Project guidelines

Project guidelines

- Follow the instructions!!!
- Stay up-to-date
  - Don’t arrive late to class
    - Or take your responsibilities for not knowing what’s going on and losing points because of it
  - Read (entirely) every class announcement
    - Configure your account to receive notification emails
  - Always refer to the latest version of the assignment
  - Monitor Piazza

True for every aspects of this class (and your future career)

Project guidelines

- Respect deadlines
  - P1: Tuesday 18th, 11:59pm
  - For each hour late: -10%

Project guidelines

- Automatic grading is 50%
- Make sure that you pass the test script that was given to you
  - And that, before you continue adding any new features!
  - Very bad idea to think that the real grading script will somehow work with your code if the test script already doesn’t…

Project guidelines

- Git bundle
  - Just what’s necessary, nothing more:
    - Your source code
    - Makefile
    - The report file
    - (Your custom tester, or other relevant files)

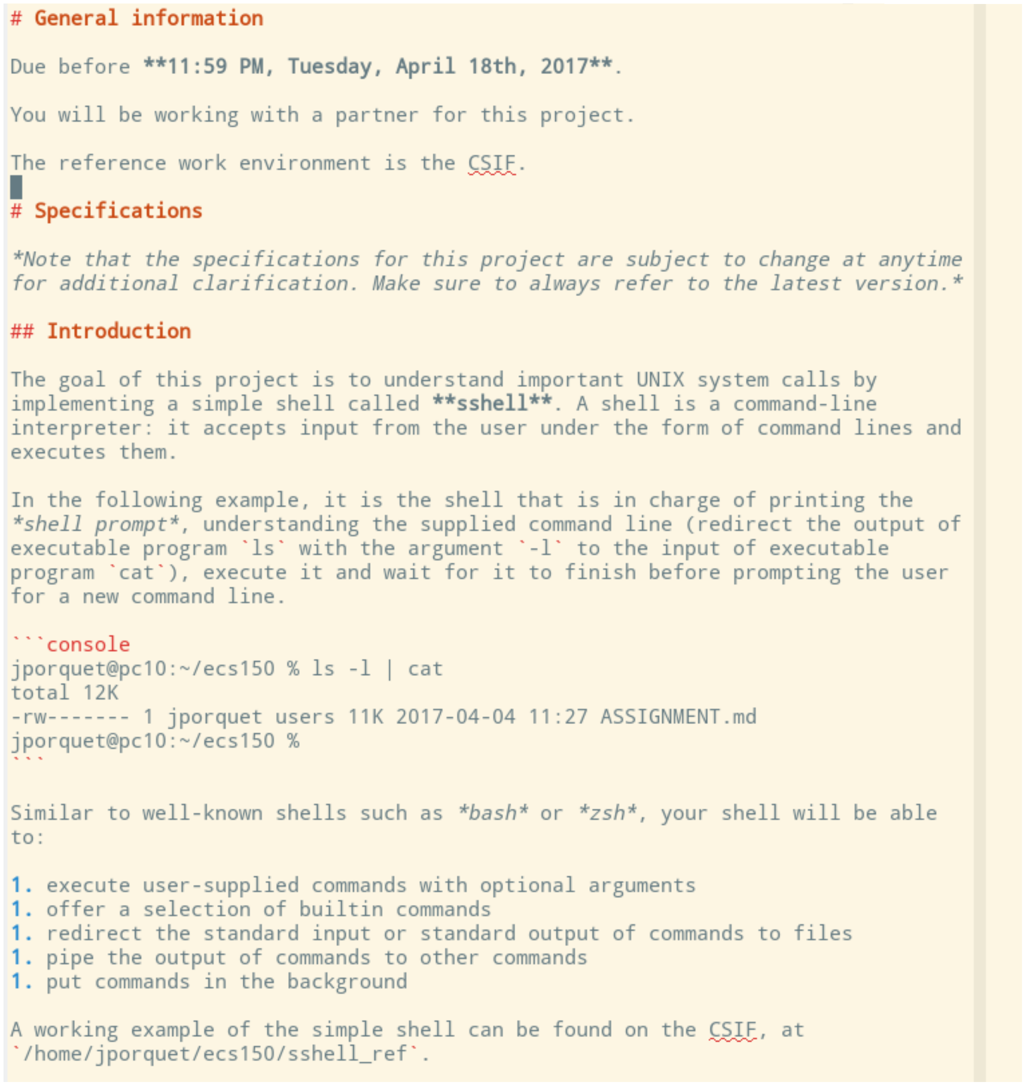
  - What’s not necessary: clutter!
    - Core dumps, backup files, object files, executable files, macos garbage folders (.__MACOSX), etc.

