT` signal which is passed on to the currently executing process.

If ctrl-C is typed when no command is running, the current prompt is discarded and a fresh prompt is printed. As-is, your shell will simply exit when ctrl-C is typed and no command is running.

Make your shell behave as `bash` does with respect to ctrl-C. See crtl_c.c for an example of detecting and ignoring a `SIGINT` signal. Also see the `man` page for `sigaction()`.

2.2 Zombie Elimination

Try running the following set of commands in the shell you have written:

```
ls &
ls &
ls &
ls &
/bin/ps -u <your-login> | grep defu
```

The last command shows all processes that show up as "defu" for ("defunct"). Such processes are called *zombie processes*: they no longer run, but wait for the parent to acknowledge that they have finished. Notice that each of the processes that are created in the background become *zombie processes*.

Despite the entertainment value these processes have with regard to tormenting Purdue IT (PIT), it is better from a proper implementation standpoint that they are cleaned up.

To cleanup these processes you will have to set up another signal handler, similar to the one you used for ctrl-C, to catch the `SIGCHLD` signals that are sent to the parent when a child process finishes. The signal handler should call `waitpid()` to cleanup the zombie child.

Take a look at the man pages for the `waitpid()` and `sigaction()` system calls.

Finally, the shell should also print the process ID of the child when a process in the background exits in the form "`[PID] exited`."

2.3 Quotes

Add support for quotes in your shell. It should be possible to pass arguments with spaces if they are surrounded by quotes. For example:

```
myshell> ls "command.cc Makefile"
```

```
command.cc Makefile not found
```

Here, "`command.cc Makefile`" is only one argument. You will need to remove the quotes before using the argument they contain.

Note: wildcard expansion will not be expected inside quotes in the next part of the project.

2.4 Escaping

Allow the escape character. Any character can be part of an argument if it comes immediately after \, including special characters such as quotation marks (`""`) and an ampersand (`&`). For example:

```
myshell> echo \"Hello between quotes\"
"Hello between quotes"
myshell> echo this is an ampersand \&
this is an ampersand &
```

2.5 Builtin Functions

Certain commands run in `bash` do not actually correspond to programs. These commands are detected by the shell during parsing to carry out certain special functions. Implement the following builtin commands:

`printenv`	Prints the environment variables of the shell. The environment variables of a process are stored in the variable `char **environ`; a null-terminated array of strings. Refer to the `man` page for `environ`.
`setenv A B`	Sets the environment variable `A` to value `B`. See article.
`unsetenv A`	Un-sets environment variable `A`
`source A`	Runs file `A` line-by-line, as though it were being typing into the shell by a user. See Multiple Input Buffers
`cd A`	Changed the current directory to `A`. If no directory is specified, default to the home directory. See the `man` page for `chdir()`.

You should be able to use built-ins like any other command (e.g. grep, cat, etc.), including with redirection and piping.

2.6 ".shellrc"

When your shell starts, it should attempt to do the equivalent of running "`source .shellrc`".

2.7 Submission

The deadline for Part 2 is **Wednesday, October 2 11:59pm**.

To submit your program for grading:

1. Login to a CS department machine
2. Navigate to your `proj3` directory
3. Run `make clean`
4. Run `make` to check that your shell builds correctly
5. Run `make submit_part2`

These Standard Rules Apply

- Follow the coding standards as specified on the course website.
- Your shell must compile and run on the lab Linux machines.
- Do not look at anyone else's source code. Do not work with any other students.

