TP1: Intérprete de comandos - shell

Introducción

AVISO: antes de comenzar, verificar que se tiene instalado **el software necesario**.

En este trabajo se va a desarrollar la funcionalidad mínima que caracteriza a un intérprete de comandos *shell* similar a lo que realizan `bash`, `zsh`, `fish`.

La implementación debe realizarse en C11 y POSIX.1-2008. *(Estas siglas hacen referencia a la versión del lenguaje C utilizada y del estándar de syscalls Unix empleado. Las versiones modernas de GCC y Linux cumplen con ambos requerimientos.)*

Implementación

Parte 1: Invocación de comandos

Búsqueda en *$PATH*

Los comandos que más se utilizan, como `ls`, `echo`, etc., están guardados (sus binarios), en el directorio `/bin`. Por este motivo existe una variable de entorno llamada `$PATH`, en la cual se declaran las rutas más usualmente accedidas por el sistema operativo (ejecutar: `echo $PATH` para ver la totalidad de las rutas almacenadas). Por ejemplo:

```
$ uptime
 05:45:25 up 5 days, 12:02,  5 users,  load average: ...
```

Asimismo, es deseable, la funcionalidad de poder pasarle *argumentos* al momento de querer ejecutar dichos comandos. Los argumentos pasados al programa de esta forma, se guardan en la famosa variable **char* argv[]**, junto con cuántos fueron en **int argc**, declaradas en la función `main` de cualquier programa en *C*.

```
$ df -H /tmp
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           8.3G  2.0M  8.3G   1% /tmp
```

Tareas

- Soportar la ejecución de binarios
  - Con búsqueda en `$PATH`
  - Con y sin argumentos
- **Responder**: ¿cuáles son las diferencias entre la *syscall* `execve(2)` y la familia de *wrappers* proporcionados por la librería estándar de *C* (*libc*) `exec(3)`?
- **Responder**: ¿Puede la llamada a `exec(3)` fallar? ¿Cómo se comporta la implementación de la *shell* en ese caso?

Función sugerida: `execvp(3)`

Archivo: `exec_cmd()` en *exec.c*

Resumen

Al finalizar la parte 1 la *shell* debe poder:

- Invocar programas y permitir pasarles argumentos.
- Esperar correctamente a la ejecución de los programas.

Parte 2: Redirecciones

AVISO: El esqueleto no soporta espacios entre los caracteres de *redirección* y el nombre del archivo proporcionado.

Es decir, se debe ejecutar **siempre**: `<in.txt` y `>out.txt`.

Flujo estándar

La redirección del flujo estándar es una de las cualidades más interesantes y valiosas de una *shell* moderna. Permite, entre otras cosas, almacenar la salida de un programa en un archivo de texto para luego poder analizarla, como así también ejecutar un programa enviándole un archivo como entrada estándar. Existen, básicamente, tres formas de redirección del flujo estándar:

- **Entrada y Salida estándares a archivos** *(`<in.txt` `>out.txt`)*

  Son los operadores clásicos del manejo de la redirección del *stdin* y el *stdout* en archivos de entrada y salida respectivamente. Por ejemplo:

```
$ ls /usr
bin etc games include lib local sbin share src

$ ls /usr >out1.txt
$ cat out1.txt
bin etc games include lib local sbin share src

$ wc -w <out1.txt
10

$ ls -C /sys /noexiste >out2.txt
ls: cannot access '/noexiste': No such file or directory

$ cat out2.txt
/sys:
block  class  devices    fs  kernel  module  power

$ wc -w <out2.txt
8
```

  Se puede ver cómo queda implícito que cuando se utiliza el operador **>** se refiere al *stdout* y cuando se utiliza el **<** se refiere al *stdin*.

