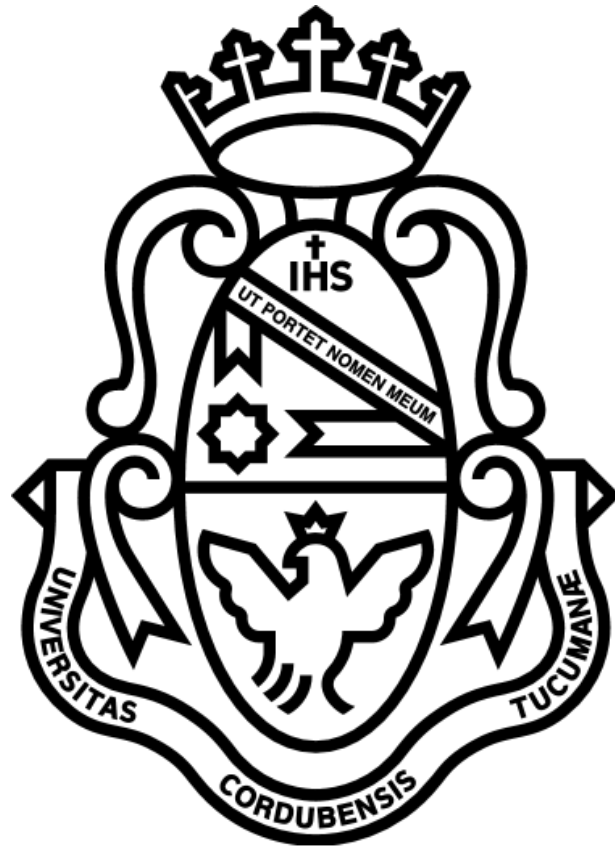


**UNIVERSIDAD NACIONAL DE CÓRDOBA**

**Facultad de Ciencias Exactas, Físicas y Naturales**



## **TRABAJO PRÁCTICO Nº2**

**Sistemas Operativos I**

**Integrantes:**

Giles García, Arian

**Año: 2015**

Tabla de contenido

Introducción .....3

Desarrollo .....3

    Parte A .....3

    Parte B .....6

    Parte C .....7

Conclusión .....9

# Introducción

---

El trabajo consiste en realizar intérprete de línea de comandos. Este programa, que se ejecuta en modo usuario, funciona en cualquier UNIX que soporte interface de caracteres y su función es aceptar comandos ingresados por entrada estándar (teclado), parsearlos, ejecutar la orden y mostrar el resultado en la salida estándar (pantalla), para luego volver a repetir el proceso.

## Desarrollo

---

### Parte A

Escriba un programa en "C" que actúe como un shell de intérprete de línea de comandos para el sistema operativo Linux. Su programa deberá utilizar la misma sintaxis de Bourne shell para correr programas.

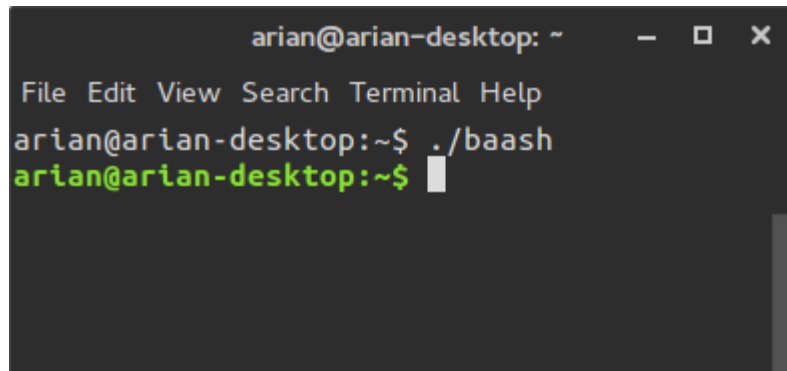
```
comando argumento1 argumento2 ... ,
```

efectuando la búsqueda del comando según se presente como path absoluto, relativo o involucre una búsqueda en la secuencia de la variable de entorno PATH. El programa deberá generar un proceso hijo para correr el comando, a fin de protegerse de comportamientos malos, y pasarle los argumentos correspondientes. Es preferible imprimir un prompt con información relevante a la izquierda de donde se introduce el comando. El nombre del host, el nombre del usuario y el camino corriente pueden ser algunos ejemplos. Los únicos comandos internos que se piden son exit, que termina con la ejecución del intérprete de comandos, y cd (cambiar de directorio actual). Fíjense en la llamada al sistema chdir(). Luego de implementar esta parte, su baash deberá ser capaz de invocar desde simples peticiones como date, que involucran búsquedas en PATH, hasta llamadas a comandos de compilación o linking utilizando path relativos y decenas de opciones.

