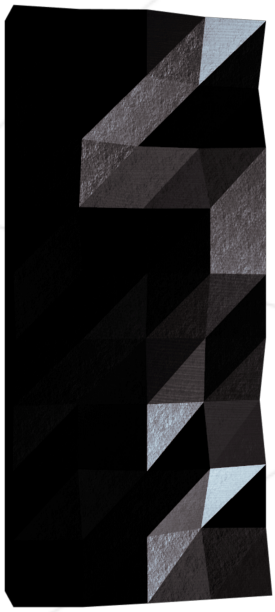

Minishell

Tan bonito como shell

Resumen: El objetivo de este proyecto es que crees un shell sencillo. Sí, tu propio bash o zsh. Aprenderás un montón de procesos y file descriptors.

Versión: 6

Índice general

Capítulo I

Introducción

La existencia de los shells se remonta a los orígenes de IT. Por aquel entonces, todos los programadores pensaban que `comunicarse con un ordenador utilizando interruptores de 1/0 era realmente frustrante`. Era cuestión de tiempo que empezaran a utilizar `líneas de comando interactivas para comunicarse con los ordenadores en un lenguaje parecido al inglés`.

Con `Minishell`, probablemente viajarás en el tiempo y descubrirás los problemas a los que la gente se enfrentaba cuando `Windows` no existía.

Capítulo II

Instrucciones generales

- Tu proyecto debe estar escrito siguiendo la Norma. Si tienes archivos o funciones adicionales, estas están incluidas en la verificación de la Norma y tendrás un `0` si hay algún error de norma dentro.
- Tus funciones no deben terminar de forma inesperada (segfault, bus error, double free, etc) ni teener comportamientos indefinidos. Si esto pasa tu proyecto será considerado no funcional y recibirás un `0` durante la evaluación.
- Toda la memoria alocada en heap deberá liberarse adecuadamente cuando sea necesario. No se permitirán leaks de memoria.
- Si el subject lo requiere, deberás entregar un `Makefile` que compilará tus archivos fuente al output requerido con las flags `-Wall`, `-Werror` y `-Wextra`, por supuesto tu `Makefile` no debe hacer relink.
- Tu `Makefile` debe contener al menos las normas `$(NAME)`, `all`, `clean`, `fclean` y `re`.
- Para entregar los bonus de tu proyecto, deberás incluir una regla `bonus` en tu `Makefile`, en la que añadirás todos los headers, librerías o funciones que estén prohibidas en la parte principal del proyecto. Los bonus deben estar en archivos distintos `_bonus.{c/h}`. La parte obligatoria y los bonus se evalúan por separado.
- Si tu proyecto permite el uso de la `libft`, deberás copiar su fuente y sus `Makefile` asociados en un directorio `libft` con su correspondiente `Makefile`. El `Makefile` de tu proyecto debe compilar primero la librería utilizando su `Makefile`, y después compilar el proyecto.
- Te recomendamos crear programas de prueba para tu proyecto, aunque este trabajo **no será entregado ni evaluado**. Te dará la oportunidad de verificar que tu programa funciona correctamente durante tu evaluación y la de otros compañeros. Y sí, tienes permitido utilizar estas pruebas durante tu evaluación o la de otros compañeros.
- Entrega tu trabajo a tu repositorio `Git` asignado. Solo el trabajo de tu repositorio `Git` será evaluado. Si Deepthought evalúa tu trabajo, lo hará después de tus compañeros. Si se encuentra un error durante la evaluación de Deepthought, la evalaución terminará.

Capítulo III

Parte obligatoria

Nombre de programa	`minishell`
Archivos a entregar	
Makefile	`Sí`
Argumentos	
Funciones autorizadas	`readline, rl_clear_history, rl_on_new_line, rl_replace_line, rl_redisplay, add_history, printf, malloc, free, write, access, open, read, close, fork, wait, waitpid, wait3, wait4, signal, sigaction, kill, exit, getcwd, chdir, stat, lstat, fstat, unlink, execve, dup, dup2, pipe, opendir, readdir, closedir, strerror, perror, isatty, ttyname, ttyslot, ioctl, getenv, tcsetattr, tcgetattr, tgetent, tgetflag, tgetnum, tgetstr, tgoto, tputs`
Se permite usar libft	`Sí`
Descripción	`Escribe un shell`

Tu shell deberá:

- No interpretar comillas sin cerrar o caracteres especiales no especificados como \ o ;.
- No usar más de una variable global, piensa por qué y prepárate para explicar el uso que le das.
- Mostrar una entrada que espere introducir un comando nuevo.
- Tener un historial funcional.
- Busca y ejecuta el ejecutable correcto (basado en la variable PATH o mediante el uso de rutas relativas o absolutas).
- Deberá implementar los builtins:

  - `echo` con la opción -n.
  - `cd` solo con una ruta relativa o absoluta.
  - `pwd` sin opciones.
  - `export` sin opciones.
  - `unset` sin opciones.
  - `env` sin opciones o argumentos.
  - `exit` sin opciones.
- `'` inhibiendo toda interpretación de una secuencia de caracteres.
- `"` inhibiendo toda interpretación de una secuencia de caracteres excepto $.
- Redirecciones:
  - `<` debe redirigir input.
  - `>` debe redirigir output.
  - "`<<`" debe leer del input de la fuente actual hasta que una línea que contenga solo el delimitador aparezca. No necesita actualizar el historial.
  - "`>>`" debe redirigir el output en modo append.
- Pipes `|`. El output de cada comando en la pipeline se conecta a través de un pipe al input del siguiente comando.
- Las variables de entorno (`$` seguidos de caracteres) deberán expandirse a sus valores.
- `$?` deberá expandirse al estado de salida del comando más reciente ejecutado en la pipeline.
- `ctrl-C` `ctrl-D` `ctrl-\` deberán funcionar como en bash.
- Cuando sea interactivo:
  - `ctrl-C` imprime una nueva entrada en una línea nueva.
  - `ctrl-D` termina el shell.
  - `ctrl-\` no hace nada.

La función `readline` puede producir algunos leaks que no necesitas arreglar.
Eso no significa que tu código pueda producir leaks.
Limítate a hacer lo que pide el enunciado.
Cualquier cosa no solicitada no se requiere.
Para cada punto, y en caso de dudas, puedes utilizar bash como una referencia.

Capítulo IV

Parte extra

- Si la parte obligatoria no está PERFECTA, olvídate de los bonus.
- `&&`, `||` con paréntesis para prioridades.
- la wildcard $*$ debe funcionar para el directorio actual.