



Campus Estado de México
Laboratorio de Sistemas Operativos
Profesora Bárbara Cervantes G.

Práctica IV

Saúl Figueroa Conde. A01747306.
René García Avilés. A01654359.

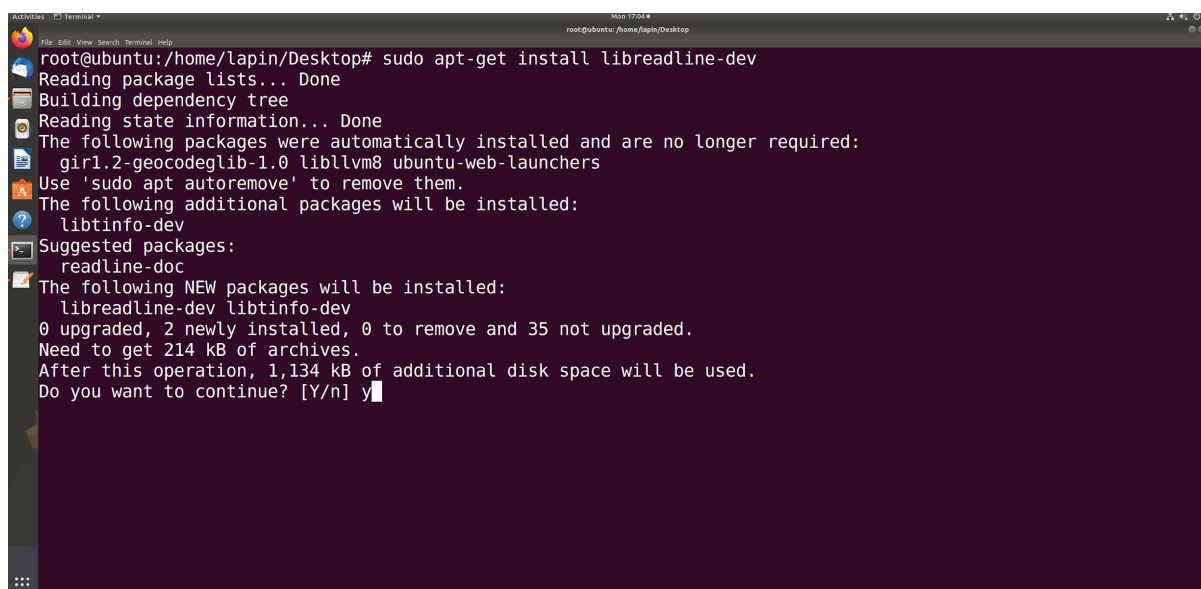
13 de marzo del 2020

Introducción

En este trabajo se exploró cómo desarrollar un propio *shell* programado en C para explorar de mejor manera cómo funcionan los comandos en Linux. Se manejaron a mayor profundidad la creación de procesos padre e hijo con `fork()`, se demostró el uso adecuado de apuntadores y la utilidad de desplegar un historial de comandos utilizados. El proceso hijo ejecutó comandos especificados en estos casos.

