

INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

LABORATORIO: Sistemas Operativos

## Práctica 5

El shell y su ambiente básico de trabajo

LosDos

*Integrantes:*

Amanda Velasco Gallardo - 154415  
Carlos Octavio Ordaz Bernal - 158525

*Fecha(s) de elaboración de la práctica:*

22 de febrero de 2019

## Introducción

El intérprete de comandos, que es el encargado de traducir las órdenes de los usuarios a un conjunto de instrucciones que el núcleo del sistema pueda entender, puede ser a través de una interfaz gráfica, o con un programa conocido como línea de comandos. El primero combina el uso de ratón y teclado, mientras que el segundo sólo hace uso de las líneas que el usuario escribe utilizando solamente el teclado.

El **shell** es un intérprete de comandos. Es el encargado de validar las instrucciones introducidas por el usuario y cuenta con variables de ambiente que permiten personalizar el entorno de trabajo. Una característica que distingue a los **shell** del resto de los intérpretes es que éste es un lenguaje de programación que cuenta con instrucciones para la ejecución selectiva y repetitiva de grupos de comandos, cada programa escrito en un **shell** se le conoce como guión (*script*).

El primer **shell** estándar fue desarrollado por Steven Bourne en 1979, en los laboratorios Bell. Dentro un ambiente UNIX es muy sencillo cambiar de un **shell** a otro, por lo general se utilizan diferentes intérpretes dependiendo de las tareas que se desean realizar. Los **shell** pueden ejecutar comandos de tres diferentes formas:

- Directamente: el **shell** como intérprete ejecuta internamente el comando.
- Directamente a través de un **subshell**: el **shell** crea una copia de sí mismo, las variables del padre con heredadas al hijo sin modificar al padre, y el hijo ejecuta el comando.
- Indirectamente a través de un **subshell**: el hijo creado como copia del **shell** original, hace uso del comando **exec** para ejecutar un programa. Las variables de ambiente son las únicas heredadas, no las locales.

Las variables de **shell** son aquellas que permiten personalizar el ambiente de trabajo. Existen un conjunto de variables predefinidas y el usuario es capaz de definir sus propias variables. Existen dos tipos de variables de **shell**: las booleanas y las que pueden recibir un valor.

La principal ventaja de utilizar **shell** es la personalización, cada intérprete puede contener variables definidas por el usuario dependiendo de las tareas

que se vayan a realizar. Es posible decir que los `shell` vienen en varios sabores por lo diferentes que pueden ser en términos de la forma en que trabajan y sus variables definidas.

## Desarrollo

1.

- A) Ejecutar el comando `ps -l` y anotar los procesos incluyendo su PID respectivo: El primer comando de la figura 1 muestra el resultado.

```
ubuntu:~> ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2687  2681  0  80   0 -  1439 rt_sig pts/1    00:00:00 tcsh
0 R  1000  2765  2687  0  80   0 -  1177 -      pts/1    00:00:00 ps
ubuntu:~> sh
$ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2687  2681  0  80   0 -  1439 rt_sig pts/1    00:00:00 tcsh
0 S  1000  2767  2687  0  80   0 -   559 wait  pts/1    00:00:00 sh
0 R  1000  2768  2767  0  80   0 -  1177 -      pts/1    00:00:00 ps
```

Fig. 1: Despliegue del comando `ps -l` dentro y fuera del subshell.

- B) Crear temporalmente un **subshell** `sh`. Al estar dentro del proceso **subshell** ejecutar `ps -l`. ¿De quién es hijo el **subshell**? Al final terminar con éste: Es hijo de `tcsh`, lo anterior se puede observar en la figura 1, el segundo comando corresponde a la creación de un **subshell** `sh` y se pueden observar los procesos gracias a `ps -l`.

