Juan Menduiña

Trabajo Práctico N° 1:
GNU/Linux, Instalación y Conceptos Básicos, Permisos, Arranque, Usuarios. Organización Interna.

Ejercicio 1: Características de GNU/Linux.

(a) *Mencionar y explicar las características más relevantes de GNU/Linux.*

GNU/Linux es un sistema operativo libre y de código abierto, basado en el núcleo Linux y en las herramientas del proyecto GNU. Sus características más relevantes son:

1. ***Software* libre y código abierto:**
   GNU/Linux se distribuye bajo licencias libres (como la GPL), lo que permite a los usuarios usar, estudiar, modificar y redistribuir el *software*. Esta libertad fomenta la colaboración y la mejora continua del sistema.
2. **Multitarea y multiusuario:**
   Permite que varios usuarios trabajen, simultáneamente, en el mismo sistema sin interferir entre sí y que múltiples procesos se ejecuten al mismo tiempo. Esto lo hace ideal para entornos de servidores y redes.
3. **Portabilidad:**
   GNU/Linux puede ejecutarse en una amplia variedad de plataformas y arquitecturas (x86, ARM, RISC-V, etc.), desde servidores y computadoras personales hasta dispositivos embebidos y teléfonos.
4. **Seguridad y estabilidad:**
   Es reconocido por su gran estabilidad, incluso en ejecuciones prolongadas, y por su robusto sistema de permisos y usuarios, que limita el alcance de posibles ataques o errores. Además, la comunidad realiza actualizaciones y parches de seguridad constantemente.
5. **Estructura modular:**
   Está compuesto por módulos independientes (núcleo, *shell*, sistema de archivos, utilidades, etc.), lo que facilita su mantenimiento y personalización.
6. **Sistema de archivos jerárquico:**
   Organiza todos los recursos (archivos, dispositivos, configuraciones) en una única estructura jerárquica, que parte del directorio raíz (*/*).
7. **Compatibilidad y flexibilidad:**
   Existen numerosas distribuciones (Debian, Ubuntu, Fedora, Arch, etc.) adaptadas a distintos tipos de usuarios y necesidades. Además, permite elegir entre múltiples entornos de escritorio, gestores de paquetes y herramientas.
8. **Comunidad y soporte colaborativo:**
   GNU/Linux cuenta con una gran comunidad internacional que desarrolla, documenta y brinda soporte de manera colaborativa.

(b) *Mencionar otros sistemas operativos y compararlos con GNU/Linux en cuanto a los puntos mencionados en el inciso (a).*

Otros sistemas operativos, ampliamente utilizados, son Windows (Microsoft), masOS (Apple) y Android (Google). A continuación, se comparan con GNU/Linux en cuanto a los puntos mencionados en el inciso (a):