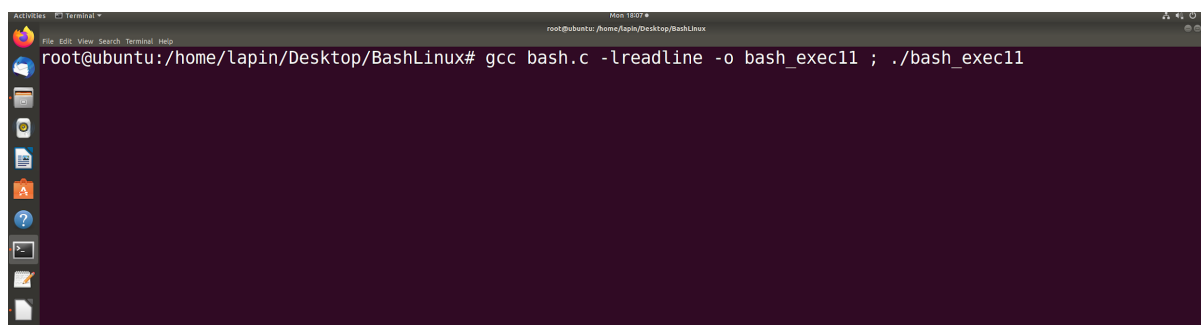
Desarrollo

Para la creación del propio *shell*, primero se instaló una librería que nos ayudó mucho. Se llama "readline" y hace que desarrollar el diseño de ciertos comandos sea mucho más sencillo. En otras palabras, sirve para el manejo de funciones en el *shell*:



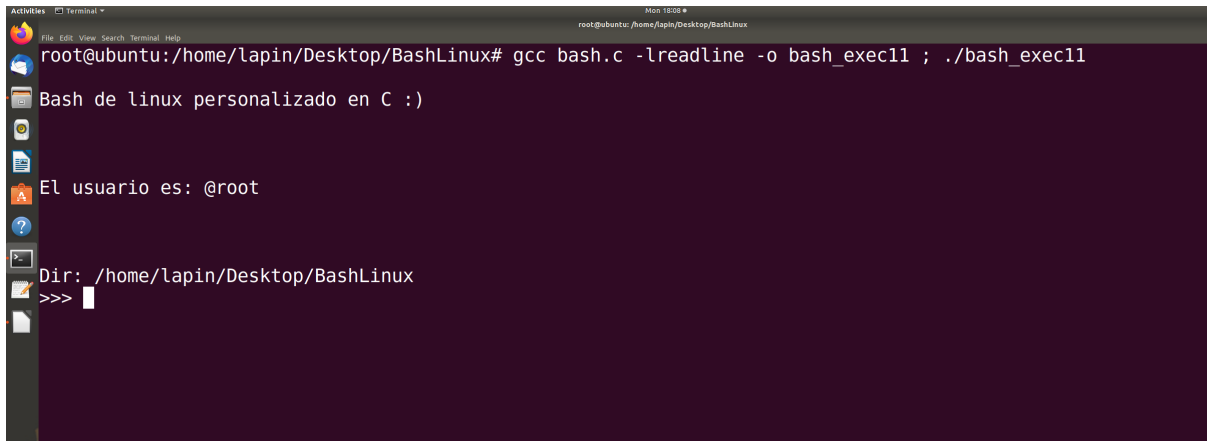
```
root@ubuntu:/home/lapin/Desktop# sudo apt-get install libreadline-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  gir1.2-geocodeglib-1.0 libllvm8 ubuntu-web-launchers
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  libtinfo-dev
Suggested packages:
  readline-doc
The following NEW packages will be installed:
  libreadline-dev libtinfo-dev
0 upgraded, 2 newly installed, 0 to remove and 35 not upgraded.
Need to get 214 kB of archives.
After this operation, 1,134 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

Imagen 1.- Se instaló la librería de 'readline'