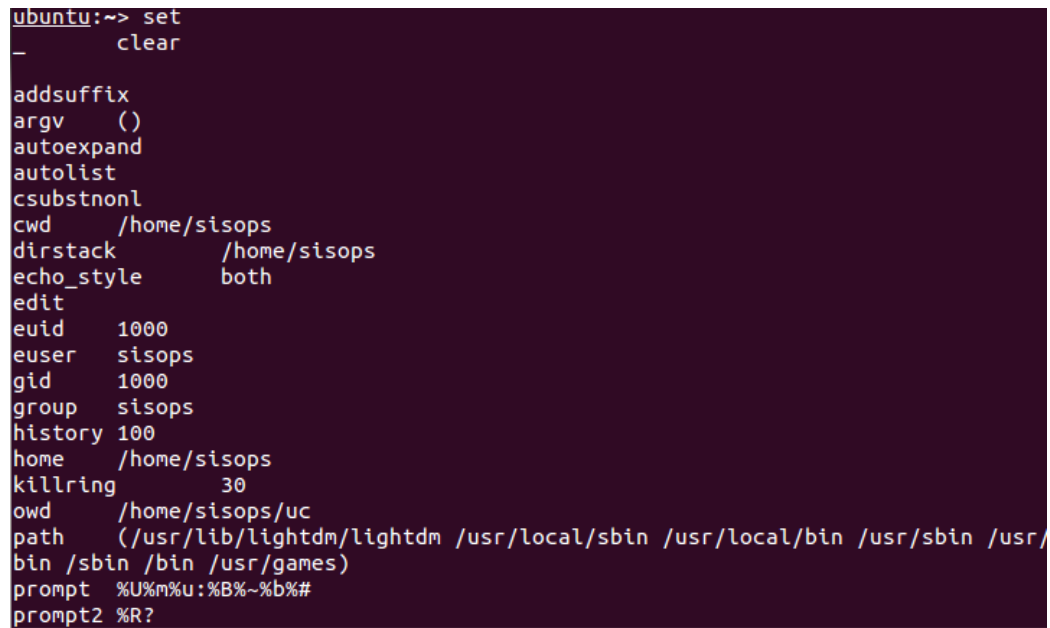
2. Crear temporalmente un subshell `csh`. Al estar dentro del proceso subshell ejecutar `ps -l`. Al final terminar con el subshell. La figura 2 muestra el comando `ps -l` para el subshell `csh`.

```
ubuntu:~% ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2687  2681  0  80   0 -  1439 rt_sig pts/1    00:00:00 tcsh
0 S  1000  2771  2687  0  80   0 -  1436 rt_sig pts/1    00:00:00 csh
0 R  1000  2775  2771  0  80   0 -  1177 -      pts/1    00:00:00 ps
```

Fig. 2: Despliegue del comando `ps -l`.

### 3. Desplegar las variables de shell.

La figura 3 muestra las variables de shell.



```
ubuntu:~> set
_      clear

addsuffix
argv   ()
autoexpand
autolist
csubstnonl
cwd     /home/sisops
dirstack      /home/sisops
echo_style    both
edit
euid      1000
euser     sisops
gid       1000
group     sisops
history   100
home      /home/sisops
killring   30
owd       /home/sisops/uc
path      (/usr/lib/lightdm/lightdm /usr/local/sbin /usr/local/bin /usr/sbin /usr/
bin /sbin /bin /usr/games)
prompt    %U%m%u:%B%~%b%#
prompt2   %R?
```

Fig. 3: Despliegue del comando `set`.

4. Primero, verificar que la variable de shell `filec` no este activada, entonces activarla. Recordar que la variable es de tipo *switch*; en esta distribución de Ubuntu dicha variable viene activada.

La siguiente figura 4 muestra que se activa la variable `filec`, y se verificó dentro de la lista de variables de shell.

```
ubuntu:~> set filec
ubuntu:~> set
_      set filec

addsuffix
argv   ()
autoexpand
autolist
csubstnonl
cwd    /home/sisops
dirstack    /home/sisops
echo_style    both
edit
euid    1000
euser   sisops
filec
```

Fig. 4: Despliegue del comando `ps -l`.

5. Activar la variable de shell `noclobber`, ver que aparezca en la lista de variables de shell y hacer una prueba de funcionamiento que no permita a un redirector reemplazar un archivo existente.

La primer figura 5, muestra el comando para activar la variable de shell y se verifica dentro de la lista que esté activada.

```
ubuntu:~> set noclobber
ubuntu:~> set
_      set noclobber

addsuffix
argv   ()
autoexpand
autolist
csubstnonl
cwd    /home/sisops
dirstack      /home/sisops
echo_style    both
edit
euid      1000
euser     sisops
filec
gid       1000
group     sisops
history   100
home      /home/sisops
killring      30
noclobber
```

Fig. 5: Activación de la variable noclobber.