1. ***Software* libre y código abierto:**
   - Windows: Es propietario y cerrado; el código fuente no está disponible y su uso está sujeto a licencias comerciales.
   - macOS: Es privativo y exclusivo del *hardware* de Apple, aunque incorpora componentes de código abierto (como partes de BSD).
   - Android: Parcialmente abierto, ya que el núcleo es Linux, pero muchas capas y servicios de Google son propietarios.
2. **Multitarea y multiusuario:**
   Todos los sistemas mencionados permiten multitarea (ejecución simultánea de procesos). Sin embargo, GNU/Linux fue diseñado desde sus orígenes como multiusuario real, algo que, en Windows y macOS, se implementó más tarde y con mayores restricciones.
3. **Portabilidad:**
   - Windows: Está optimizado para arquitecturas x86/x64; su portabilidad es limitada.
   - macOS: Funciona, exclusivamente, en equipos Apple.
   - Android: Muy extendido en dispositivos móviles, pero no tan flexible fuera de ese entorno.
4. **Seguridad y estabilidad:**
   - Windows: Más vulnerable a virus y *malware*, principalmente por su popularidad y arquitectura de permisos más laxa.
   - *macOS:* Tiene buena seguridad, aunque no alcanza la flexibilidad ni el control de GNU/Linux.
   - Android: Depende de las capas del fabricante y de la actualización; puede presentar vulnerabilidades si no se actualiza.
5. **Estructura modular:**
   - GNU/Linux: Altamente modular; el usuario puede modificar o reemplazar componentes del sistema.
   - Windows y macOS: Mucho más cerrados; el usuario tiene escaso control sobre los módulos del sistema.
   - Android: Parcialmente modular, pero condicionado por Google y los fabricantes.
6. **Sistema de archivos jerárquico:**
   Todos utilizan una estructura jerárquica, aunque en GNU/Linux es más uniforme (todo se organiza bajo "/"). En Windows, existen múltiples unidades (C:, D:, etc.), lo que fragmenta la estructura.
7. **Compatibilidad y flexibilidad:**
   - Windows: Ampliamente compatible con programas comerciales y *hardware*, pero poco flexible; el usuario no puede modificar ni personalizar, en profundidad, el sistema.
   - macOS: Ofrece buena compatibilidad dentro del ecosistema Apple y gran estabilidad, pero es cerrado y limitado al *hardware* de la marca.
   - Android: Posee gran compatibilidad con aplicaciones móviles, pero su flexibilidad depende del fabricante y de las capas de *software* agregadas, que restringen la personalización completa.

8. **Comunidad y soporte:**
   - Windows y macOS: El soporte depende de las empresas propietarias.
   - Android: Tiene soporte de Google y comunidades específicas, pero menos abiertas que en GNU/Linux.

En síntesis, GNU/Linux se destaca por su libertad, seguridad, estabilidad y flexibilidad, mientras que los sistemas operativos propietarios como Windows y macOS ofrecen mayor facilidad de uso y compatibilidad comercial, pero a costa de menor control por parte del usuario.

(c) *¿Qué es GNU?*

GNU es un proyecto de *software* libre iniciado en 1983 por Richard Stallman, con el objetivo de crear un sistema operativo completamente libre, compatible con Unix, pero sin incluir ningún componente privativo. El nombre GNU es un acrónimo recursivo que significa "GNU's Not Unix" ("GNU No es Unix").

El proyecto forma parte del movimiento del *software* libre, que defiende las cuatro libertades fundamentales del usuario:

- Usar el programa con cualquier propósito.
- Estudiar cómo funciona y adaptarlo a las necesidades propias.
- Redistribuir copias.
- Mejorar el programa y publicar esas mejoras.

El sistema GNU proporciona las herramientas básicas de un sistema operativo (compiladores, bibliotecas, *shells*, utilidades de administración, editores de texto, etc.).

En la actualidad, la mayoría de las distribuciones conocidas como "Linux" son, en realidad, combinaciones del núcleo Linux con las herramientas del proyecto GNU, motivo por el cual su nombre correcto es GNU/Linux.

(d) *Indicar una breve historia sobre la evolución del proyecto GNU.*

El proyecto GNU fue iniciado en 1983 por Richard Stallman en el Instituto Tecnológico de Massachusetts (MIT), con la idea de desarrollar un sistema operativo completamente libre y compatible con Unix.

En 1985, Stallman fundó la *Free Software Foundation* (FSF) para apoyar el desarrollo y la difusión del *software* libre, así como para promover licencias que garantizaran la libertad de los usuarios, como la *General Public Licence* de GNU (GPL).

Durante la segunda mitad de la década de 1980, el proyecto GNU avanzó en la creación de herramientas esenciales como el compilador GCC (GNU *Compiler Collection*), el editor Emacs, el intérprete de comandos Bash y muchas utilidades del sistema. Sin

embargo, el sistema GNU aún no contaba con un núcleo funcional (llamado Hurd), que se encontraba en desarrollo.