```
root@ubuntu:/home/lapin/Desktop/BashLinux# gcc bash.c -lreadline -o bash_exec11 ; ./bash_exec11
```

Imagen 2.- Se compiló el archivo en C con ayuda de la librería instalada. Obsérvese la opción `-lreadline` que es importante para compilar el comando correctamente.

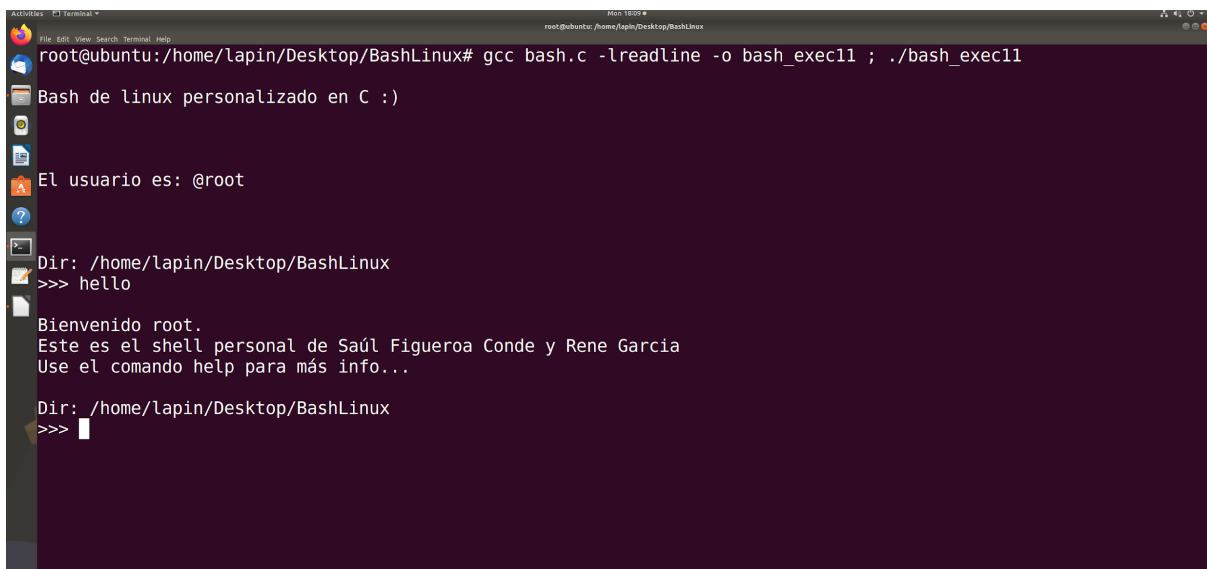


```

root@ubuntu:/home/lapin/Desktop/BashLinux# gcc bash.c -lreadline -o bash_exec11 ; ./bash_exec11
Bash de linux personalizado en C :)
El usuario es: @root
Dir: /home/lapin/Desktop/BashLinux
>>>

```

Imagen 3.- Se da la bienvenida “personalizada”. Ahora ya se pueden ejecutar comandos en el shell.

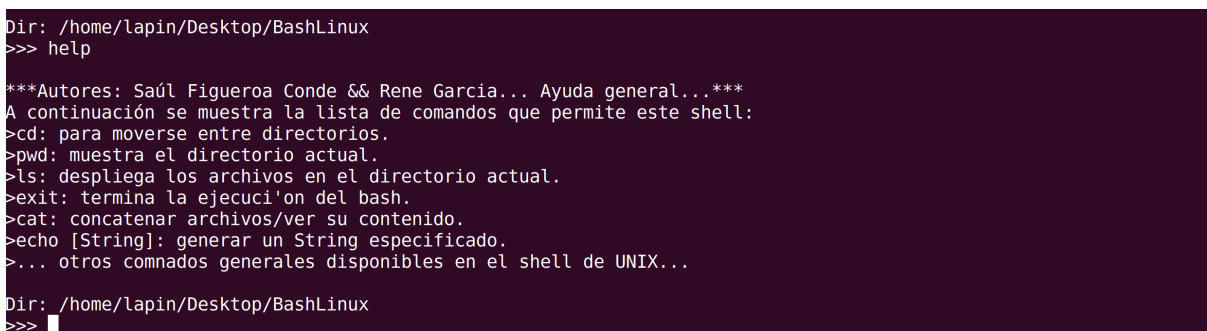


```

root@ubuntu:/home/lapin/Desktop/BashLinux# gcc bash.c -lreadline -o bash_exec11 ; ./bash_exec11
Bash de linux personalizado en C :)
El usuario es: @root
Dir: /home/lapin/Desktop/BashLinux
>>> hello
Bienvenido root.
Este es el shell personal de Saúl Figueroa Conde y Rene Garcia
Use el comando help para más info...
Dir: /home/lapin/Desktop/BashLinux
>>>

```

Imagen 4.- Comando “hello”, comando personalizado que proporciona la bienvenida al usuario y sugiriendo ejecutar el comando help.