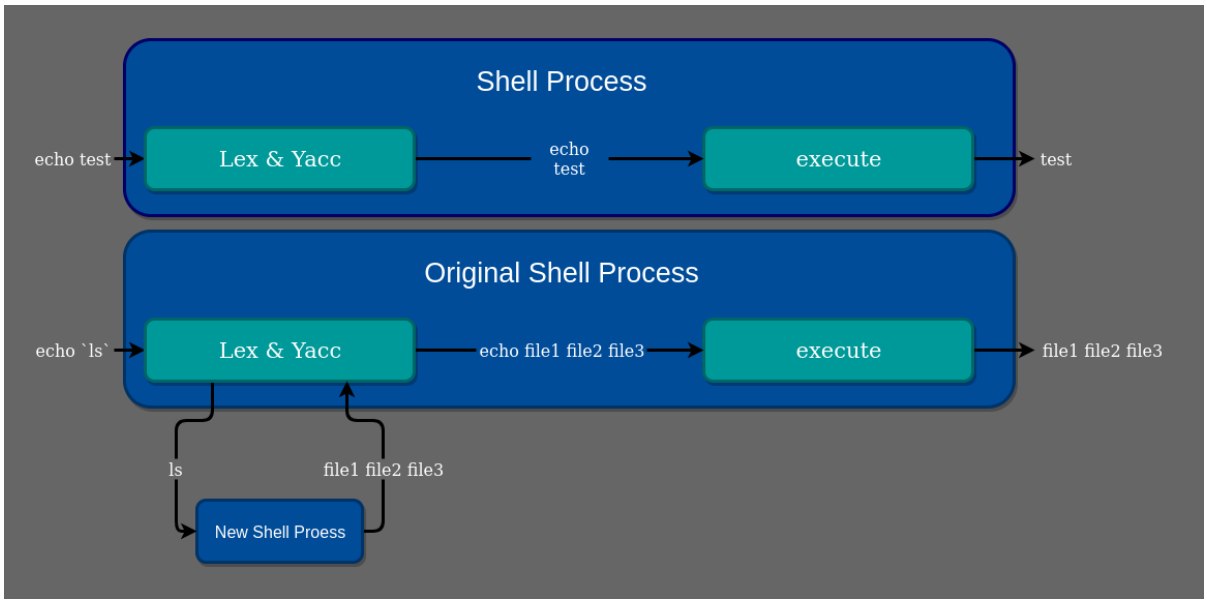
3 Moar Features - Part 3

The final part of this project involves adding a few more major usability features to your shell. You will add support for subshell creation as well as some expansions including wildcards. Finally, support for a line editor will also be added as well as command history search.

3.1 Subshells

Sometimes one needs to use the output of a command or set of commands elsewhere in the shell. Subshells permit this execution and capture of output. Any argument of the form `$(command and args)` will be processed by another shell (the subshell) which is executed as a child process and the output will be fed back into the original parent shell. For example:

- `echo $(expr 1 + 1)` will become `echo 2`
- `echo a b > dir; ls $(cat dir)` will list the contents of directories `a` and `b`.



This approach is contrasted below.

```
myshell> echo test
```

In the above case, Lex and Yacc parses the command and executes it normally.

```
myshell> echo $(ls) "and more"
file1 file2 file3 and more
```

In the above case, Lex and Yacc parses the command, but the `ls` command is evaluated and its output substituted before the echo command is executed.

The steps that take place for this latter example are as follows:

1. A command containing a subshell command is passed to the shell

   ```
   Input buffer=echo $(ls) "and more"
   Command word=
   ```

```
    Arguments=
```

2. The shell parses the `echo` command normally.
   `Input buffer=echo $(ls) "and more"`

   ```
   Command word=echo
       Arguments=
   ```

3. The shell identifies and executes the subshell command `` `ls` ``.
   `Input buffer=echo $(ls) "and more"`

   ```
   Command word=echo
       Arguments=
   ```

4. After executing the command in the subshell, the input is injected at the head of the buffer.
   `Input buffer=echo file1 file2 file3 "and more"`

   ```
   Command word=echo
   Arguments=
   ```

5. Finally the shell parses file2, file2, file3, and "and more" as the arguments to echo.
   `Input buffer=echo file1 file2 file3 "and more"`

   ```
   Command word=echo
   Arguments=file1, file2, file3, "and more"
   ```

You should implement this feature by:

1. Extracting the command between "$(" and ")" in `shell.l`
2. Invoking your own shell as a child process and providing the extracted command as input. You will need two pipes to communicate with the child subshell process; one to provide (write) the command to the child, and the other to obtain (read) the output from the child.
3. Placing the output obtained from the subshell into the scanner's buffer using the function `yy_unput(int c)` in reverse order. See the FAQ for more details.

Hint: It is common for students to redirect the current shell's stdin and stdout file descriptors to communicate with the subshell process, however this is not necessary. The current shell can communicate with the subshell by writing to the pipes directly.

IMPORTANT: Do not use the `popen()` call or a temporary file for the interprocess communication. You must use the method discussed above.

3.2 Environment variable expansion

You will implement environment variable expansion. Recall that in the previous part of the project, you allowed users to set and retrieve environmental variables using builtin functions.

When a string of the form `${var}` appears in an argument, it will be expanded to the value that corresponds to the variable `var` in the environment table. For example:

```
myshell> setenv A Hello
myshell> setenv B World
myshell> echo ${A} ${B}
Hello World
myshell> setenv C ap
myshell> setenv D les
myshell> echo I like ${C}p${D}
I like apples
```

Additionally, the following special expansions should be implemented:

`${$}`	The PID of the shell process.
`${?}`	The return code of the last executed simple command (ignoring commands sent to the background).
`${!}`	The PID of the last process run in the background.
`${!!}`	Run the last command again.
`${_}`	The last argument in the fully expanded previous command. Note that this excludes redirects.
`${SHELL}`	The path of your shell executable. Hint: realpath() can expand a relative path to an absolute path. You can obtain the relative path to the shell is `argv[0]`

3.3 Tilde expansion

When the character ~ appears itself or before "/" it will be expanded to the home directory of the current user. If ~ appears before a word, the characters after the ~ up to the first "/" will be expanded to the home directory of the user with that login. For example:

`ls ~` – List the home directory
`ls ~turkstra` – List turkstra's home directory
`ls ~turkstra/dir` – List subdirectory "dir" in turkstra's directory

Tilde expansion should not happen within double quotes.