En 1991, Linus Torvalds publicó el núcleo Linux, que era libre y compatible con las herramientas GNU. La combinación de ambos permitió conformar un sistema operativo completo: GNU/Linux, el cual se difundió, rápidamente, en universidades, empresas y entornos domésticos.

Desde entonces, el proyecto GNU continúa activo, manteniendo y desarrollando múltiples programas libres, y promoviendo los principios éticos y sociales del movimiento del *software* libre.

(e) *Explicar qué es la multitarea e indicar si GNU/Linux hace uso de ella.*

La multitarea es la capacidad de un sistema operativo para ejecutar varios procesos o programas de manera simultánea, compartiendo los recursos del procesador, la memoria y otros dispositivos.

En realidad, el procesador alterna, rápidamente, entre las distintas tareas, dando la sensación de ejecución paralela (especialmente, en sistemas con un solo núcleo). En los procesadores multinúcleo, varias tareas pueden ejecutarse, verdaderamente, al mismo tiempo.

Existen dos tipos principales de multitarea:

- Cooperativa: Cada proceso cede, voluntariamente, el control al sistema operativo.
- Preventiva: El sistema operativo decide cuándo interrumpir un proceso para dar tiempo de CPU a otro.

GNU/Linux implementa multitarea preventiva, lo que significa que el núcleo administra, de forma automática, el uso del procesador entre los distintos procesos. Esto permite que el sistema siga funcionando de manera fluida, incluso cuando muchos programas están activos al mismo tiempo.

(f) *¿Qué es POSIX?*

POSIX significa *Portable Operating System Interface* (Interfaz Portátil para Sistemas Operativos). Es un conjunto de estándares definidos por IEEE (Instituto de Ingenieros Eléctricos y Electrónicos) que especifica una interfaz común para los sistemas operativos tipo Unix.

El objetivo de POSIX es garantizar que los programas puedan compilarse y ejecutarse en distintos sistemas operativos compatibles, sin necesidad de modificar su código fuente. Para lograrlo, define normas sobre aspectos como:

- Llamadas al sistema (*system calls*).

- Estructura del sistema de archivos.
- Manejo de procesos e hilos.
- Señales y comunicación entre procesos.
- Comportamiento del *shell* y utilidades básicas.

GNU/Linux cumple, en gran medida, con el estándar POSIX, lo que permite que muchas aplicaciones desarrolladas para sistemas Unix (como BSD o macOS) puedan ejecutarse en Linux con muy pocas modificaciones.

En resumen, POSIX promueve la portabilidad y la compatibilidad entre los distintos sistemas operativos de la familia Unix y sus derivados.

Ejercicio 2: Distribuciones de GNU/Linux.

(a) *¿Qué es una distribución de GNU/Linux? Nombrar, al menos, 4 distribuciones de GNU/Linux y citar diferencias básicas entre ellas.*

Una distribución de GNU/Linux es un paquete completo de *software* que combina el núcleo Linux, las herramientas del proyecto GNU y otros programas adicionales (como gestores de paquetes, entornos de escritorio y aplicaciones) para ofrecer un sistema operativo listo para instalar y usar. Cada distribución se adapta a distintos propósitos, como servidores, escritorios, educación o seguridad informática.

Algunas distribuciones populares son:

1. **Debian:**
   - Enfocada en estabilidad y robustez.
   - Ideal para servidores y entornos críticos.
   - Sus paquetes son más conservadores; menos actualizaciones frecuentes.
2. **Ubuntu:**
   - Orientada a usuarios de escritorio y principiantes.
   - Fácil instalación y amplia comunidad de soporte.
   - Basada en Debian, con un ciclo de actualizaciones regular y soporte a largo plazo (LTS).
3. **Fedora:**
   - Distribución vanguardista, incluye *software* más reciente.
   - Buena para desarrolladores y pruebas de nuevas tecnologías.
   - Respaldo de *Red Hat*, aunque con un ciclo de vida más corto.
4. **Arch Linux:**
   - Minimalista y, altamente, personalizable.
   - Filosofía "*rolling release*" (actualizaciones continuas).
   - Requiere mayor conocimiento técnico para su instalación y mantenimiento.