```

Dir: /home/lapin/Desktop/BashLinux
>>> help
***Autores: Saúl Figueroa Conde && Rene Garcia... Ayuda general...***
A continuación se muestra la lista de comandos que permite este shell:
>cd: para moverse entre directorios.
>pwd: muestra el directorio actual.
>ls: despliega los archivos en el directorio actual.
>exit: termina la ejecución del bash.
>cat: concatenar archivos/ver su contenido.
>echo [String]: generar un String especificado.
>... otros comandos generales disponibles en el shell de UNIX...
Dir: /home/lapin/Desktop/BashLinux
>>>

```

Imagen 5.- Comando “help” proporciona información general al usuario sobre los comandos disponibles.

```

Dir: /home/lapin/Desktop/BashLinux
>>> cd ..

Dir: /home/lapin/Desktop
>>> ls
'2018-Operating System Concepts-10th.pdf'  bashExample_exec  codigos2  Procesos  worksExec
bash                                         bash_exec        hello.txt  running.c
bash.c                                     BashLinux       myBash    running_exec
bashExample.c                             codigos         names2    tester.txt

Dir: /home/lapin/Desktop
>>> cd BashLinux

Dir: /home/lapin/Desktop/BashLinux
>>> ls
bash.c  bash_exec11

Dir: /home/lapin/Desktop/BashLinux
>>>

```

Imagen 6.- Comando “cd” el cual permite moverse entre directorios

Comando “ls” el cual despliega los archivos en el directorio actual. Su implementación está documentada en el archivo de código fuente (sección donde se emplean *switches* y *cases* para lista de comandos). Para ello no se accedió al código en /bin de cada comando (como en el ejemplo de ls), sino que se buscó la manera adecuada de implementarlo en C realizando distintas llamadas a funciones/servicios del sistemas con la librería *readline* importada.

```

bash.c                                     BashLinux       myBash    running_exec
bashExample.c                             codigos         names2    tester.txt