3.4 Wildcarding

In most shells, including `bash`, you can use * and ? as wildcard characters in file and directory names. This is called file globbing. The * wildcard matches 0 or more non-blank characters, except the "." character if it is the first character in the file name. Try wildcarding in `bash` to see the results.

You will implement wildcarding as follows:

1. First, handle wildcarding only within the current directory:
    - Before inserting a new argument in the current simple command, check if the argument contains a wildcard character (* or ?). If it does, then insert the file names that match the wildcard (include their absolute paths).

- Use `opendir` and `readdir` to get all the entries of the current directory (check the `man` pages).
- Use the functions `regcomp` and regexec to find the entries that match the wildcard. Check the example provided in `regular.cc` to see how to do this. Notice that the wildcards and the regular expressions used in the library are different, so you will have to convert from wildcards to regular expressions.
- You should then be able to run commands like:

```
echo *
echo *.c
echo shell.?
```

2. Once your wildcarding implementation works for the current directory, make it work for any path. Some examples:

```
echo */*
echo */*/*
```

IMPORTANT: Do not use the `glob()` call. You must use the functions discussed above.

Reminder: you do not need to handle wildcard expansion between quotation marks!

3.5 Edit mode

`tty_raw_mode.c` contains the sample code that you will need to change your terminal's input from canonical to raw mode. In raw mode you will have more control over the terminal, passing the characters to the shell as they are typed. `read_line.c` has the sample code for line editing. You should implement your full line editor in this file.

There are two example programs to look at: `keyboard_example` and `read_line_example`. Run `keyboard_example` and type letters from your keyboard. You will see the corresponding ASCII code immediately printed on the screen.

The other program, `read_line_example`, is a simple line editor. Run this program and type `ctrl-?` to see the options of this program. The up-arrow causes the program to print the previous command in its history. You can use `read_line_example` to test and debug your line editor implementation.

To connect the line editor to your shell, add the following code to `shell.l` after the `#include` lines:

```
%{

#include <string.h>
#include "y.tab.h"

////////////  Start added code ///////////
```

```
extern \C" char *read_line();

int mygetc(FILE *f) {
   static char *p;
   char ch;

   if (!isatty(0)) {
     // stdin is not a tty. Call real getc
     return getc(f);
   }

   // stdin is a tty. Call our read_line.
   if ((p == NULL) || (*p == 0)) {
     char *s = read_line();
     p = s;
   }

   ch = *p;
   p++;

   return ch;
}

#undef getc
#define getc(f) mygetc(f)

/////////// End added code ///////////

%}

%%
```

Next modify your `Makefile` to compile your shell with the line editor. To do this just define the `EDIT_MODE_ON` variable in the `Makefile` to be something, for example "`yes`"

```
EDIT_MODE_ON=yes
```

Now modify `read_line.c` to add the following editor commands:

- Left arrow key: Move the cursor to the left and allow insertion at that position. If the cursor is at the beginning of the line it does nothing.
- Right arrow key: Move the cursor to the right and allow insertion at that position. If the cursor is at the end of the line it does nothing.
- Delete key (ctrl-D): Removes the character at the cursor. The characters in the right side are shifted to the left.

- Backspace key (ctrl-H): Removes the character at the position before the cursor. The characters in the right side are shifted to the left.
- Home key (ctrl-A): The cursor moves to the beginning of the line.
- End key (ctrl-E): The cursor moves to the end of the line.

IMPORTANT: Do not use the `readline` library. You must implement your own line editor.

3.6 History

In addition to the line editor above, also implement a history list. Currently the provided history is static. You need to update the history by creating your own history table. Every time the user runs a new command, a row will be added to the table. Implement the following editor commands:

- Up arrow key: Shows the previous command in the history list.
- Down arrow key: Shows the next command in the history list.

3.7 ctrl-R

In Bash, you can use ctrl-R to search through your command history. To use this, you can press ctrl-R and then start typing the command (any part of the command) that you are looking for. You'll start seeing the match, you can press "`Enter`" to execute the command. You can also press ctrl-R again and it will take you through other matches from your command history. For example, let's assume you execute some `ssh` commands and then want to search over them quickly:

```
bash$ ssh jeff@data.cs.purdue.edu
bash$ ssh jeff@xinu17.cs.purdue.edu
bash$ (reverse-i-search)‘ssh’: ssh jeff@data.cs.purdue.edu
```

As you may notice, there are two matches; if you press ctrl-R again, it will show you the second match.

```
bash$ (reverse-i-search)‘ssh’: ssh jeff@xinu17.cs.purdue.edu
```

Building up on the previous history task, implement the ctrl-R feature for your shell.