La segunda figura 6, muestra que no se permitió modificar el archivo 12.txt ya existente.

```

ubuntu:~> ls
12.txt                examples.desktop      otro01.cc             SistemasOperativos
154415_158525_p04.txt hijofinal.cc          otro01.exe            subs.cc
9b.txt               hijofinal.exe        padre.cc              subs.exe
argumentos.cc        index.html            padre.cc~             Templates
Desktop              index.html.1         padre.exe             ua
Documents            index.html.2         pahi.src              uc
Downloads            mod1                 Pictures              uctar
ejer21.cc            mod2                 Public                Videos
ejer21.cc~           mod3                 resultados.txt        workspace
ejer21.exe           Music                resultados.txt~
ubuntu:~> echo "No hubo examen" > 12.txt
12.txt: File exists.

```

Fig. 6: Prueba de la variable noclobber.

6. Desactivar la variable de shell noclobber. Listar el total de variables activas y verificar que dicha variable ya no aparezca. También verificar que ya fue desactivado haciendo la misma prueba que en el inciso anterior.

La siguiente figura 7 muestra que no se devolvió ningún error al reemplazar el archivo 12.txt.

```

ubuntu:~> echo "No hubo examen" > 12.txt
ubuntu:~>

```

Fig. 7: Despliegue del comando ps -l.

7. Elaborar su propia variable de shell que contenga su propio nombre.

La figura 8 muestra el comando para crear las variables y el resultado que arroja nuestros nombres.

```

ubuntu:~> set nombre1='Octavio' nombre2='Amandine'
ubuntu:~> $nombre1
Octavio: Command not found.
ubuntu:~> $nombre2
Amandine: Command not found.

```

Fig. 8: Variables de shell con nuestros nombres.

## 8. Imprimir:

La siguiente figura 9 muestra lo solicitado a desplegar.

- El path del `shell` con una variable de éste:*
- El directorio base con una variable global (entorno o global):*

```
ubuntu:~> echo $path
/usr/lib/lightdm/lightdm /usr/local/sbin /usr/local/bin /usr/sbin /usr/bin /sbin
/bin /usr/games
ubuntu:~> echo $HOME
/home/sisops
```

Fig. 9: Resultados del path y del directorio base.

## 9. Desplegar los alias existentes.

La figura 10 muestra que no existe alias alguno definido.

```
ubuntu:~> alias
ubuntu:~>
```

Fig. 10: Despliegue del comando alias.

## 10. Crear alias `ldir`, este alias al ejecutarse deberá ser equivalente al comando `ls -l`.

La siguiente figura 11 muestra la creación del alias y su ejecución.

```
ubuntu:~> alias ldir 'ls -l'
ubuntu:~> ldir
total 192
-rw-rw-r-- 1 sisops sisops 15 Feb 22 09:22 12.txt
-r--r--r-- 1 sisops sisops 3562 Feb 15 09:48 154415_158525_p04.txt
-rw-rw-r-- 1 sisops sisops 150 Feb 15 09:19 9b.txt
-rw-rw-r-- 1 sisops sisops 320 Jan 14 04:30 argumentos.cc
drwxr-xr-x 2 sisops sisops 4096 Jun 4 2012 Desktop
drwxr-xr-x 2 sisops sisops 4096 Feb 19 11:15 Documents
drwxr-xr-x 2 sisops sisops 4096 Feb 21 11:42 Downloads
-rw-rw-r-- 1 sisops sisops 309 Feb 21 12:05 ejer21.cc
-rw-rw-r-- 1 sisops sisops 309 Feb 21 12:04 ejer21.cc~
-rwxrwxr-x 1 sisops sisops 7250 Feb 21 12:05 ejer21.exe
```

Fig. 11: Alias y su resultado.