Dir: /home/lapin/Desktop
>>> cd BashLinux

Dir: /home/lapin/Desktop/BashLinux
>>> ls
bash.c  bash_exec11

Dir: /home/lapin/Desktop/BashLinux
>>> touch texto.txt

Dir: /home/lapin/Desktop/BashLinux
>>> ls
bash.c  bash_exec11  texto.txt

Dir: /home/lapin/Desktop/BashLinux
>>> cat texto.txt

Dir: /home/lapin/Desktop/BashLinux
>>> file texto.txt
texto.txt: empty

Dir: /home/lapin/Desktop/BashLinux
>>>

```

Imagen 7.- Se utiliza el comando “touch” para generar un documento del tipo establecido en el directorio actual

Comando “cat” muestra el contenido en el archivo seleccionado

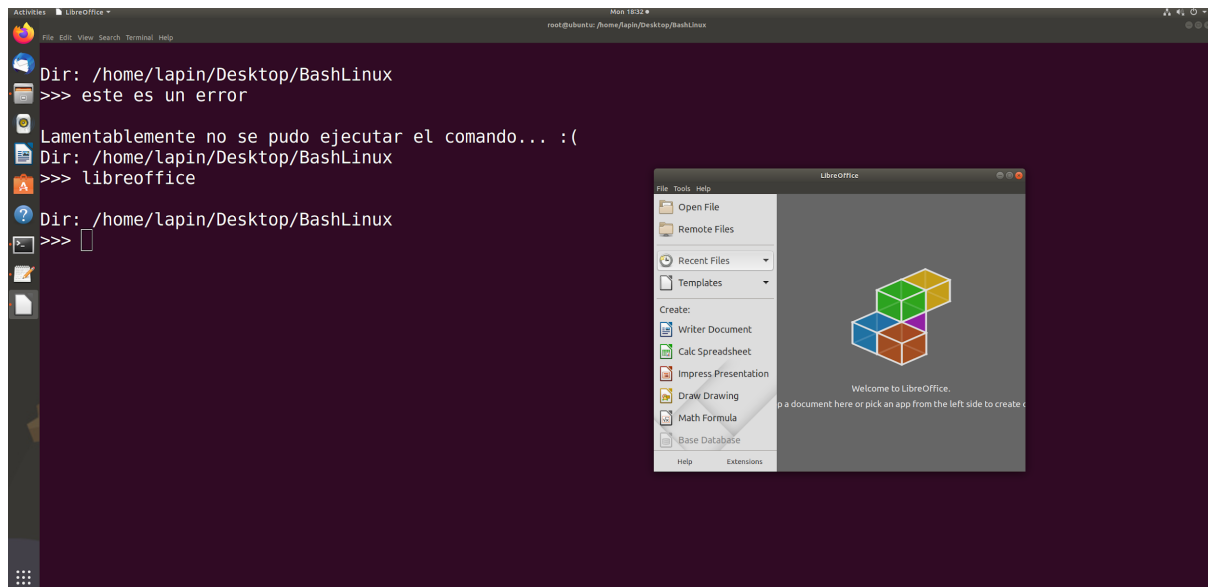
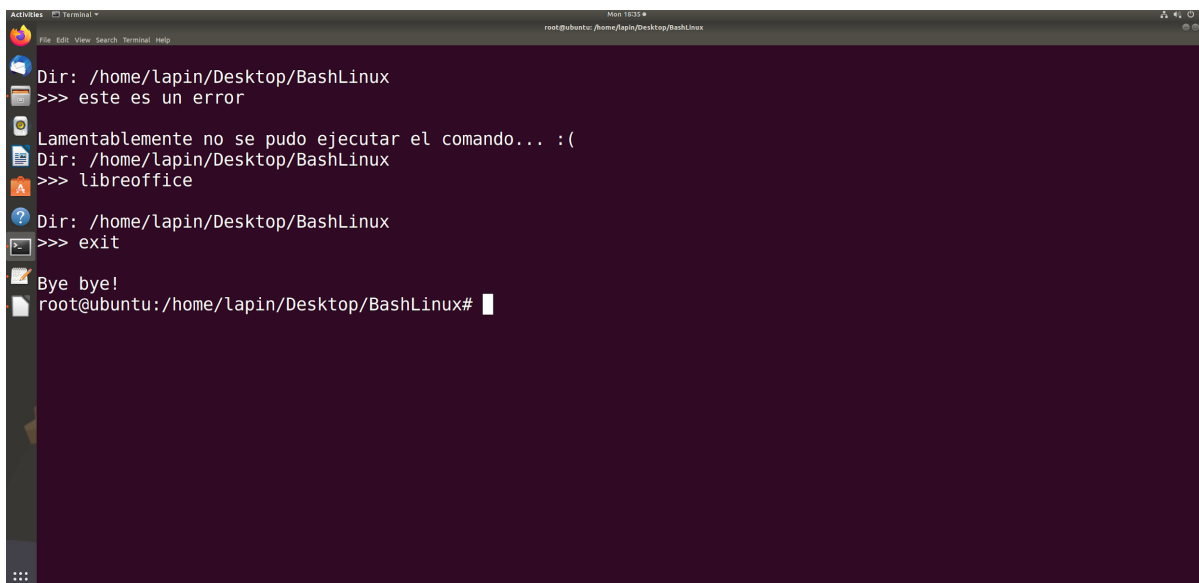


Imagen 8.- Se muestra el manejo de errores personalizado, al igual que el abrir una aplicación externa desde el prompt. También se muestra cómo se abre una aplicación en segundo plano, en este caso LibreOffice. Todos los comandos que puedan ejecutarse en segundo plano (es decir, comandos “inmediatos” como `ls` y `cd` no siguen esta norma ya que no es necesarios ejecutarlos en segundo plano) lo hacen por defecto. La filosofía que sigue esta implementación es que con las capacidades del equipo de cómputo y los requisitos del SO, no hay razón por la cual no sea buena práctica siempre ejecutar los comandos en segundo plano, ya que esto no afecta al *performance* del sistema. Para implementar esto, el proceso padre no hace `wait()` del hijo, pero no se crea un proceso zombie ya que gracias a la librería importada de *readline*, estos hijos “huérfanos” son identificados por el sistema y se terminan una vez que se cierra la aplicación o función que están ejecutando en segundo

plano. Esta es una alternativa al tener que usar todo el tiempo la opción del carácter “&”.