### Resolución

Los primeros pasos para comenzar a realizar el 'baash' fue comenzar con la impresión del 'prompt'. El prompt que se imprime es de la forma **username@hostname:path\$** donde **username** es el nombre del usuario del sistema que se obtuvo con la función **getlogin\_r()**,

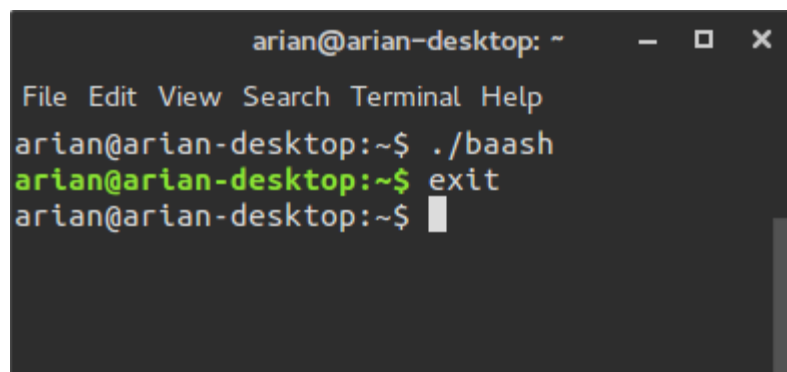
**hostname** es el nombre de la pc que se obtuvo con la función **gethostname()**, y **path** es el directorio actual donde se está trabajando, que se obtiene con la función **getcwd()**.



```
arian@arian-desktop: ~  
File Edit View Search Terminal Help  
arian@arian-desktop:~$ ./baash  
arian@arian-desktop:~$
```

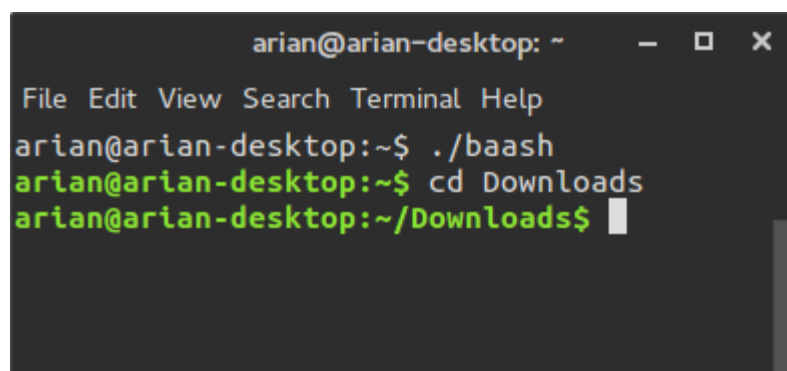
*Ilustración 1 - Impresión del Prompt*

A continuación, se creó un bucle infinito, donde se imprimía el prompt y se esperaba por comandos. Luego se procedió con la detección y ejecución de los comandos 'built-in'. Dado los comandos ingresados por el usuario, se los parsea y se ve si corresponden con los comandos built-in requeridos, que eran el comando **exit**, que detiene la ejecución del baaash y también **cd [path]** que cambia el directorio de trabajo. Para el cambiar de directorio, se utilizó la llamada a sistema **chdir()**, y para salir del programa se hizo con un simple **exit()**.



```
arian@arian-desktop: ~  
File Edit View Search Terminal Help  
arian@arian-desktop:~$ ./baash  
arian@arian-desktop:~$ exit  
arian@arian-desktop:~$
```

*Ilustración 2 - Built-In 'exit'*



```
arian@arian-desktop: ~  
File Edit View Search Terminal Help  
arian@arian-desktop:~$ ./baash  
arian@arian-desktop:~$ cd Downloads  
arian@arian-desktop:~/Downloads$
```

