Entregable numero dos

1. ¿Qué es un usuario root en Linux?

En sistemas operativos del tipo Unix, el superusuario o root es el nombre convencional de la cuenta de usuario que posee todos los derechos en todos los modos (monousuario o multiusuario). Normalmente es la cuenta de administrador. El usuario root puede hacer muchas cosas que un usuario común no puede, tales como cambiar el dueño o permisos de archivos y enlazar a puertos de numeración pequeña. No es recomendable utilizar el superusuario root para una simple sesión de uso habitual, ya que pone en riesgo el sistema al garantizar acceso privilegiado a cada programa en ejecución. Es preferible utilizar una cuenta de usuario normal y utilizar el comando su para acceder a los privilegios de Root en caso de ser necesario.

1. ¿Cuáles son los procesos típicos de Linux?

Los principales estados en los que pueden encontrarse los procesos en Linux/Unix son los siguientes:

* running (R) : Procesos que están en ejecución.
* sleeping (S) : Procesos que están esperando su turno para ejecutarse.
* stopped (D) : Procesos que esperan a que se finalice alguna operación de Entrada/Salida.
* zombie (Z) : Procesos que han terminado pero que siguen apareciendo en la tabla de procesos.

1. ¿Cómo identificarlos?

Top es un gestor de procesos integrado en la mayoría de sistemas Linux. Mientras que ps nos muestra un listado de procesos estático, es decir, nos informa de los procesos, nombres, usuarios o recursos que se están usando en el momento de la petición; top nos da un informe en tiempo real de los mismos.

1. Investigar y establecer una contraseña para el usuario root.

Permite la autenticación del usuario root con contraseña en su archivo sshd\_config :

* ~$ sudo sed -i 's/#PermitRootLogin prohibit-password/PermitRootLogin yes/g' /etc/ssh/sshd\_config
* ~$ sudo sed -i 's/PasswordAuthentication no/PasswordAuthentication yes/g' /etc/ssh/sshd\_config
* Reinicie el servicio SSH:
* ~$ service sshd restart

Una vez hecho esto, debería poder acceder al servidor con el usuario root y la contraseña establecida.

Para cambiar de clave

* Abre la Terminal (Control + Alt + T)
* Teclea (sin comillas) "sudo su"
* Introduce tu clave actual
* Teclea "passwd root" y escribe tu nueva clave
* Pulsa enter y cierra la terminal

Procesos de Linux Apoyo de compañeras del curso:

Un proceso es una instancia de un programa en ejecución (corriendo). El contexto de un programa que esta en ejecución es la esencia del proceso.

Este contexto implica que el proceso está definido por el programa, los semáforos, el espacio de memoria que ocupa y los procesos hijos que mande a llamar. Los hijos de un proceso comienzan siendo copias exactas del proceso padre, para así llamar al programa o tarea correspondiente. De esta manera el proceso principal (proceso padre), comparte con sus hijos los recursos del sistema que este consumiendo, sus atributos de seguridad (tales como su propietario y permisos de archivos así como roles y semaforización.

Linux es un sistema operativo multitarea y multiusuario, esto implica que múltiples procesos pueden operar simultáneamente sin interferirse unos con los otros. Cada proceso tiene la "ilusión" que es el único proceso en el sistema y que tiene acceso exclusivo a todos los servicios del sistema operativo.

sin embargo en un sentido más estricto, Programas y procesos son entidades distintas. En un sistema operativo multitarea, múltiples instancias de un programa pueden ejecutarse simultáneamente; si éste es el caso, cada instancia es un proceso separado. Por ejemplo, si cinco usuarios desde equipos diferentes, ejecutan el mismo programa al mismo tiempo, habría cinco instancias del mismo programa, es decir, cinco procesos distintos.