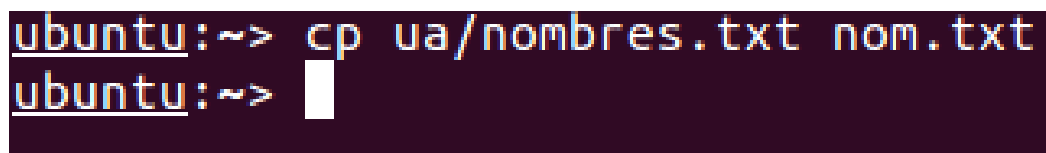


**11. ¿Qué hace el alias `cam: cd !*`; `echo $cwd`?**

El comando especificado se mueve al directorio base e imprime el directorio de trabajo.

**12. Hacer un ejemplo con el comando copiar (`cp`).**

A continuación, en la figura 12 se muestra la ejecución del comando `cp`.

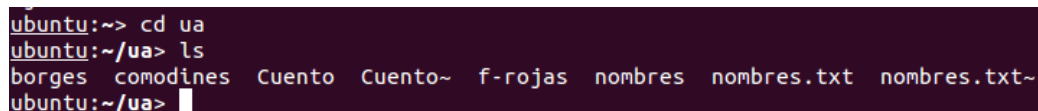


```
ubuntu:~> cp ua/nombres.txt nom.txt
ubuntu:~> 
```

Fig. 12: Ejecución del comando `cp`.

**13. Listar los nombres de los archivos que se encuentran en `ua`.**

A continuación, en la figura 13 se muestra la ejecución del comando `ls` en el directorio `ua`.

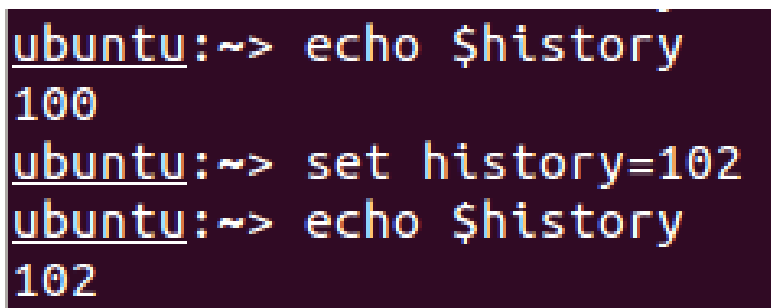


```
ubuntu:~> cd ua
ubuntu:~/ua> ls
borges comodines Cuento Cuento~ f-rojas nombres nombres.txt nombres.txt~
ubuntu:~/ua> 
```

Fig. 13: Ejecución del comando `ls` en el directorio `ua`.

**14. Mostrar el tamaño de la lista de eventos de `history` e incrementarla en dos. Comprobar que aumentó el tamaño de la lista.**

La figura 14 se muestra el valor de la lista de eventos original, y su valor incrementada en dos.



```
ubuntu:~> echo $history
100
ubuntu:~> set history=102
ubuntu:~> echo $history
102
```

Fig. 14: Valor de la lista `history` original e incrementado en dos.

15. Modificar la variable de shell prompt para que ahora despliegue su nombre, seguido del número de evento y los símbolos + >.

La figura 15 muestra el comando utilizado para cambiar la variable.

```
ubuntu:~> set prompt='A0[\\!]+>'
A0[4]+>
```

Fig. 15: Cambio en el prompt.

16. Buscar en la lista de eventos el comando utilizado para la pregunta 13. Con los mecanismos de sustitución de eventos, listar el contenido del directorio \$home/ua.

La siguiente figura 16 muestra la ejecución del comando `ls ua` por medio de repetición.

```
A0[4]+>history
 1  9:53  ls ua
 2  9:53  cp ua/nombres.txt nomnom.txt
 3  9:54  set prompt='A0[\\!]+>'
 4  9:54  history
A0[5]+>!1
ls ua
borges comodines Cuento Cuento~ f-rojas nombres nombres.txt nombres.txt~
A0[6]+>█
```

Fig. 16: Ejecución del comando `ls ua` por medio de repetición.

17. Inmediatamente después de haber ejecutado los comandos de las preguntas, ejecutar el siguiente comando y con ayuda del desplegado explicar qué hace: `!ls:0 -R1 !cp:$`.

El comando especificado muestra los archivos que han sido copiados. La siguiente figura 17 muestra el resultado del comando.

```
A0[32]+>!ls:0 -R1 !cp:$
ls -R1 NHE.txt
ls: cannot access '-R1': No such file or directory
NHE.txt
```

Fig. 17: Resultado del comando `!ls:0 -R1 !cp:$`.

