Actividad Práctica 2

Crear una Shell

Profesor: Ismael Figueroa
Equipo docente: Diego Arias, Fabián Díaz, Andrés Gallardo, Blaz Korecic

En la actividad anterior se estudió como utilizar Bash a un nivel introductorio, el cual es una shell popular que nos permite interactuar con el sistema ejecutando binarios y moviéndonos a través del sistema de archivos.

El objetivo de esta actividad es dar el siguiente paso: implementar nuestra propia shell simple desde cero. Al construirla, entenderás mejor cómo funcionan internamente estos intérpretes y cómo interactúan con las funciones básicas del sistema operativo.

Podrás realizar esta implementación en el lenguaje que desees, aunque recomendamos fuertemente usar C, C++ u otro lenguaje de sistemas. Estos lenguajes suelen permitir un acceso directo a las llamadas al sistema (syscalls) necesarias para crear una shell, lo cual es parte clave del aprendizaje de esta actividad. Considera que **si no usas C, no te aseguramos poder responder tus dudas de código**.

Además, se provee una solución base en C por sobre la cual podrás implementar tu tarea, en caso de decidir implementar la tarea en C.

~~P1.~~ Shell minimalista (1 pt)

Comenzaremos construyendo una shell que solamente lea la entrada del usuario y reemplace el proceso con el ejecutable entregado.

Ejemplo de uso esperado:

```
1 [user@hostname ~]$ ./mi-shell
2 mi-shell-prompt> echo hola
3 hola
```

En este ejemplo primero ejecutamos nuestra shell ya compilada desde Bash. Así, la primera línea correspondería al *prompt* de Bash y la segunda al prompt de nuestra shell, en donde podemos escribir algún comando que será ejecutado por la shell.

Indicaciones:

a) Muestra un *prompt* simple al usuario (por ejemplo, `mi-shell-prompt>`) mediante la salida estándar (stdout).
b) Lee una línea completa de texto ingresada por el usuario desde la entrada estándar (stdin).
c) Procesa la línea leída:
   - Necesitas separar el comando de sus argumentos. Por ahora usaremos los espacios como delimitadores. A esto se le llama *tokenizar*, y en `C` puedes usar la función `strtok_r`.

~~P2.~~ Shell con *fork* (1 pt)

Habrán notado que la shell minimalista anterior tiene un problema: al usar `exec`, el proceso de la shell es reemplazado por el comando ejecutado, terminando la shell tras un solo comando.

Para permitir múltiples comandos, usamos la llamada al sistema `fork()`. Esta crea un **proceso hijo**, una copia casi idéntica del **proceso padre** (nuestra shell). La diferencia clave es el valor devuelto por `fork()`:

- `0` en el proceso hijo.
- El **PID** del hijo en el proceso padre.
- `-1` si hay un error.

El flujo en nuestra shell con `fork()` es:

a) La shell (padre) lee y parsea el comando.
b) Llama a `fork()`.
c) **Proceso hijo** (`fork()` devuelve 0): Llama a `exec` (ej. `execvp`) para ejecutar el comando solicitado. Si `exec` falla, el hijo debe reportar el error y terminar (`exit()`).
d) **Proceso padre** (`fork()` devuelve PID > 0): No llama a `exec`. Espera a que el hijo termine usando `wait()` o `waitpid()`.
e) Una vez el hijo termina, el padre vuelve al paso 1 para leer el siguiente comando.

Así, la shell original (padre) persiste mientras los comandos se ejecutan en procesos hijos separados.

P3. Agregando *builtins* a nuestra shell (2.5 pts)

En esta parte agregaremos *builtins* a nuestra shell. Éstos son comandos propios de nuestra shell, que no corresponden a un ejecutable externo:

- **(0.4 pts)** `cd`: Cambia el directorio actual del proceso de la shell. **Hint**: Usa la syscall chdir.
- **(0.4 pts)** `exit`: Termina el proceso de la shell, opcionalmente con un código de estado numérico. **Hint**: Usa la función exit de la librería estándar de C (stdlib.h). Asegúrate de manejar el argumento opcional (el código de estado) si decides implementarlo.
- **(0.4 pts)** `pwd`: Muestra (imprime en la salida estándar) la ruta absoluta del directorio de trabajo actual. **Hint**: Usa la función getcwd para obtener la ruta actual y luego imprímela en la salida estándar (stdout). Recuerda manejar posibles errores de getcwd.
- **(0.4 pts)** `export`: Define o modifica una variable de entorno para la shell actual y los procesos que esta ejecute. Usualmente toma la forma export NOMBRE=VALOR. **Hint**: Usa la función setenv. Deberás parsear el argumento para separar el nombre de la variable y su valor (buscar el primer =).
- **(0.4 pts)** `unset`: Elimina una variable de entorno de la shell actual y, por lo tanto, de los futuros procesos hijos. **Hint**: Usa la función unsetenv, pasándole como argumento el nombre de la variable a eliminar.
- **(0.5 pts)** `history`: Muestra el historial de comandos que se han introducido en la sesión actual de la shell. **Hint**: Necesitarás mantener una estructura de datos interna (como un array o lista enlazada) para almacenar cada línea de comando ingresada. El comando history simplemente recorre e imprime el contenido de esta estructura.

P4. Agregando funcionalidad avanzada (1.5 pt)

Debes escoger algún subconjunto de esta lista de funcionalidades para implementar. Esta parte no te otorgará más de 1.5 puntos para la nota.

Ahora agregaremos funcionalidades adicionales a nuestra shell:

- **(1.5 pts)** Redirección stdin/stdout usando `<` y `>`. Ejemplo: `sort < input.txt > output.txt`. No es necesario implementar otras variantes que hay en Bash, como `2>`, `&>`, etc.
- **(1.5 pts)** Pipes, igual que en Bash. Ejemplo: `cat logs.txt | grep error`. El pipe toma el stdout de un comando y se lo entrega como stdin al siguiente.
- **(1 pt)** Manejo de quotes (`"`) y escaping (`\`) en los argumentos. Esto permite que todo lo siguiente funcione:
  - `cat "Mi archivo con espacios"`
  - `cat Mi\ archivo\ con\ espacios`
  - `echo "Comillas dobles: \" y un backslash \\"`

  Solo es necesario implementar quoting con comillas dobles (`"`) y escaping para comillas dobles (`\"`), backslash (`\\`) y espacios (`\ `).
- **(1 pt)** Historial persistente (`~/.shell-history`). Esto significa guardar el historial en el archivo `~/.shell-history`, y extender el builtin `history` para que lea desde ese archivo. Esto permite que al salir de la shell y volver a abrirla el historial se preserve.
- **(1 pt)** Globbing sencillo con `*`. Para los ejemplos imaginemos que tenemos los archivos `data1.txt`, `data2.png` y `data3.txt` en el directorio actual.
  - `ls *.txt`: se expande a `ls data1.txt data3.txt`
  - `cat data*`: se expande a `cat data1.txt data2.png data3.txt`
  - `echo *`: se expande a `echo data1.txt data2.png data3.txt`
  - `rm data*.png` se expande a `rm data2.png`

  En esta versión simplificada consideramos que:
  - Solo necesitas implementar la expansión del `*`. No consideramos otros comodines como `?` o `[]`.
  - Basta con que funcione solo en el directorio actual (no es necesario que `cat dir1/dir2/*` funcione).
  - El patrón `*` solo se puede usar una vez en el comando.
- **(0.5 pts)** Sustitución de variables de entorno: Agregamos sustitución de variables de entorno usando `$NOMBRE`, al igual que en Bash.

  Ejemplo de uso:

```shell
export API_KEY=KEY_SUPER_SECRETA
echo $API_KEY # imprime KEY_SUPER_SECRETA
```

  Si escogiste implementar escaping y quoting también debes agregar el escape de `$` usando `\$`.
- **(0.5 pts)** `cd` mejorado: implementamos `cd` sin argumentos (lleva al *home directory*) y `cd -` (lleva al directorio anterior):

```shell
cd /home/user/fotos # esto funciona de la P3
cd # cd sin argumentos nos devuelve al home directory (/home/user)
cd - # cd - nos lleva al directorio anterior, en este caso /home/user/fotos
```

- **(0.5 pts)** Ejecución en segundo plano si un comando termina en `&`. Debe mostrar el stdout (igual que en Bash).

Entregables

Deberán entregar un archivo `.tar` que contenga el codigo fuente de su proyecto y cualquier otro que sea necesario para la ejecución de su programa, además de un README.md con las instrucciones para compilar y eventualmente ejecutar su proyecto.