Cada proceso que se inicia es referenciado con su Process ID (PID), que es siempre un entero positivo entre 0 y 65,535 aunque esto puede modificarse. Prácticamente todo lo que se está ejecutando en el sistema en cualquier momento es un proceso, incluyendo el shell, el ambiente gráfico, el stack de protocolos de la red, etc. La excepción a lo anterior es el kernel, el cual es un conjunto de rutinas que residen en memoria y que entre otras responsabilidades administra el planificador que se encarga de controlar los procesos.

COMANDO **ps**

El comando ps devuelve el estado de los procesos. Su framework es el directorio /proc; lo que significa que lee directamente la información de archivos que se encuentran en este directorio. ps Tiene una gran cantidad de opciones, que incluso varían dependiendo del estilo en que se use el comando. Estas variaciones son:

• Estilo UNIX, donde las opciones van precedidas por un guión -

• Estilo BSD, donde las opciones no llevan guión

• Estilo GNU, donde se utilizan nombres de opciones largas y van precedidas por doble guión

Probemos en la terminal: si tecleamos el comando ps a secas, nos devuelve algo similar esto:

usuario@ubuntu: ~$ ps

PID TTY TIME CMD

2973 pts/0 00:00:00 bash

2993 pts/0 00:00:00 ps

Observa que tanto el shell Bash, como el propio comando ps son procesos, en este caso ambos corren en la terminal del escritorio, llevan un tiempo de ejecución 00:00:00 y su nombre respectivo.

**En las columnas que devuelve ps podemos encontrar:**

p o PID Process ID, número único o de identificación del proceso.

P o PPID Parent Process ID, padre del proceso

U o UID User ID, usuario propietario del proceso

t o TT o TTY Terminal asociada al proceso, si no hay terminal aparece entonces un '?'

T o TIME Tiempo de uso de cpu acumulado por el proceso

c o CMD El nombre del programa o comando que inició el proceso

RSS Resident Size, tamaño de la parte residente en memoria en kilobytes

SZ o SIZE Tamaño virtual de la imagen del proceso

NI Nice, valor nice (prioridad) del proceso, un número positivo significa menos tiempo de procesador y negativo más tiempo (-19 a 19)

C o PCPU Porcentaje de CPU utilizado por el proceso

**STIME Starting Time, hora de inicio del proceso**

S o STAT Status del proceso, estos pueden ser los siguientes

• R runnable, en ejecución, corriendo o ejecutándose

• S sleeping, proceso en ejecución pero sin actividad por el momento, o esperando por algún evento para continuar

• T sTopped, proceso detenido totalmente, pero puede ser reiniciado

• Z zombie, difunto, proceso que por alguna razón no terminó de manera correcta, no debe haber procesos zombies

• D uninterruptible sleep, son procesos generalmente asociados a acciones de IO del sistema

• X dead, muerto, proceso terminado pero que sigue apareciendo, igual que los Z no deberían verse nunca

Ahora probemos algunas opciones de ps: explica que hace cada comando.

#> ps -e

#> ps -ef (-f muestra opciones completas)

#>ps-eF(-F muestra opciones completas extra)

#>ps ax

#>ps aux

#>ps -eouser,pid,tty

**pstree**

Muestra los procesos en forma de árbol, pstree --help te da las opciones más comunes.

Es recomendable usarlo con la opción -A y -G para que te un árbol con líneas con líneas estilo ASCII y de terminal VT100 respectivamente, puedes añadir también -u para mostrar entre

paréntesis al usuario propietario del proceso:

#> pstree -AGu

**kill**

El comando kill, que literalmente quiere decir matar, sirve no solo para matar o terminar procesos sino principalmente para enviar señales (signals) a los procesos. La señal por default (cuando no se indica ninguna es terminar o matar el proceso), y la sintaxis es kill PID, siendo PID el número de ID del proceso. Asi por ejemplo, es posible enviar una señal de STOP al proceso y se detendrá su ejecución, después cuando se quiera mandar una señal de CONTinuar y el proceso continuara desde donde se quedo.