En resumen, cada distribución combina los mismos elementos básicos de GNU/Linux, pero se diferencian en facilidad de uso, frecuencia de actualizaciones, estabilidad y personalización.

(b) *¿En qué se diferencia una distribución de otra?*

Aunque todas las distribuciones de GNU/Linux comparten el núcleo Linux y las herramientas GNU, se diferencian, principalmente, en los siguientes aspectos:

1. **Gestor de paquetes:**
   Cada distribución utiliza su propio sistema para instalar y actualizar *software*:
   - Debian/Ubuntu usan APT y paquetes .deb.
   - Fedora/Red Hat usan DNF/YUM y paquetes .rpm.
   - Arch Linux usa Pacman.
2. **Ciclo de actualizaciones y estabilidad:**

- Algunas distribuciones (Debian Stable, Ubuntu LTS) priorizan estabilidad, con versiones de *software* más conservadoras.
- Otras (Fedora, Arch Linux) priorizan lo último en *software*, con actualizaciones frecuentes.

3. **Facilidad de uso e instalación:**
   - Distribuciones como Ubuntu están orientadas a usuarios principiantes, con instaladores gráficos y entornos de escritorio listos para usar.
   - Arch Linux o Gentoo requieren conocimientos avanzados, con instalación y configuración manual.
4. **Entorno de escritorio y personalización:**
   - Algunas distribuciones vienen con entornos gráficos predeterminados (GNOME, KDE, XFCE).
   - Otras permiten al usuario elegir o instalar el entorno que prefiera.
5. **Propósito y enfoque:**
   - Algunas distribuciones se enfocan en servidores (Debian, CentOS), otras en escritorio general (Ubuntu, Linux Mint) y otras en seguridad o pruebas de penetración (Kali Linux, Parrot OS).

En resumen, las diferencias entre distribuciones se centran en gestión de paquetes, frecuencia de actualizaciones, facilidad de uso, entornos gráficos y finalidad del sistema, mientras que el núcleo Linux y las herramientas GNU siguen siendo comunes a todas.

(c) *¿Qué es Debian? Acceder al sitio https://www.debian.org/ e indicar cuáles son los objetivos del proyecto y una breve cronología del mismo.*

Debian es un sistema operativo libre y universal, compuesto por el núcleo Linux y herramientas del proyecto GNU. Es mantenido por una comunidad internacional de desarrolladores y usuarios comprometidos con el *software* libre. Su nombre proviene de la combinación de los nombres de sus creadores: Debra Lynn y Ian Murdock.

Objetivos del proyecto: Según su filosofía, Debian busca crear un sistema operativo libre, disponible para todo el mundo, sin importar su ubicación geográfica, idioma o nivel de conocimiento técnico. El proyecto se basa en principios de apertura, transparencia y colaboración comunitaria. No se enfoca en el beneficio económico, sino en el desarrollo ético y técnico del *software* libre.

Cronología del proyecto Debian:

- Agosto de 1993: Ian Murdock, estudiante de la Universidad de Purdue, inicia el proyecto Debian con el objetivo de crear una distribución de Linux completamente libre.
- 1994: Se publica la versión 0.91, que incluye un sistema de empaquetado básico y una estructura inicial de paquetes.
- 1996: Debian 1.1, conocida como “Buzz”, es lanzada, marcando la primera versión oficial del sistema.
- 1999: Se inicia el soporte para arquitecturas adicionales y se publica la versión 2.0, conocida como “Hamm”.

- 2005: Debian 3.1, conocida como “Sarge”, es lanzada después de un largo período de desarrollo.
- 2015: Se implementa el soporte a largo plazo (LTS) para versiones antiguas, extendiendo su vida útil y seguridad.
- 2025: Debian 13, conocida como “Trixie”, es lanzada, continuando con el compromiso de ofrecer un sistema operativo libre, estable y seguro.