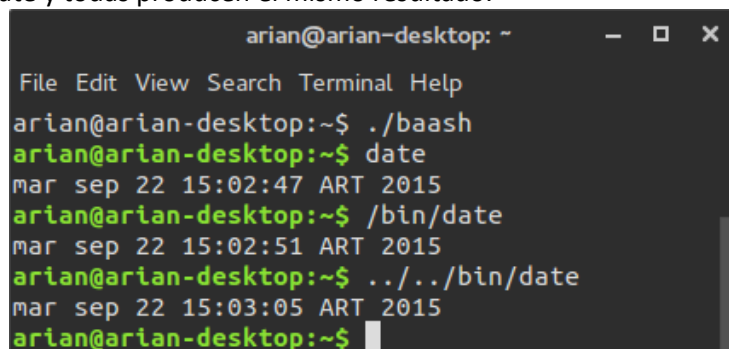
*Ilustración 3 - Built-In 'cd'*

Como siguiente paso, se creó una **función parse\_arguments(char \*command, char \*\*argv, int \*argc)** que dado un comando, lo parsea y guarda sus argumentos en variables auxiliares que utilizaremos luego para ejecutar dichos comandos. Para parsear los argumentos, simplemente se dividió el comando ingresado en espacios, y se guardó cada uno de estos argumentos en la variable argv y la cantidad de argumentos en argc.

Una vez realizado el parseo de argumentos, necesitamos ejecutar los comandos. Tenemos que tener en cuenta que los comandos pueden ser rutas absolutas, relativas, o rutas que necesitemos encontrar en la variable de entorno PATH. Si el comando que se encuentra en **argv[0]** es una ruta absoluta, no necesitamos hacer nada. Si, en cambio, es una ruta relativa, la convertimos en una ruta absoluta utilizando la función **realpath()**. Si no se especificó ninguna ruta, debemos ver si podemos encontrar el comando en una de las rutas contenidas en la variable PATH, luego, como primera instancia, obtenemos el contenido de dicha variable con la función **getenv()**, para luego parsear la cadena de caracteres que nos devolvió esa función, dividiéndola en : y guardando cada ruta en **path\_array** y la cantidad de rutas en **path\_length**. Luego concatenando cada ruta con el comando y comprobando si existía el archivo en esa ruta verificamos si podemos ejecutar o no ese comando.

Una vez que conseguimos la ruta, solo queda ejecutar el comando. Para ello utilizamos uso de las funciones **fork()** y **execv()**. Utilizamos fork para crear una copia del proceso, luego desde el proceso hijo, utilizando la función execv para reemplazar la imagen del programa por la del programa que se quería ejecutar desde la línea de comandos. Como parámetros de execv, necesitábamos la ruta del ejecutable, que se encontraba en nuestro caso en my\_argv[0], y el vector de parámetros que es **my\_argv**. Luego desde el padre, esperamos que se termine la ejecución del hijo con la función **wait()**, y volvemos a imprimir el prompt para esperar otro comando.

Con esto, ya podemos ejecutar cualquier tipo de comandos, sin importar la cantidad de argumentos, ni el tipo de ruta que se le pasa a la línea de comandos. Por ejemplo, podemos ejecutar tanto date como /bin/date como así también parados en el home del usuario, hacer ../../bin/date y todas producen el mismo resultado.



```
arian@arian-desktop: ~  
File Edit View Search Terminal Help  
arian@arian-desktop:~$ ./baash  
arian@arian-desktop:~$ date  
mar sep 22 15:02:47 ART 2015  
arian@arian-desktop:~$ /bin/date  
mar sep 22 15:02:51 ART 2015  
arian@arian-desktop:~$ ../../bin/date  
mar sep 22 15:03:05 ART 2015  
arian@arian-desktop:~$
```