3.8 Submission

The deadline for part 3 submission is **Friday, October 11 11:59pm**.

Add a `README` to the proj3/ directory with the following:

1. Features specified in the handout that work.
2. Features specified in the handout that do not work.
3. Extra features you have implemented.

To turn in your part 3 submission:

1. Login to a CS department machine
2. Navigate to your `proj3` directory
3. Run `make clean`
4. Run `make` to check that your shell builds correctly
5. Run `make submit_part3`

These Standard Rules Apply

- Follow the coding standards as specified on the course website.
- Your shell must compile and run on the lab Linux machines.
- Do not look at anyone else's source code. Do not work with any other students.

4 Extra Credit

All parts below may be optionally completed to earn extra credit points.

4.1 Extra credit - Process Substitution (8 points)

Subshells allow one to obtain a command's output and use it within the shell. Pipes allow one to chain the output of one command to the input of another's. Both are particularly useful and powerful tools, but they have some limitations.

```
myshell> vim $(echo hello)
myshell> echo hello | vim

myshell> vim <(echo hello)

myshell> mkfifo fifo
myshell> echo hello > fifo &
myshell> vim fifo
myshell> rm fifo
```

In the example above, we want to edit the output of a command in vim. The problem is that vim does not read its input from stdin and it expects filenames for its arguments. Therefore, we cannot use pipes or subshells.

To do this, we can use process substitution. Process substitution allows us to pass the output of a command to another command as if the output was the contents of some file. To achieve this, the shell creates a named pipe to communicate between the two processes (echo and vim in the example above).

A named pipe, unlike the pipes we have been using so far, is created at some specified location in the file system (it has an inode associated with it and a directory entry). This allows us to pass a command a named pipe as a file path and interact with it that way.

To implement this feature you should use `mkdtemp()` to generate a temporary directory for the named pipe. Within the temp directory create a named pipe with `mkfifo()`. In your shell

connect the substituted command's output to the named pipe and pass the name of the pipe to the other command. Be sure to clean up any resources (fifo and directory) after the command has been executed. `unlink()` and `rmdir()` functions are likely helpful for this.

4.2 Extra Credit - Path Completion (2 points)

Implement path completion. When the `<tab>` key is typed, the editor will attempt to expand the current word to a matching file similar to what Bash does.

```
bash$ ls
cart.text card.txt
bash$ c<tab>
```

When tab is pressed, the line above becomes:

```
bash$ car
```

With the line indicator after c.

4.3 Extra Credit - Aliases (2 points)

Aliases allow one to define commands that encompass other commands. They are often used to create shortcuts for longer commands or to add default arguments. For example, you can create an alias for `ls` to always colorize the output, list entries by columns and append indicator to entries:

```
bash$ alias ls='ls -al --color=auto'
```

After this, whenever you use `ls`, it will be replaced by the longer command specified above.

Implement this functionality for your shell.

4.4 Extra Credit - Jobs (8 points)

Shells like Bash provide job control features that allow one to execute and manage multiple processes.

`jobs`	Display status of jobs in the current session. Refer to the `man` page `jobs` for operands.
`fg`	Run jobs in the foreground. Refer to the `man` page for `fg` for operands.
`bg`	Run jobs in the background. Refer to the `man` page for `bg` for operands.
`ctrl-Z`	Suspend a foreground process by sending it a `SIGTSTP` signal. You'll implement a similar signal handler as you did for ctrl-C

Below is an example usage of these commands. To run a foreground process, one could:

```
bash$ ping google.com
PING google.com (172.217.0.14) 56(84) bytes of data.
```

Reminder: You can run a background process using `ping google.com &`

Now let's say you want to suspend it. One can use ctrl-Z for that. We can use the `jobs` command to list the job ids and their statuses.

```
bash$ jobs
[1]+  Stopped ping google.com
```

Here [1] is the job id. We can use it to run it in the foreground or background. Let's run it in the background using `bg`:

```
bash$ bg %1
```

Now you'll again start seeing the output. Let's say you want to stop it, you can use `fg` to bring it to the foreground and then stop using ctrl-C.

```
bash$ fg %1
```

Grading

Grading is subject to change, but the point breakdown will roughly be as follows:

30 Part 1 (./testall part1)
30 Part 2 (./testall part2)
30 Part 3 (./testall part3)
10 Grading of readline and ctrl-C
-5 For Memory Leaks
-5 For File Descriptor Leaks

Note that the parts build on each other. A subset of points will be allocated to Part 1 in Part 2. A further subset of points will be obtained from Parts 1 and 2 for Part 3.