#> kill -l

La lista que devuelve el comando contiene todas las posibles señales que pueden mandarse a un proceso y estas pueden ser invocadas a través del número de la señal o de su código, por ejemplo:

#> kill -9 11428 (termina, mata un proceso completamente)

#> kill -SIGKILL 11428 (Lo mismo que lo anterior)

Las señales más comunes son la 19 y 20 que detienen momentáneamente la ejecución de un proceso o programa, 18 la continua, 1 que es la señal de hang up que obliga al proceso a releer sus archivos de configuración estando en ejecución y 9 que termina rotundamente un proceso.

**killall**

El comando killall, que funciona de manera similar a kill, pero con la diferencia de en vez de pasarle un PID se indica el nombre del programa, lo que afectará a todos los procesos que tengan ese nombre. Así por ejemplo si se tienen varias instancias ejecutándose del proxy server squid, con killall squid eliminará todos los procesos que se estén ejecutando con el nombre 'squid'

#> killall -l (lista de posibles señales)

#> killall -HUP httpd (manda una señal de "colgar", detenerse

releer sus archivos de configuración y

reiniciar)

#> killall -KILL -i squid (manda señal de matar a todos los

procesos squid pero pide confirmación)

**nice**

Permite cambiar la prioridad de un proceso. Por defecto, todos los procesos tienen una prioridad igual ante el CPU que es de 0. Con nice es posible iniciar un programa (proceso) con la prioridad modificada, más alta o más baja según se requiera. Las prioridades van de -20 (la más alta) a 19 la más baja. Solo root puede establecer prioridades negativas que son más altas. Con la opción ps -l es posible observar la columna NI que muestra este valor.

#> nice (sin argumentos, devuelve la prioridad por defecto )

#> nice -n -5 comando (inicia comando con una prioridad de -5, lo que le da más tiempo de cpu)

**renice**

Así como nice establece la prioridad de un proceso cuando se inicia su ejecución, renice permite alterarla en tiempo real, sin necesidad de detener el proceso.

#> nice -n -5 yes (se ejecuta el programa 'yes' con prioridad -5)

Dejemos ejecutando yes y en otra terminal consultemos ps

#> ps -el

Ahora cambiemos su prioridad:

#> renice 7 12826 (ojo:usar el id del proceso )

Vemos que la prioridad antigua era -5, y la nueva prioridad que pedimos es 7

Volvemos a checar la prioridad

#> ps -el

Observa el campo NI en el primer caso en -5, y en el segundo con renice quedó en 7, en tiempo real.

**nohup y &**

Cuando se trata ejecutar procesos en background (segundo plano) se utiliza el comando nohup o el operador & . Aunque realizan una función similar, no son lo mismo.

Si se desea liberar la terminal de un programa que se espera durará un tiempo considerable ejecutándose, entonces se usa . Esto funciona mejor cuando el resultado del proceso no es necesario mandarlo a la salida estándar (stdin), como por ejemplo cuando se ejecuta un respaldo o se abre un programa Xwindow desde la consola o terminal. Para lograr esto basta con escribir el comando en cuestión y agregar al final el símbolo &.

$> nohup yes > /dev/null &

$> nohup czf respaldo /documentos/\* > /dev/null/

$> nohup konqueror

Así se evita que el proceso se "cuelgue" al cerrar la consola.

Otra forma de usar nohup es para evitar que un programa que ejecutó el usuario no se corte o se termine abruptamente si este hace logout:

# nohup miprograma &

# exit

**jobs**

Si por ejemplo, se tiene acceso a una única consola o terminal, y se tienen que ejecutar varios comandos que

se ejecutarán por largo tiempo, se pueden entonces como ya se vió previamente con nohup y el operador '&'

mandarlos a segundo plano o background con el objeto de liberar la terminal y continuar trabajando.