- **Error estándar a archivo** *(`2>err.txt`)*

  Es una de las dos formas de redireccionar el *flujo estándar de error* análogo al caso anterior del flujo de salida estándar en un archivo de texto. Por ejemplo:

```
$ ls -C /home /noexiste >out.txt 2>err.txt

$ cat out.txt
/home:
patricio

$ cat err.txt
ls: cannot access '/noexiste': No such file or directory
```

  Como se puede observar, `ls` no informa ningún error al finalizar, como sí lo hacía en el ejemplo anterior. Su salida estándar de error ha sido redireccionada al archivo *err.txt*

- **Combinar salida y errores** *(`2>&1`)*

  Es la segunda forma de redireccionar el flujo estándar producido por errores en la ejecución de un programa. Su funcionamiento se puede observar a través del siguiente ejemplo:

```
$ ls -C /home /noexiste >out.txt 2>&1

$ cat out.txt
---????---
```

Existen más tipos de redirecciones que nuestra *shell* no soportará (e.g. `>>` o `&>`)

Tareas

- Soportar cada una de las **tres formas de redirección** descritas arriba: `>`, `<`, `2>` y `2>&1`.

- **Responder**: Investigar el significado de `2>&1`, explicar cómo funciona su *forma general*
  - Mostrar qué sucede con la salida de `cat out.txt` en el ejemplo.
  - Luego repetirlo, invirtiendo el orden de las redirecciones (es decir, `2>&1 >out.txt`). ¿Cambió algo? Compararlo con el comportamiento en `bash(1)`.

Ayuda: Pueden valerse de las páginas del manual de bash: `man bash`.

Syscalls sugeridas: `dup2(2)`, `open(2)`

Archivo: `open_redir_fd()` en *exec.c* y usarla en `exec_cmd()`

Tuberías simples *(pipes)*

Al igual que la redirección del flujo estándar hacia archivos, es igual o más importante, la redirección hacia otros programas. La forma de hacer esto en una *shell* es mediante el operador `|` (*pipe* o *tubería*). De esta forma se pueden concatenar dos o más programas para que la salida estándar de uno se redirija a la entrada estándar del siguiente. Por ejemplo:

```
$ ls -l | grep Doc
drwxr-xr-x  7 patricio patricio  4096 mar 26 01:20 Documentos
```

Tareas

- Soportar **pipes** entre dos comandos.
- La shell debe esperar a que **ambos** procesos terminen antes de devolver el prompt: `echo hi | sleep 5` y `sleep 5 | echo hi` *ambos* deben esperar 5 segundos.
- Los procesos de cada lado del pipe no deben quedar con *fds* de más.
- Los procesos deben ser lanzados *en simultáneo*.

Syscalls sugeridas: `pipe(2)`, `dup2(2)`

Archivo: `exec_cmd()` en *exec.c*

Tuberías múltiples

Extender el funcionamiento de la *shell* para que se puedan ejecutar **n** comandos concatenados.

```
$ ls -l | grep Doc | wc
     1       9      64
```

Tareas

- Soportar **múltiples pipes** anidados.
- **Responder**: Investigar qué ocurre con el *exit code* reportado por la *shell* si se ejecuta un pipe
  - ¿Cambia en algo?
  - ¿Qué ocurre si, en un pipe, alguno de los comandos falla? Mostrar evidencia (e.g. salidas de terminal) de este comportamiento usando `bash`. Comparar con su implementación.

Hint: Las modificaciones necesarias sólo atañen a la función `parse_line()` en *parsing.c*

Resumen

Al finalizar la parte 2 la *shell* debe poder:

- Redireccionar la entrada y salida estándar de los programas vía `<`, `>` y `2>`.
  - Además se soporta específicamente la redirección de tipo `2>&1`
- Concatenar la ejecución de dos o más programas mediante *pipes*

Parte 3: Variables de entorno

Expansión de variables

Una característica de cualquier intérprete de comandos *shell* es la de expandir variables de entorno (ejecutar: `env` para ver una lista completa de las variables de entorno definidas), como **PATH**, o **HOME**.

```
$ echo $TERM
xterm-16color
```

Las variables de entorno se indican con el caracter `$` antes del nombre, y la *shell* se encarga de *reemplazar* en la línea leída todos los tokens que comiencen por `$` por los valores correspondientes del entorno. Esto ocurre *antes* de que el proceso sea ejecutado.

Las variables vacías y las variables no setteadas *no deben* traducirse a argumentos en la etapa `exec`. Por ejemplo `echo hola $VARIABLE_VACIA mundo` es equivalente a `echo "hola" "mundo"`, **dos** argumentos solamente.