Esta cronología destaca los hitos importantes en el desarrollo y evolución del proyecto Debian, reflejando su crecimiento y consolidación como una de las distribuciones más respetadas y utilizadas en el mundo del *software* libre.

Ejercicio 3: Estructura de GNU/Linux.

(a) *Nombrar cuáles son los 3 componentes fundamentales de GNU/Linux.*

Los tres componentes fundamentales de GNU/Linux son:

1. **Núcleo (*Kernel*):**
   - Es el corazón del sistema operativo.
   - Se encarga de gestionar el *hardware*, la memoria, los procesos y la comunicación entre dispositivos y programas.
   - Ejemplos: Linux kernel 6.x, 5.x, etc.
2. ***Shell* o intérprete de comandos:**
   - Es la interfaz entre el usuario y el núcleo.
   - Permite ejecutar comandos, *scripts* y programas.
   - Ejemplos: Bash, Zsh, Fish.
3. **Sistema de archivos y utilidades GNU:**
   - Conjunto de herramientas y programas esenciales para operar el sistema, como compiladores, editores de texto, utilidades de gestión de archivos y bibliotecas.
   - Incluye la estructura jerárquica de directorios, desde la raíz (/) hasta las carpetas de configuración y datos de usuario.

(b) *Mencionar y explicar la estructura básica del Sistema Operativo GNU/Linux.*

El sistema operativo GNU/Linux se organiza en varias capas jerárquicas, que permiten separar las funciones y facilitan la gestión del sistema. La estructura básica es la siguiente:

1. **Núcleo (*Kernel*):**
   - Es la capa más interna, responsable de la comunicación entre el *hardware* y el *software*.
   - Gestiona la CPU, memoria, dispositivos de entrada/salida, procesos y control de acceso.
   - Funciona como intermediario entre los programas y los recursos físicos del sistema.
2. ***Shell* o intérprete de comandos:**
   - Es la capa intermedia que permite al usuario interactuar con el sistema.
   - Puede ser texto (CLI - *Command Line Interface*) o gráfica (GUI), dependiendo de la configuración.
   - A través del *shell*, se pueden ejecutar comandos, *scripts* y programas, controlar procesos y administrar archivos.
3. **Sistema de archivos y utilidades:**
   - Incluye las herramientas básicas de GNU y otros programas esenciales (compiladores, editores, utilidades de red, bibliotecas, etc.)
   - Organiza los archivos en una estructura jerárquica que comienza en la raíz (*/*) y se ramifica en directorios como:
     - */bin*: programas esenciales.

- */etc*: archivos de configuración.
- */home*: directorios de usuarios.
- */usr*: programas y utilidades adicionales.
- */var*: archivos variables (logs, bases de datos).

4. **Aplicaciones y entorno de escritorio:**
   - Son los programas que el usuario final utiliza, como navegadores, procesadores de texto, reproductores de multimedia o entornos gráficos (GNOME, KDE, XFCE).
   - Esta capa depende de las anteriores para funcionar, ya que utiliza los servicios del *kernel*, el *shell* y las bibliotecas del sistema.

En resumen, GNU/Linux tiene una estructura modular y jerárquica, *Kernel* ⟶ *Shell* ⟶ Sistema de archivos/utilidades ⟶ Aplicaciones, lo que permite seguridad, estabilidad y flexibilidad en la gestión del sistema.

Ejercicio 4: *Kernel*.