Project guidelines

- REPORT.md
  - Formatted in Markdown
    - There is a specification
      - https://guides.github.com/features/mastering-markdown/
    - Nothing fancy, but use at least:
      - Headers
      - Text formatting (bold, italic)
      - Lists
      - Code blocks (inline or not)
  - Formatted in lines of 80 characters max!

Project guidelines

- Good formatting:



~~~markdown
# General information

Due before **11:59 PM, Tuesday, April 18th, 2017**.

You will be working with a partner for this project.

The reference work environment is the CSIF.

# Specifications

*Note that the specifications for this project are subject to change at anytime
for additional clarification. Make sure to always refer to the latest version.*

## Introduction

The goal of this project is to understand important UNIX system calls by
implementing a simple shell called **sshell**. A shell is a command-line
interpreter: it accepts input from the user under the form of command lines and
executes them.

In the following example, it is the shell that is in charge of printing the
*shell prompt*, understanding the supplied command line (redirect the output of
executable program `ls` with the argument `-l` to the input of executable
program `cat`), execute it and wait for it to finish before prompting the user
for a new command line.

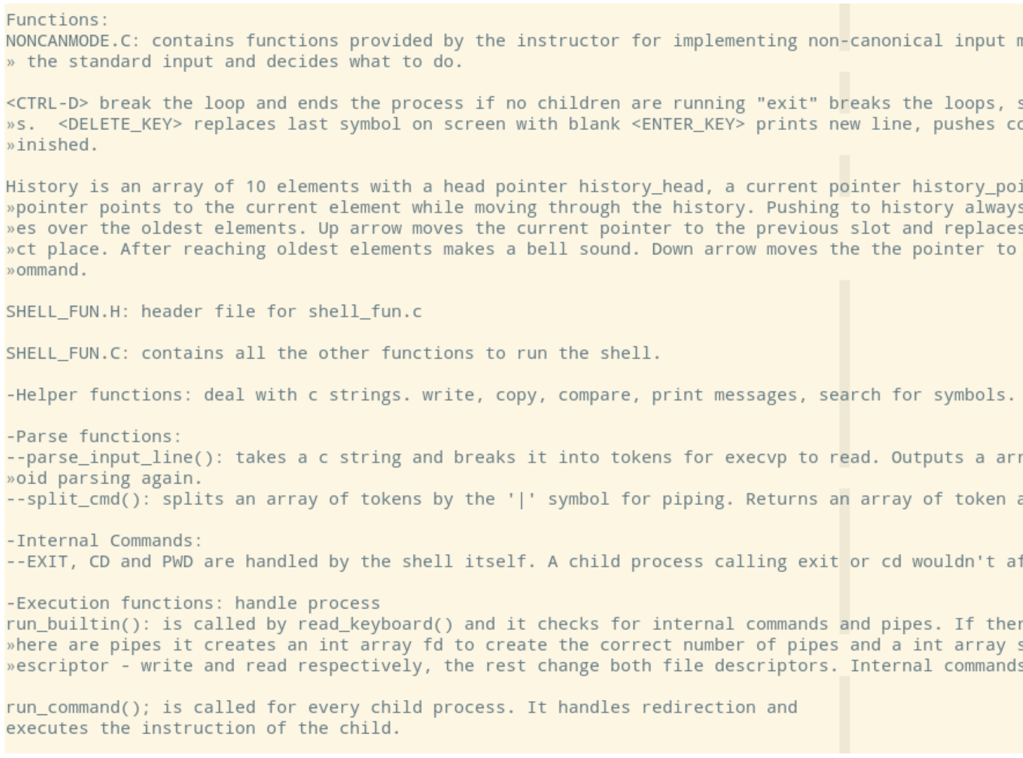
```console
jporquet@pc10:~/ecs150 % ls -l | cat
total 12K
-rw------- 1 jporquet users 11K 2017-04-04 11:27 ASSIGNMENT.md
jporquet@pc10:~/ecs150 %
~~~

Similar to well-known shells such as *bash* or *zsh*, your shell will be able
to:

1. execute user-supplied commands with optional arguments
1. offer a selection of builtin commands
1. redirect the standard input or standard output of commands to files
1. pipe the output of commands to other commands
1. put commands in the background

A working example of the simple shell can be found on the CSIF, at
`/home/jporquet/ecs150/sshell_ref`.
```