Tareas

- Soportar la expansión de variables al ejecutar un comando.
- Se debe reemplazar las variables que no existan con una cadena vacía (`""`).

Función sugerida: `getenv(3)`

Archivos: `expand_environ_var()` y `parse_exec()` en *parsing.c*,

Variables de entorno temporarias

En esta parte se va a extender la funcionalidad de la *shell* para que soporte el poder incorporar nuevas variables de entorno a la ejecución de un programa. Cualquier programa que hagamos en C, tiene acceso a todas las variables de entorno definidas mediante la variable externa *environ* (`extern char** environ`).[1]

Se pide, entonces, la posibilidad de incorporar de forma dinámica nuevas variables, por ejemplo:

```
$ /usr/bin/env
--- todas las variables de entorno definidas hasta el momento ---

$ USER=nadie ENTORNO=nada /usr/bin/env | grep =nad
USER=nadie
ENTORNO=nada
```

Tareas

- Soportar variables de entorno temporales.
- **Responder**: ¿Por qué es necesario hacerlo luego de la llamada a `fork(2)` ?
- **Responder**: En algunos de los *wrappers* de la familia de funciones de `exec(3)` (las que finalizan con la letra *e*), se les puede pasar un tercer argumento (o una lista de argumentos dependiendo del caso), con nuevas variables de entorno para la ejecución de ese proceso. Supongamos, entonces, que en vez de utilizar `setenv(3)` por cada una de las variables, se guardan en un arreglo y se lo coloca en el tercer argumento de una de las funciones de `exec(3)` .
  - ¿El comportamiento resultante es el mismo que en el primer caso? Explicar qué sucede y por qué.
  - Describir brevemente (sin implementar) una posible implementación para que el comportamiento sea el mismo.

Ayuda: luego de llamar a `fork(2)` , realizar, por cada una de las variables de entorno a agregar, una llamada a `setenv(3)` .

Función sugerida: `setenv(3)`

Archivo: implementar `set_environ_vars()` en *exec.c* y usarla en `exec_cmd()`

Pseudo-variables

Existen las denominadas variables de entorno *mágicas*, o pseudo-variables. Estas variables son propias de la *shell* (no están formalmente en *environ*) y cambian su valor dinámicamente a lo largo de su ejecución. Implementar `?` como única variable *mágica* (describir, también, su propósito).

```
$ /bin/true
$ echo $?
0

$ /bin/false
$ echo $?
1
```

Tareas

- Soportar para la *pseudo-variable* `$?`.
  - Esto implicará actualizar correctamente la variable *global* `status` cuando se ejecute un *built-in* (ya que los mismos no corren en procesos separados).
- **Responder**: Investigar al menos otras tres variables mágicas estándar, y describir su propósito.
  - Incluir un ejemplo de su uso en `bash` (u otra terminal similar).

Archivo: `expand_environ_var()` en *parsing.c*, ver también la variable *global* `status`.

Resumen

Al finalizar la parte 3 la *shell* debe poder:

- Expandir variables de entorno
- Incluyendo la pseudo-variable `$?`
- Ejecutar procesos con variables de entorno adicionales

Parte 4: Comandos *built-in*

Los comandos *built-in* nos dan la oportunidad de realizar acciones que no siempre podríamos hacer si ejecutáramos ese mismo comando en un proceso separado. Éstos son propios de cada *shell* aunque existe un estándar generalizado entre los diferentes intérpretes, como por ejemplo `cd` y `exit`.

Es evidente que si `cd` no se realizara en el mismo proceso donde la *shell* se está ejecutando, no tendría el efecto deseado. Lo mismo se aplica a `exit` y a muchos comandos más (aquí se muestra una lista completa de los comando *built-in* que soporta *bash*).