(a) *¿Cuáles son sus funciones principales?*

El *kernel* es el núcleo del sistema operativo y su función principal es actuar como intermediario entre el *hardware* y el *software*. Sus funciones más importantes son:

1. **Gestión de procesos:**
   - Controla la creación, la planificación y la terminación de los procesos en ejecución.
   - Implementa la multitarea y asigna tiempo de CPU a cada proceso de manera eficiente.
2. **Gestión de memoria:**
   - Administra la memoria principal (RAM) y el espacio de intercambio (*swap*).
   - Se encarga de asignar memoria a los procesos y de proteger áreas de memoria para que un proceso no interfiera con otro.
3. **Gestión de dispositivos:**
   - Controla todos los dispositivos de *hardware* mediante los *drivers*.
   - Permite que los programas accedan a los recursos de forma estandarizada y segura.
4. **Gestión del sistema de archivos:**
   - Maneja la lectura, la escritura y la organización de los datos en los discos y otros medios de almacenamiento.
   - Garantiza la integridad de la información y permite acceso jerárquico a los archivos.
5. **Gestión de la seguridad y permisos:**
   - Controla los permisos de usuario y el acceso a recursos, evitando que procesos no autorizados realicen acciones críticas.
   - Implementa mecanismos de protección de memoria y de ejecución de código.
6. **Comunicación entre procesos:**
   - Facilita la interacción y la sincronización entre procesos mediante señales, tuberías, *sockets* y memoria compartida.

En resumen, el *kernel* es el cerebro del sistema, responsable de que los procesos, la memoria, el *hardware* y los archivos funcionen de manera coordinada y segura, garantizando la estabilidad y la eficiencia del sistema operativo.

(b) *¿Cuál es la versión actual? ¿Cómo se definía el esquema de versionado del Kernel en versiones anteriores a la 2.4? ¿Qué cambió en el versionado que se impuso a partir de la versión 2.6?*

Versión actual del kernel: Hasta el 16 de octubre de 2025, la versión estable más reciente del *kernel* de Linux es la 6.17.3. Esta versión se lanzó el 15 de octubre de 2025 y es la que se utiliza en distribuciones como Ubuntu 25.10 y Fedora 43.

Esquema de versionado antes de la versión 2.4: Antes de la versión 2.4, el esquema de versionado del *kernel* de Linux seguía una convención basada en números impares y pares:

- Números impares (por ejemplo, 2.1, 2.3, 2.5): Indicaban versiones en desarrollo, con cambios experimentales y no estables.
- Números pares (por ejemplo, 2.0, 2.2): Representaban versiones estables, listas para su uso general.

Este enfoque permitía a los desarrolladores y usuarios identificar, rápidamente, si una versión era estable o aún estaba en desarrollo.

Cambios en el esquema de versionado a partir de la versión 2.6: Con la introducción de la versión 2.6, el esquema de versionado cambió para reflejar mejor el ciclo de desarrollo y mantenimiento del *kernel*:

- Número mayor (2): Indicaba la serie principal del kernel.
- Número menor (6): Representaba la versión de desarrollo.
- Número de revisión (x): Denotaba actualizaciones menores o correcciones de errores.
- Número de parche (y): Se añadía para indicar parches específicos o actualizaciones de seguridad.

Este nuevo esquema proporcionaba una estructura más clara y coherente para el seguimiento de las versiones del *kernel* y facilitaba la gestión de actualizaciones y mantenimiento.

(c) *¿Es posible tener más de un Kernel de GNU/Linux instalado en la misma máquina?*

Sí, es posible tener más de un *kernel* de GNU/Linux instalado en la misma máquina. Esto se logra porque cada versión del *kernel* se instala en su propio directorio dentro de /boot y se registra en el GRUB (el gestor de arranque), que permite seleccionar qué *kernel* iniciar al arrancar el sistema.

Ventajas de tener múltiples *kernels*:

- Seguridad ante fallos: Si la versión más reciente del *kernel* causa problemas, se puede iniciar el sistema con una versión anterior que funcione correctamente.
- Compatibilidad: Permite probar nuevas versiones del *kernel* sin comprometer la estabilidad del sistema principal.
- Flexibilidad para desarrolladores: Facilita el desarrollo y las pruebas de módulos o *drivers* específicos para distintas versiones del *kernel*.