18. Después de haber realizado del directorio con `ls`, utilizar únicamente el comando de repetición para realizar:

A) *Un listado largo del directorio base: !33 -l*

```
A0[40]>+!33 -l
ls -l
total 1944
-rw-rw-r-- 1 maker maker    15 Mar  4 07:26 12.txt
-r--r--r-- 1 maker maker 956671 Feb 12 19:19 152854-158279p04.txt
drwxrwxr-x 3 maker maker   4096 Apr  9 2018 Arduino
drwxr-xr-x 2 maker maker   4096 Mar  4 06:43 Desktop
```

Fig. 18: Resultado del comando `!33 -l`.

B) *Un listado largo del directorio raíz: !33 -l ~*

```
A0[45]>+!33 -l ~
ls -l ~
total 1944
-rw-rw-r-- 1 maker maker    15 Mar  4 07:26 12.txt
-r--r--r-- 1 maker maker 956671 Feb 12 19:19 152854-158279p04.txt
drwxrwxr-x 3 maker maker   4096 Apr  9 2018 Arduino
```

Fig. 19: Resultado del comando `!33 -l ~`.

C) *Un listado largo del directorio `$home/ua`: !33 -l ua*

```
A0[47]>+!33 -l ua
ls -l ua
total 44
-rw-rw-r-- 1 maker maker 14007 Feb 21 16:25 arch
drwxrwxr-x 2 maker maker  4096 Feb 21 15:02 borges
drwxrwxr-x 2 maker maker  4096 Feb 21 15:02 comodines
```

Fig. 20: Resultado del comando `!33 -l ua`.

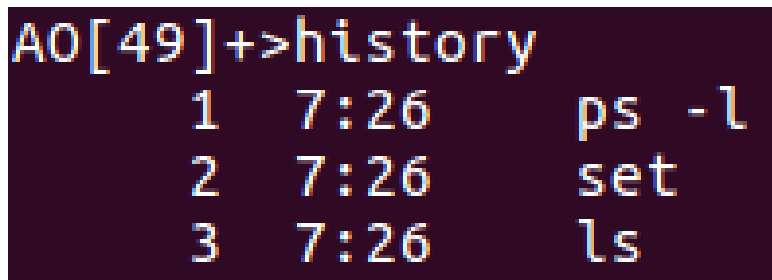
D) *Un listado largo del directorio `$home/uc` con una sola sustitución: `^ua ^uc`*

```
A0[48]>+^ua^uc
ls -l uc
total 8
-rw-r--r-- 1 maker maker 3781 Feb 15 2018 Ejercicios
-rw-r--r-- 1 maker maker  10 Feb 15 2018 x
-rw-r--r-- 1 maker maker   0 Feb 15 2018 x1y
```

Fig. 21: Resultado del comando `^ua ^uc`.

### 19. Mostrar los tres primeros eventos de history.

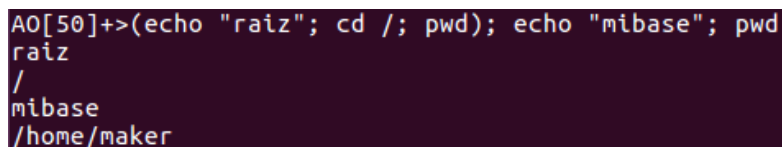
La figura 22 muestra los primeros tres eventos de history.



```
A0[49]+>history
      1  7:26      ps -l
      2  7:26      set
      3  7:26      ls
```

Fig. 22: Primeros tres eventos de history.

### 20. Ejecutar el comando: (echo 'raíz'; cd /; pwd); echo 'mibase'; pwd. La siguiente figura 23 muestra el resultado del comando.



```
A0[50]+>(echo "raíz"; cd /; pwd); echo "mibase"; pwd
raíz
/
mibase
/home/maker
```

Fig. 23: Resultado del comando.