Tareas

- Soportar los *built-ins*:
  - `cd` - *change directory* (cambia el directorio actual)
  - `exit` - exits nicely (termina una terminal de forma *linda*)
  - `pwd` - *print working directory* (muestra el directorio actual de trabajo)
- **Responder**: ¿Entre `cd` y `pwd`, alguno de los dos se podría implementar sin necesidad de ser *built-in*? ¿Por qué? ¿Si la respuesta es sí, cuál es el motivo, entonces, de hacerlo como *built-in*? (para esta última pregunta pensar en los *built-in* como `true` y `false`)

Funciones sugeridas: `chdir(3)`, `exit(3)`, `getcwd(3)`

Archivo: `cd()`, `exit_shell()` y `pwd()` en *builtin.c*

Parte 5: Procesos en segundo plano

Los procesos en segundo plano o procesos en el “fondo”, o *background*, son muy útiles a la hora de ejecutar comandos que no queremos esperar a que terminen. En estos casos la *shell* nos devuelva el prompt inmediatamente, para seguir realizando tareas. Por ejemplo, si queremos ver un documento `.pdf` o una imagen y queremos seguir trabajando sin tener que abrir una terminal.

Esta implementación debe manejar la liberación de *recursos* del proceso, en el *mismo* momento en que termina.

Para poder realizar ésto, vamos a manejar *señales* (o `signals`). La señal que nos interesa *atrapar* es `SIGCHLD`, la cual se genera cada vez que un proceso hijo termina (es decir, llama a `exit(3)`).

El *sistema operativo* nos da la posibilidad de poder ejecutar lógica *custom* para cada señal que se precise manejar de manera particular. Con la *syscall* `sigaction(2)` vamos a poder configurar lo que se denomina un **handler** (a.k.a una *función*) para dicha señal y liberar los recursos del proceso en *segundo plano* que haya finalizado.

Desde el punto de vista del usuario de la *shell*, el comportamiento al final esta tarea, debería ser el siguiente:

```
$ sleep 2 &
PID=2489

$ sleep 5
<pasan dos segundos, y entonces:

==> terminado: PID=2489

:ahora pasan otros tres segundos antes de retornar>
$
```

En otras palabras, se notifica de la terminación en cuanto ocurre, sin esperar al siguiente prompt.

También se observa el comportamiento en la ausencia de un segundo comando en primer plano; simplemente, se escribiría en la línea del *prompt* actual:

```
$ sleep 2 &
PID=2489

$ ==> terminado: PID=2489
 ^
 dos segundos después, se imprime a continuación del prompt
```

Se recomienda, de hecho, realizar las primeras pruebas con este segundo ejemplo, para trabajar con una solo evento de la señal `SIGCHLD`.

Una vez hecho ésto, se debe resolver el problema de que, los procesos en *primer plano* (o `foreground`) también generan `SIGCHLD` cuando finalizan. El *handler* los aceptaría, quedando la llamada a `waitpid(2)` en `run_cmd()` incapaz de obtener el estado de salida del hijo.

La solución más fácil es asegurarse de que todos los procesos en *segundo plano* tengan un mismo *process group*. Y que la llamada a `waitpid(2)` en el *handler* no use -1 como argumento (es decir, esperar por *cualquier proceso*), sino un valor numérico que restrinja la llamada a los procesos en segundo plano.

Sugerencia: configurar el uso de grupos tal que ese primer argumento de `waitpid(2)` pueda ser, sencillamente, `0`.

Tareas

- Manejar los proceso en segundo plano inmediatamente cuando finalizan.
- Explicar detalladamente el mecanismo completo utilizado.
- **Responder**:
  - ¿Por qué es necesario el uso de señales?

Ayuda: Leer el funcionamiento del flag `WNOHANG` de la syscall `wait(2)`

Páginas de manual: `man 7 signal`, `man 7 signal-safety`

Syscalls sugeridas: `setpgid(2)`, `getppid(2)`, `sigaction(2)`, `sigaltstack(2)`

Archivos: *exec.c*, *runcmd.c*, *sh.c*

Esqueleto y compilación

AVISO: El esqueleto se encuentra disponible en **fisop/shell**.

IMPORTANTE: leer el archivo `shell/README.md` que se encuentra en el proyecto. Contiene información sobre cómo realizar la compilación de los archivos, y cómo ejecutar el formateo de código.

Para que no tengan que implementar todo desde cero, se provee un esqueleto. Éste tiene gran parte del parseo hecho, y está estructurado indicando con comentarios los lugares en donde deben introducir el código crítico de cada punto.