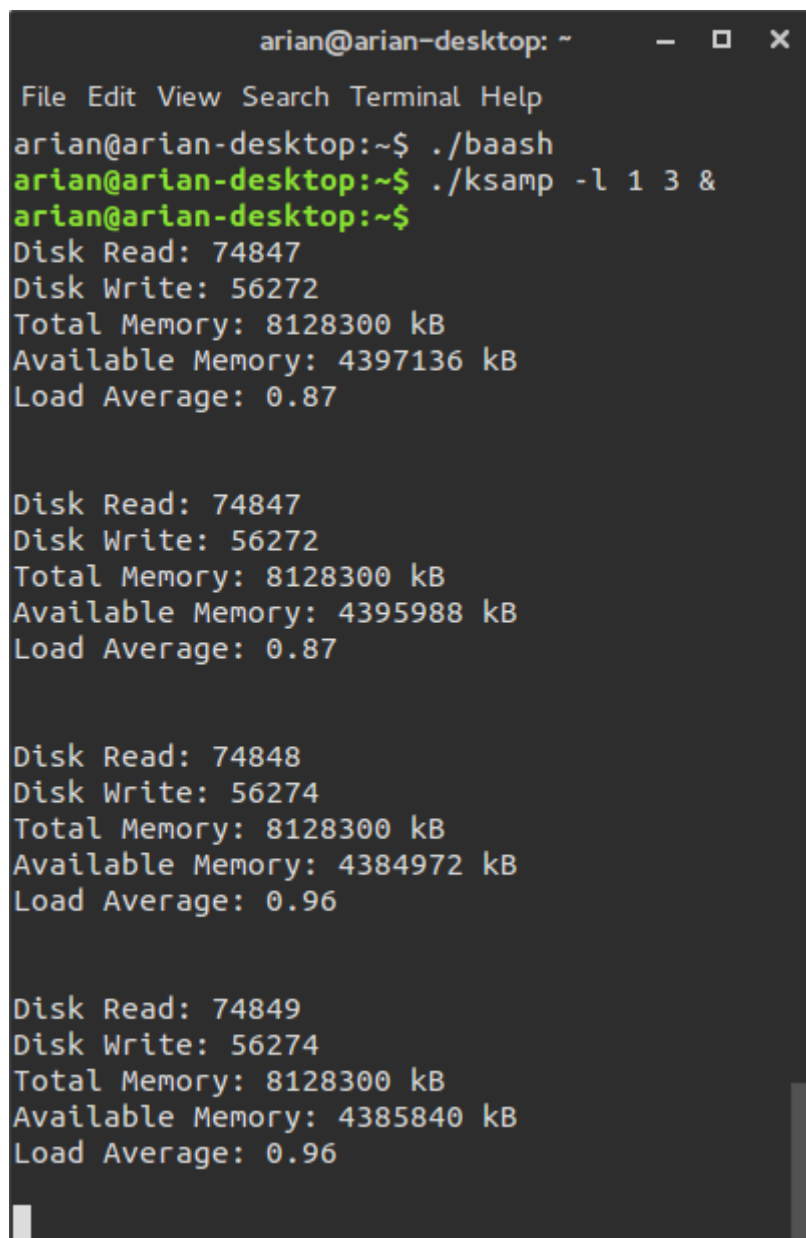
Ilustración 4 - Distintas formas de llamar a 'date'

## Parte B

Agregue al shell de la parte anterior, la funcionalidad de correr programas como procesos en background, o mejor dicho concurrentemente con el mismo baash. El usuario incluirá el operador '&' a la derecha del último argumento para activar esta capacidad.

## Resolución

Para el desarrollo de esta parte lo único que se tuvo que hacer, es leer los argumentos parseados, y si el último argumento era un ampersand **&**, se omitía la parte del **wait()** a la hora de esperar a que el hijo termine.

A screenshot of a terminal window titled 'arian@arian-desktop: ~'. The terminal shows a menu with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The user enters './baash' and then './ksamp -l 1 3 &'. The prompt changes to 'arian@arian-desktop:~\$' and then to 'arian@arian-desktop:~\$' again. The output shows system statistics: 'Disk Read: 74847', 'Disk Write: 56272', 'Total Memory: 8128300 kB', 'Available Memory: 4397136 kB', and 'Load Average: 0.87'. This output is repeated three times, indicating the script is running in the background and the user is still at the prompt.

```
arian@arian-desktop: ~  
File Edit View Search Terminal Help  
arian@arian-desktop:~$ ./baash  
arian@arian-desktop:~$ ./ksamp -l 1 3 &  
arian@arian-desktop:~$  
Disk Read: 74847  
Disk Write: 56272  
Total Memory: 8128300 kB  
Available Memory: 4397136 kB  
Load Average: 0.87  
  
Disk Read: 74847  
Disk Write: 56272  
Total Memory: 8128300 kB  
Available Memory: 4395988 kB  
Load Average: 0.87  
  
Disk Read: 74848  
Disk Write: 56274  
Total Memory: 8128300 kB  
Available Memory: 4384972 kB  
Load Average: 0.96  
  
Disk Read: 74849  
Disk Write: 56274  
Total Memory: 8128300 kB  
Available Memory: 4385840 kB  
Load Average: 0.96
```

Ilustración 5 - Uso del ampersand

## Parte C

Incorporen las funcionalidades de redirección y tuberías a baash, de manera tal que se interpreten los operadores

< filename,

> filename

| command2 argumento2\_1 argumento2\_2 ...