- Bad formatting:



```
Functions:
NONCANMODE.C: contains functions provided by the instructor for implementing non-canonical input m
» the standard input and decides what to do.

<CTRL-D> break the loop and ends the process if no children are running "exit" breaks the loops, s
»s.  <DELETE_KEY> replaces last symbol on screen with blank <ENTER_KEY> prints new line, pushes co
»inished.

History is an array of 10 elements with a head pointer history_head, a current pointer history_poi
»pointer points to the current element while moving through the history. Pushing to history always
»es over the oldest elements. Up arrow moves the current pointer to the previous slot and replaces
»ct place. After reaching oldest elements makes a bell sound. Down arrow moves the the pointer to
»ommand.

SHELL_FUN.H: header file for shell_fun.c

SHELL_FUN.C: contains all the other functions to run the shell.

-Helper functions: deal with c strings. write, copy, compare, print messages, search for symbols.

-Parse functions:
--parse_input_line(): takes a c string and breaks it into tokens for execvp to read. Outputs a arr
»oid parsing again.
--split_cmd(): splits an array of tokens by the '|' symbol for piping. Returns an array of token a

-Internal Commands:
--EXIT, CD and PWD are handled by the shell itself. A child process calling exit or cd wouldn't af

-Execution functions: handle process
run_builtin(): is called by read_keyboard() and it checks for internal commands and pipes. If ther
»here are pipes it creates an int array fd to create the correct number of pipes and a int array s
»escriptor - write and read respectively, the rest change both file descriptors. Internal commands

run_command(); is called for every child process. It handles redirection and
executes the instruction of the child.
```
```

# Project guidelines

- REPORT.md
  - Don’t tell me what I already know
    - Remember, I wrote the assignment ;)
  - Tell me a story that explains how you transformed the assignment into code
    - I.e. not “What does your code do?”
    - But “How does it do it?”
    - Mention the limitations if any
    - Don’t write a book though and don’t explain every line of code...
  - Reminder: Half of the manual review grade
    - Take care of the report before adding the last feature at the last minute!

## Project guidelines

- Makefile
  - Use -Wall -Werror
  - Compile without any warnings or errors

## Project guidelines

- Coding style
  - Consistent
    - Don’t mix tab and spaces
    - Keep the same indentation (at least 4)
  - Comment your code (with meaningful comment)
    - Example of poor comment: `i++; // increment variable i`
  - Name your variable properly
    - Example of poor name: `int temp;`
- If you want to become a pro, start acting like one now:
  - Don’t submit your draft
  - Submit a program that is nice to read

# Project guidelines

- Implementation quality
  - Writing some code that implements certain specs is not enough
  - A good code is easy to extend
  - Use the right data structures
    - If you're using char *** in your code, that's probably the wrong approach
  - Split your functions the right way
    - Ideally, one function per logical functionality
  - Don't over-complicate your code: simple is always the best option