```

Dir: /home/lapin/Desktop/BashLinux
>>> este es un error
Lamentablemente no se pudo ejecutar el comando... :(
Dir: /home/lapin/Desktop/BashLinux
>>> libreoffice
Dir: /home/lapin/Desktop/BashLinux
>>> exit
Bye bye!
root@ubuntu:/home/lapin/Desktop/BashLinux#
  
```

Imagen 9.- Comando “exit” termina la ejecución del shell, con un mensaje personalizado

Familia de funciones *exec()*

Esta familia de funciones consta del comando “exec” proseguido por ‘l’ o ‘v’ y posteriormente por ‘e’ o ‘p’, las cuales tienen la característica de reemplazar el proceso en la ruta actual por el que sea especificado por dicho comando.

Este es el motivo por el cual el comando “exec” suele ser acompañado por la llamada al sistema “fork”.

Esta familia de comandos recibe primero por convenio, el fichero en donde será ejecutado, cada función requiere de distintos argumentos, sin embargo para todas, cada argumento es separado por comas (,) y se escriben entre comillas (“”), para así parecerse a los espacios que serían ingresados en el cmd y que este reconozca los comandos, el último argumento que reciben debe de ser “NULL” para señalar que ya no serán ingresados nuevos argumentos en dicha línea.

Para esta práctica de laboratorio se utilizó, el comando “**execvp**”

Este comando recibe dos parámetros fundamentales, los cuales son:

1. String que contiene el nombre del archivo a ser ejecutado
2. Un apuntador a un array de caracteres string

Este comando, al recibir la dirección y el “string” en donde se especificará el comando del shell a realizar, lo cual asemeja a escribir dicho comando en el shell.

Esta función permite una implementación relativamente sencilla ya que cuenta con una característica especial:

Facilita el manejo y threads repetidos y/o procesos, ya que al llamar a la función, todos los anteriores son inmediatamente destruidos y el nuevo proceso es cargado y ejecutado, sin la necesidad de llamar a un destructor extra.

Una función alternativa para la implementación de esta práctica sería “**execve**”

Esta función en primera instancia recibe de igual manera la dirección del fichero en donde se ejecutará el proceso.

Este comando como segundo parámetro requiere de un archivo binario ejecutable o un script válido (es decir, que cuente con el intérprete necesario). La dirección especificada en el primer argumento funge como apuntador, ya que será esa dirección en donde se lleve a cabo el proceso.

Esta función, en caso de que se le sea negado un permiso, continuará buscando en el resto del camino hasta que ejecute su instrucción, en caso contrario devolverá un EACCES.

Conclusiones

En este trabajo se exploró cómo funciona un *shell* de la forma más interactiva posible, creando uno personalizado desde cero. Este trabajo ayudó a explorar el manejo de procesos (creación, terminación e incluso ejecución de código). Se observó el valor de conocer y manejar de manera oportuna tanto las llamadas a sistema como los procesos que se puedan generar, para tener un manejo óptimo del sistema operativo. Encontramos este trabajo muy interesante, ya que ayudó a desarrollar más habilidades de programación en C y sirvió para fortalecer varios de los conceptos teóricos, vistos durante clase, acerca del manejo de procesos.

Referencias

1. http://linuxcommand.org/lc3_lts0010.php
2. <http://trajano.us.es/~fjfj/shell/shellscrip.htm>
3. <http://docencia.udea.edu.co/cci/linux/dia8/shell.html>
4. <https://www.geeksforgeeks.org/making-linux-shell-c/>