Pero si solo se está en una terminal esto puede ser difícil de controlar, y para esos casos tenemos el comando

jobs que lista los procesos actuales en ejecución:

#> yes > /dev/null &

[1] 26837

#> ls -laR > archivos.txt &

[2] 26854

#> jobs

[1]- Running yes >/dev/null &

[2]+ Running ls --color=tty -laR / >archivos.txt &

En el ejemplo previo, se ejecutó el comando yes y se envió a background (&) y el sistema devolvió [1] 26837, indicando asi que se trata del trabajo o de la tarea [1] y su PID, lo mismo con la segunda tarea que es un listado recursivo desde la raíz y enviado a un archivo, ésta es la segunda tarea.

Con los comandos fg (foreground) y bg background es posible manipular procesos que estén suspendidos temporalmente, ya sea porque se les envió una señal de suspensión como STOP (20) o porque al estarlos ejecutando se presionó ctrl-Z. Entonces para reanudar su ejecución en primer plano usaríamos fg:

#> jobs

[1]- Stopped yes >/dev/null &

[2]+ Stopped ls --color=tty -laR / >archivos.txt &

#> fg %1

#> jobs

[1]+ Running yes >/dev/null &

[2]- Stopped ls --color=tty -laR / >archivos.txt &

|  |
| --- |
| **top**  Es una utilería muy útil para el monitoreo en tiempo real del estado de los procesos y de otras variantes del sistema; se ejecuta desde la línea de comandos, es interactivo y por defecto se actualiza cada 3 segundos.  $> top  top - 13:07:30 up 8 days,  6:44,  4 users,  load average: 0.11, 0.08, 0.08  Tasks: 133 total,   1 running, 131 sleeping,   0 stopped,   1 zombie  Cpu(s):  0.0%us,  0.2%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.2%si,  0.0%st  Mem:    497356k total,   472352k used,    25004k free,    21500k buffers  Swap:  1156640k total,   257088k used,   899552k free,    60420k cached    PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND  26156 sergon    15   0  2160 1016  784 R    1  0.2   0:00.93 top      1 root      15   0  2012  616  584 S    0  0.1   0:00.98 init      2 root      RT   0     0    0    0 S    0  0.0   0:00.29 migration/0      3 root      34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/0      4 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/0      5 root      RT   0     0    0    0 S    0  0.0   0:00.38 migration/1  Estando adentro de la aplicación, presionando 'h' muestra una ayuda de los posibles comandos que permiten configurar **top**, por ejemplo, al presionar 's' pregunta por el tiempo en segundos de actualización, etc.  **fork**  Es un mecanismo de control de procesos e hilos. Dos conceptos relacionados, pero con sutiles diferencias.  Partiendo de la idea posible de "**ejecutar varias cosas a la vez**” surgen los hilos o threads, pero su aspecto mas relevante es la ejecución fuera del contexto de una terminal o una ventana, evitando en todo momento el cuello de botella con la intefaz de usuario y liberando éstos recursos para la aplicación en primer plano.  Si queremos que nuestro programa empiece a ejecutar varias cosas "a la vez", tenemos dos opciones.  Podemos crear un nuevo proceso o crear un nuevo hilo de ejecución (un thread).  **Procesos y su relación con fork**  En un sistema Linux, (multihilo), se pueden estar ejecutando distintas acciones a la par, y cada acción es un  proceso que consta de uno o más hilos, memoria de trabajo compartida por todos los hilos e información de  planificación. Cada hilo consta de instrucciones y estado de ejecución.  Cuando ejecutamos un comando en el shell, sus instrucciones se copian en algún sitio de la memoria RAM del  sistema para ser ejecutadas. Cuando las nstrucciones ya cumplieron su cometido, el programa es borrado de la  memoria del sistema, dejándola libre para que más programas se puedan ejecutar a la vez. Por tanto cada uno de  estos programas son los procesos.  Los procesos son creados y destruidos por el sistema operativo, pero lo hace a petición de otros procesos. El  mecanismo por el cual un proceso crea otro proceso se denomina bifurcación (fork). Los nuevos procesos son  independientes y no comparten memoria (es decir, información) con el proceso que los ha creado.  