Se recomienda, antes de empezar, leer el código para entender bien cómo funciona, y qué hace cada una de las funciones. **Particularmente recomendamos entender qué significa cada uno de los campos en los structs definidos en `types.h`.**

Compilación

Simplemente alcanza con ejecutar `make`.

Ejecución

Se proveen dos formas para ejecutar la *shell*: `make run` y `make valgrind` que ejecuta el binario dentro de una sesión de `valgrind`.

Depurando con printf

Es importante mencionar que es requisito usar las funciones `printf_debug` y `fprintf_debug` si se desea mostrar información por pantalla; o bien encapsular todo lo que se imprima por stdout o stderr utilizando la macro `SHELL_NO_INTERACTIVE` (como ejemplo, ver las funciones definidas en `utils.c`).

Esto es debido a que al momento de corregir es mucho más fácil ejecutar una shell en modo no interactivo (que no imprima *prompt*) y así poder comparar el output de forma automática.

Cualquier mensaje que se imprima por pantalla al momento de hacer la entrega tiene que hacerse con las funciones `printf_debug` (en lugar de `printf`) o bien encapsulando el código con la directiva del preprocesador `#ifndef SHELL_NO_INTERACTIVE`.

Desafíos

Las tareas listadas aquí no son obligatorias, pero suman para el régimen de final alternativo.

Historial

Si bien el *historial* es también un *built-in*, lo tratamos de forma separada dada su importancia y dificultad técnica

Implementar el *built-in* `history`, el mismo muestra la lista de comandos ejecutados hasta el momento. De proporcionarse como argumento `n`, un número entero, solamente se mostrarán los últimos `n` comandos.

De estar definida la variable de entorno **HISTFILE**, la misma contendrá la ruta al archivo con los comandos ejecutados en el pasado. En caso contrario, utilizar como ruta por omisión a

`~/.fisop_history`.

Permitir navegar el historial de comandos con las teclas ↑ y ↓, de modo tal que se pueda volver a ejecutar alguno de ellos. Con la tecla ↑ se accede a un comando anterior, y con la tecla ↓ a un comando posterior, si este último no existe, eliminar el comando actual de modo que solo se vea el prompt.

La tecla BackSpace debe funcionar para borrar los caracteres de un comando de hasta una línea de largo. Al presionar Ctrl + d, la shell debe terminar su ejecución.

Ayuda:

- Para tener mayor control sobre la entrada de caracteres, la terminal debe configurarse en modo **no canónico** y **sin** eco. Puede usarse como guía el ejemplo Noncanonical Mode presente en Low-Level Terminal Interface.
- Pueden ser de utilidad algunas secuencias de escape ANSI.
- Pueden obtener información sobre la terminal con la llamada al sistema `ioctl`, ver: ioctl_tty(2).

Tareas

- Soportar el comando `history [n]`
- Soportar las teclas ↑ y ↓
- La tecla “backspace” permite borrar comandos de hasta una línea de largo
- La tecla Ctrl + d debe terminar la ejecución de la shell (como hasta ahora)
- Soportar la variable de entorno `HISTFILE`
- Implementar al menos una de las siguientes tareas:
  - Permitir borrar con la tecla BackSpace los caracteres de un comando de cualquier número de líneas, independientemente que la escritura del mismo haya ocasionado el desplazamiento de la pantalla, esto ocurre al continuar escribiendo tras alcanzar la posición inferior derecha de la terminal.
  - Desplazar el cursor de a un caracter con las teclas ← y →, de a una palabra con Ctrl + ← y Ctrl + →, al comienzo del comando con Home, y al final del mismo con End, permitiendo insertar nuevos caracteres en cualquier posición.
  - Implementar los designadores de eventos `!!` y `!-n`, ver sección *Event Designators* en bash(1).
- **Responder**: ¿Cuál es la función de los parámetros `MIN` y `TIME` del modo no canónico? ¿Qué se logra en el ejemplo dado al establecer a `MIN` en `1` y a `TIME` en `0`?

1. No es necesario realizar el *include* de ningún header para hacer uso de esta variable. ↩