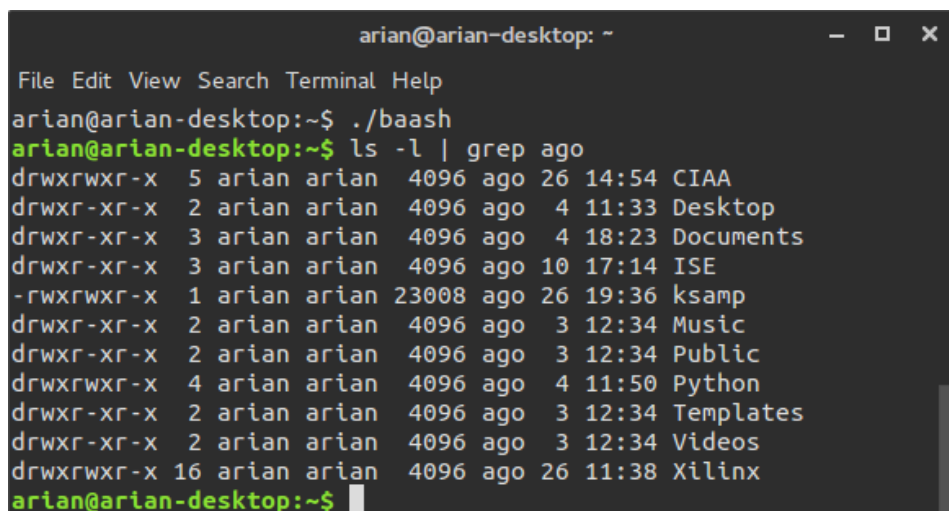
al final de los argumentos del comando.

Deberá soportar sólo uno de los cuatro operadores al mismo tiempo.

### Resolución

Como primera instancia se comenzó identificando que los comandos ingresados eran comandos con pipe o con algun redireccionamiento de entrada o salida. Para esto, se dividió el comando en |, > o <, y si la división era exitosa, estabamos frete alguno de los 3 casos.

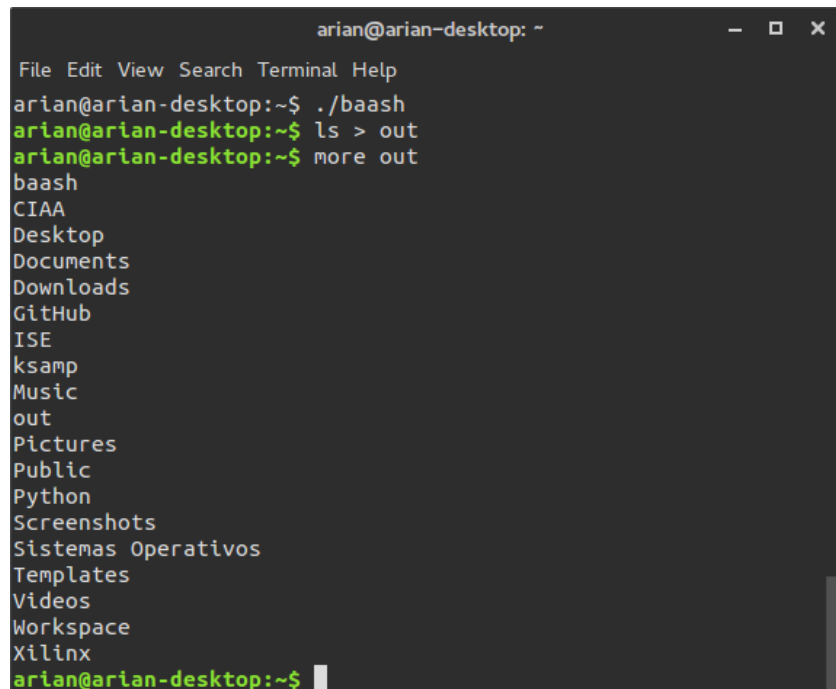
Para el caso del pipe |, parseamos el comando a cada lado por separado y guardabamos sus argumentos en variables diferentes, my\_argv y my\_argv\_2. Luego creamos un pipe con la función **pipe()**, y desde el padre, creabamos dos hijos, en el primero, redireccionabamos la salida al pipe, y ejecutaba la primera parte del comando, mientras que en el segundo, redireccionabamos la entrada del pipe y ejecutabamos el segundo comando. Luego desde el padre esperabamos que terminen de ejecutarse sus hijos, y volviamos a la ejecución normal del baash.



```
arian@arian-desktop: ~  
File Edit View Search Terminal Help  
arian@arian-desktop:~$ ./baash  
arian@arian-desktop:~$ ls -l | grep ago  
drwxrwxr-x 5 arian arian 4096 ago 26 14:54 CIAA  
drwxr-xr-x 2 arian arian 4096 ago 4 11:33 Desktop  
drwxr-xr-x 3 arian arian 4096 ago 4 18:23 Documents  
drwxr-xr-x 3 arian arian 4096 ago 10 17:14 ISE  
-rwxrwxr-x 1 arian arian 23008 ago 26 19:36 ksamp  
drwxr-xr-x 2 arian arian 4096 ago 3 12:34 Music  
drwxr-xr-x 2 arian arian 4096 ago 3 12:34 Public  
drwxrwxr-x 4 arian arian 4096 ago 4 11:50 Python  
drwxr-xr-x 2 arian arian 4096 ago 3 12:34 Templates  
drwxr-xr-x 2 arian arian 4096 ago 3 12:34 Videos  
drwxrwxr-x 16 arian arian 4096 ago 26 11:38 Xilinx  
arian@arian-desktop:~$
```