En definitiva, es posible crear tanto hilos como procesos. La diferencia estriba en que un proceso solamente puede  crear hilos para sí mismo y en que dichos hilos comparten toda la memoria reservada para el proceso.  **Hilos**  Los hilos son similares a los procesos ya que ambos representan una secuencia simple de instrucciones  ejecutada en paralelo con otras secuencias. Los hilos son una forma de dividir un programa en dos o más  tareas que corren simultáneamente, compitiendo, en algunos casos, por la CPU.  La diferencia más significativa entre los procesos y los hilos, es que los primeros son típicamente independientes,  llevan bastante información de estados, e interactúan sólo a través de mecanismos de comunicación dados por el sistema. Por otra parte, los hilos generalmente comparten la memoria, es decir, acceden a las mismas variables  globales o dinámicas, por lo que no necesitan costosos mecanismos de comunicación para sincronizarse. Por  ejemplo un hilo podría encarguese de la interfaz gráfica (iconos, botones, ventanas), mientras que otro hace una  larga operación internamente. De esta manera el programa responde más ágilmente a la interacción con el  usuario.  En sistemas operativos que proveen facilidades para los hilos, es más rápido cambiar de un hilo a otro dentro del  mismo proceso, que cambiar de un proceso a otro.  Es posible que los hilos requieran de operaciones atómicas para impedir que los datos comunes sean cambiados  o leídos mientras estén siendo modificados. El descuido de esto puede generar estancamiento.  La tabla siguiente resume algunas diferencias entre procesos e hilos, y la figura 1 muestra una representación de  los conceptos proceso e hilo. Nótese cómo en la figura 1 se puede apreciar que los procesos son entidades  independientes, mientras que los hilos son entidades relacionadas por la sección de datos en el interior del  proceso que los contiene.  [RELACION PROCESOS-HILOS](https://sites.google.com/site/mrtripus/home/procesos-en-linux/comparativo_procesos_hilos.jpg?attredirects=0)  [Diagrama  Descripción generada automáticamente](https://sites.google.com/site/mrtripus/home/procesos-en-linux/relacion_procesos_hilos_arquitext.jpg?attredirects=0)  **Creación de procesos: fork y clone**  A la hora de crear procesos linux provee de dos funciones para dicho cometido, la función clone() y la función  fork(). Ambas crean un nuevo proceso a partir del proceso padre pero de una manera distinta.  Cuando utilizamos la llamada al sistema fork, el proceso hijo creado es una copia exacta del padre (salvo por el  PID y la memoria que ocupa). Al proceso hijo se le facilita una copia de las variables del proceso padre y de  los descriptores de archivo. Es importante destacar que las variables del proceso hijo son una copia de las del  padre (no se refieren físicamente a la misma variable), por lo que modificar una variable en uno de los procesos  no se refleja en el otro.  La llamada al sistema clone es mucho más genérica y flexible que el fork, ya que nos permite definir qué van a  compartir los procesos padre e hijo. La tabla 2 resume las diferencias entre las llamadas al sistema fork y clone.  Las llamadas al sistema fork y clone tienen la misma funcionalidad, pero distintas características:  **fork:**  En el momento de la llamada a fork el proceso hijo:  • es una copia exacta del padre excepto el PID.  • tiene las mismas variables y archivos abiertos.  • las variables son independientes (padre e hijo tienen distintasmemorias).  • los archivos son compartidos (heredan el descriptor).  **clone:**  Permite especificar qué queremos que compartan padre e hijo.  • espacio de direccionamiento  • información de control del sistema de archivos (file system)  • descriptores de archivos abiertos.  • gestores de señales o PID. |