- *¿Quién ejecuta los comandos que están entre ( )?:* Un subshell hijo de tcsh.
- *¿Para qué sirve el “;”?:* Para separar la ejecución de comandos cuando se escriben en una sola línea.
- *¿Explique que hizo todo el comando?:* La parte que está entre paréntesis desplegó en pantalla “raíz”, luego fue al directorio /, y mostró la ruta. La segunda parte desplegó en pantalla “mibase” y mostró la ruta hasta el directorio base (home).
- *¿Por qué los `pwd` reportan resultados distintos?:* Porque el primero se ejecutó para el subshell, y las variables del padre no fueron heredadas. Mientras que el segundo, fue ejecutado dentro del shell y sí conoce los valores de las variables.

21. Activar la variable de shell notify y ejecutar el comando `(sleep 5; echo "desperte")&`. Inmediatamente ejecutar el comando `ps -l` para ver la situación de los procesos. Elaborar un árbol jerárquico de procesos (incluyendo su PID respectivo) de lo desplegado por `ps -l`. ¿Hay algún proceso que parezca fuera de lugar? ¿Esta situación la provocan los paréntesis?

Se creó un nuevo proceso `sleep` con PID 2534 hijo de `tcsh`. Los paréntesis crearon un `subshell` para ejecutar en background el comando `sleep`, y pasado el tiempo desplegó en pantalla el texto “desperté”.

```

AO[51]>>set notify
AO[52]>> (sleep 5; echo "desperte")&
[1] 2533
AO[53]>>ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2282  2275  0  80   0 -  5833 wait  pts/4    00:00:00 bash
0 S  1000  2315  2282  0  80   0 -  5335 sigsus pts/4    00:00:00 tcsh
1 S  1000  2533  2315  0  80   0 -  5335 sigsus pts/4    00:00:00 tcsh
0 S  1000  2534  2533  0  80   0 -  1822 hrtime pts/4    00:00:00 sleep
0 R  1000  2535  2315  0  80   0 -  7229 -      pts/4    00:00:00 ps
AO[54]>>"desperte"
[1]      Done
AO[54]>>      ( sleep 5; echo "desperte" )

```

Fig. 24: Resultado del comando.

22. ¿Qué efecto tienen los siguientes comandos?

- `prompt> cat < $HOME/ua/f-rojas/Nequeteje > arch`: Dirige el archivo `Nequeteje` como entrada al comando `cat`, el cual redirige el contenido a un archivo `arch`.
- `prompt> cat $HOME/ua/f-rojas/Nequeteje > arch`: Envía el contenido del archivo `Nequeteje` a un nuevo archivo `arch`.
- `prompt> cat $HOME/ua/f-rojas/Nequeteje | tee arch > /dev / null`: Envía el contenido del archivo `Nequeteje` a un nuevo archivo `arch` y con el uso del comando `tee` copia el contenido dirigiéndolo a `dev/null`.

23. En el directorio `uc`, tratar de predecir el resultado que se obtendrá de los siguientes comandos:

- `prompt> date;who | wc -l`  
Se imprimirá la fecha actual y en la siguiente línea el número de líneas que devuelve `who`.

- `prompt> (date;who) | wc -l`  
Se imprimirá el número de líneas que devuelven los comandos `date` y `who`.
- `prompt> echo *`  
El resultado será similar a ejecutar `ls`.
- `prompt> echo '*'`  
Se imprimirá el símbolo `*`.
- `prompt> echo \*`  
Se imprimirá el símbolo `*`.
- `prompt> echo x*y`  
Se imprimirá toda variable que empiece con `x` y termine con `y`.
- `prompt> echo x'*y`  
Se imprimirá el texto `x*y`.

## Conclusiones

La presente práctica nos permitió comprender de manera más tangible el concepto de `shell`, ya que teóricamente no se había profundizado acerca de sus funciones y potencial como intérprete de comandos. Pudimos trabajar, activar y desactivar, crear, y visualizar el estado de las diferentes variables de `shell`. Observamos las diferencias de ejecutar comandos comunes dentro de la terminal y dentro de los diferentes tipos de `shell`. Además, conocimos el comando `history` que nos permite saber de los comandos que previamente se han ejecutado, así como las formas en que podemos modificar este registro de actividad.

## Referencias

- Ríos, J. (2019). Notas del curso de Sistemas Operativos. Recuperado el 19 de febrero de 2019, del sitio web: Comunidad ITAM.