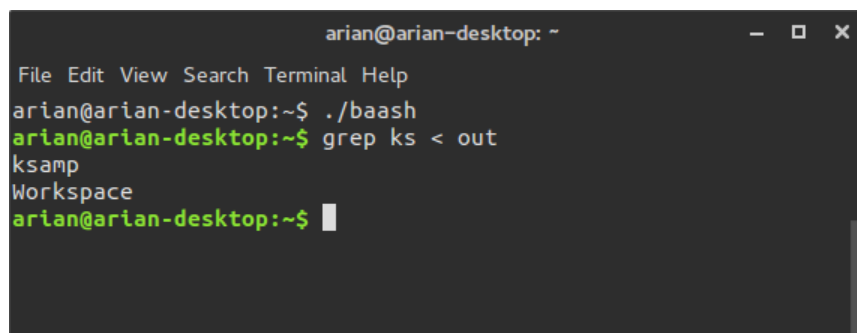
Ilustración 6 - Pipe |

Si el comando contiene un `>`, esto quiere decir que la salida del comando anterior al símbolo `>`, va a parar a un archivo, cuyo nombre esta después de dicho símbolo. Esto se hizo creando un hijo con `fork`, para luego desde el hijo, redireccionar su salida a un file descriptor que apuntaba al archivo señalado en el comando, y luego ejecutar el comando.

A terminal window titled 'arian@arian-desktop: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The user enters './baash' and then 'ls > out'. The prompt changes to 'arian@arian-desktop:~\$'. Then the user enters 'more out', and the terminal displays a list of directory contents: 'baash', 'CIAA', 'Desktop', 'Documents', 'Downloads', 'GitHub', 'ISE', 'ksamp', 'Music', 'out', 'Pictures', 'Public', 'Python', 'Screenshots', 'Sistemas Operativos', 'Templates', 'Videos', 'Workspace', and 'Xilinx'. The prompt returns to 'arian@arian-desktop:~\$'.

*Ilustración 7 - Redireccionamiento >*

Por último, si el comando contiene un `<`, esto quiere decir que la entrada del comando anterior al símbolo `<`, va a ser el contenido de un archivo, cuyo nombre esta después de dicho símbolo. Esto se hizo creando un hijo con `fork`, para luego desde el hijo, redireccionar su entrada a un file descriptor que apuntaba al archivo señalado en el comando, y luego ejecutar el comando.

A terminal window titled 'arian@arian-desktop: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The user enters './baash' and then 'grep ks < out'. The prompt changes to 'arian@arian-desktop:~\$'. The terminal then displays the output of the grep command: 'ksamp' and 'Workspace'. The prompt returns to 'arian@arian-desktop:~\$'.

*Ilustración 8 - Redireccionamiento <*



# Conclusión

---

Al concluir con la resolución de este trabajo práctico, se logró comprender el concepto de proceso, la forma en la que un proceso puede crear otro proceso. También fue muy útil utilizar y entender cómo hacer para comunicar un proceso con otro.