Práctica de Sistemas Operativos

v1.1

Descripción

La práctica consiste en la realización de un intérprete de comandos simple para el SO Linux en el lenguaje de programación C. El trabajo de dicho programa consiste en leer líneas de la entrada estándar y ejecutar los comandos pertinentes. Para leer de la entrada estándar es recomendable usar `stdio` (véase `fgets(3)`).

Nótese que el programa debe funcionar correctamente si su entrada estándar procede de un fichero y no de un terminal.

Para solicitar la ejecución del programa `comando` con un número indeterminado de argumentos `arg1 arg2...` la línea de comandos tendrá la siguiente sintaxis:

```
comando arg1 arg2 ...
```

Nótese que podría no haber ningún argumento.

El comando podrá ser, en este orden de prioridad, un *builtin* de el shell, un fichero ejecutable que se encuentre en el directorio de trabajo o un fichero ejecutable que se encuentre en alguno de los directorios de la variable `PATH`. No se permite utilizar `pexec` en la implementación del shell.

Una línea de comandos podrá describir en algunos casos un pipeline como en este ejemplo:

```
cmd1 arg11... | cmd2 arg21 arg22... | cmd3 arg31... | ...
```

Dicho pipeline habrá de ejecutarse creando un proceso por cada comando, de tal modo que la salida estándar de cada comando se utilice (mediante un pipe) como entrada estándar del siguiente. En general, un pipeline se debe considerar como un único comando: se considera que ha terminado cuando ha terminado el último comando, se considera que ha fallado cuando ha fallado el último comando, etc.

Un comando puede tener una indicación de redirección de entrada estándar como se muestra en este ejemplo:

```
cmd1 args... < fichero
```

que indica que la entrada del comando ha de proceder de `fichero`.

Un comando puede tener una indicación de redirección de salida estándar con esta sintáxis

```
cmd1 args... > fichero
```

en cuyo caso la salida estándar del comando tiene como destino `fichero` . En el caso de un pipeline sólo

se permite dicha redirección en el último comando, por razones obvias.

El shell deberá esperar a que una línea de comandos termine su ejecución antes de leer la siguiente, a no ser que dicha línea termine en &, como en este ejemplo:

```
cmd1 args | cmd2 args... | ... &
```

Cuando la línea termina en & el shell no debe esperar a que termine de ejecutar dicho comando. Además, no debe permitirse al comando ejecutado leer de la entrada del shell. Así pues, en caso de que no se haya redirigido la entrada estándar de dicho comando a otro fichero, el shell debe redirigirla a `/dev/null` para evitar que tanto el shell como el comando lean a la vez de la entrada (normalmente la consola).

Un ejemplo de línea de comandos que incluye todos los elementos mencionados sería:

```
cat | grep ex | wc < /tmp/fich >/tmp/out &
```

El shell debe implementar un comando llamado `cd` que debe cambiar el directorio actual al indicado como argumento en dicho comando. Si no recibe argumentos, cambiará el directorio de trabajo al directorio `HOME` del usuario.

El shell debe entender el comando

```
x=y
```

que deberá dar el valor `y` a la variable de entorno `x`. Para cualquier variable de entorno, el shell deberá reemplazar `$x` por el valor de `x`. Por ejemplo:

```
cmd=ls
arg=/tmp
$cmd $arg
```

es equivalente a ejecutar el comando

```
ls /tmp
```

Si la variable no existe, el shell debe imprimir un mensaje de error informativo y no ejecutar la línea de comandos. Por ejemplo (suponiendo que la variable de entorno patata no existe):

```
echo $patata
```

debería dar un error:

```
error: var patata does not exist
```

Entre los caracteres especiales de el shell (pipe, redirección, etc.) y los comandos/argumentos podrá haber cero, uno, o más espacios y/o tabuladores. El shell debe funcionar correctamente en cualquier caso.

El intérprete debe funcionar correctamente cuando se alimenta su entrada estándar desde otro proceso o

directamente de un fichero, esto es, cuando no se está usando de forma interactiva a través de la consola. Dos ejemplos:

```
cat mi_script | shell
shell < mi_script
```

siendo `mi_script` un fichero con varias líneas de comandos.

Opcional 1

Las líneas que no utilizan & ni redirección de entrada o salida, podrán terminar en "`[` ", para indicar un *here document*. En este caso, el comando utilizará como entrada estándar las líneas escritas por el usuario hasta aquella que conste sólo de "`]` ". Un ejemplo de este uso es el que sigue:

```
cat | wc -l [
una y
otra lin.
]
```

que ejecuta

```
cat | wc -l
```

de tal modo que la entrada estándar de `cat` recibirá las líneas

```
una y
otra lin.
```

como entrada.

No se permite utilizar ficheros temporales para implementar esta opción.

Opcional 2

Requiere implementar también la opción anterior.

Al ejecutar una línea como la que sigue:

```
a % ls -l
```

el contenido de la variable de entorno `a` será la salida estándar del comando que se indica después del carácter `%`. En este caso, `a` contendría la salida del comando `ls -l`. Detrás del carácter `%` puede escribirse un pipeline y debe de funcionar del mismo modo.

Opcional 3

Requiere implementar también las opciones anteriores.

El shell debe definir una variable de entorno `result` que debe estar siempre disponible y debe contener el estatus devuelto por el último comando (equivalente a `$?` en el shell de UNIX).

El shell debe incluir los siguientes comandos builtin: `ifok`, `ifnot` .

El comando `ifok` deberá ejecutar sus argumentos como un comando sólo si el comando anterior terminó su ejecución correctamente. Para esto debe utilizar la variable `result`. Por ejemplo:

```
test -e /tmp
ifok ls -l /tmp
```

Deberá ser equivalente a ejecutar `ls -l /tmp` cuando el directorio `/tmp` existe.

El comando `ifnot` es similar, pero ejecuta el comando sólo si el comando anterior falló.