En resumen, tener más de un *kernel* instalado es común y seguro, y permite mantener la estabilidad del sistema mientras se prueban actualizaciones o nuevas características.

(d) *¿Dónde se encuentra ubicado dentro del File System?*

En GNU/Linux, el *kernel* no es un programa como cualquier otro, es un archivo especial que se carga en memoria al arrancar el sistema. Dentro del *File System*, se encuentra ubicado en el directorio */boot*, que contiene los archivos del *kernel* instalados en el sistema.

Ejercicio 5: Intérprete de comandos (*Shell*).

(a) *¿Qué es?*

El *shell* es un programa que actúa como intermediario entre el usuario y el núcleo (*kernel*) del sistema operativo. Su función principal es interpretar los comandos que ingresa el usuario y comunicarlos al *kernel* para que se ejecuten.

El *shell* es la interfaz de línea de comandos que permite a los usuarios interactuar con GNU/Linux de manera flexible y poderosa, tanto de forma interactiva como mediante *scripts*.

(b) *¿Cuáles son sus funciones?*

Sus funciones son:

- Interpretar comandos: Permite ejecutar programas, *scripts* o instrucciones del sistema.
- Automatización de tareas: A través de *scripts*, se pueden automatizar procesos repetitivos.
- Gestión de procesos: Permite iniciar, detener o supervisar procesos.
- Interacción con el sistema de archivos: Navegar por directorios, copiar, mover o eliminar archivos.
- Variables y programación básica: Permite usar variables, condicionales y bucles para tareas más complejas.

(c) *Mencionar, al menos, 3 intérpretes de comandos que posee GNU/Linux y compararlos entre ellos.*

GNU/Linux ofrece varios intérpretes de comandos (*shells*), cada uno con características particulares. A continuación, se mencionan 3 de los más utilizados y una breve comparación entre ellos:

1. **Bash (*Bourne Again Shell*):**
   - Es el *shell* por defecto en la mayoría de las distribuciones GNU/Linux.
   - Ventajas: compatible con *scripts* del *shell* original (sh); soporta historial de comandos, autocompletado y variables de entorno; muy estable y, ampliamente, documentado.
   - Uso típico: Ideal para administración de sistemas y *scripting* estándar.
2. **Zsh (*Z Shell*):**
   - Es una versión más avanzada y configurable que Bash.
   - Ventajas: ofrece autocompletado inteligente, sugerencias en tiempo real y temas visuales (como Oh My Zsh); permite una mayor personalización del *prompt* y funciones avanzadas de historial.

- Desventajas: puede consumir más recursos; requiere configuración inicial para aprovechar sus ventajas.
- Uso típico: Preferido por usuarios avanzados y desarrolladores.

3. **Fish (*Friendly Interactive Shell*):**
   - Diseñado para ser fácil de usar y, visualmente, más amigable.
   - Ventajas: autocompletado automático sin configuración adicional; coloreado de sintaxis por defecto; no requiere editar archivos de configuración complejos.
   - Desventajas: no es totalmente compatible con *scripts* de Bash, lo que limita su uso en entornos de producción.
   - Uso típico: Ideal para usuarios nuevos o tareas interactivas.

En resumen:

- Bash: clásico, estable y, ampliamente, usado.
- Zsh: más potente y configurable.
- Fish: moderno y simple para principiantes.

(d) *¿Dónde se ubican (path) los comandos propios y externos al Shell?*

En GNU/Linux, los comandos que se ejecutan en el *shell* pueden ser de dos tipos: internos (propios del *shell*) o externos (programas ejecutables del sistema). Cada tipo se encuentra en lugares distintos del sistema de archivos.

1. **Comandos internos (propios del *shell*):**
   - Son instrucciones integradas dentro del propio intérprete (por ejemplo, *cd*, *echo*, *pwd*, *export*, *history*).
   - No son archivos ejecutables en el sistema, sino funciones incorporadas en el shell.
   - Ubicación:
     - Están dentro del intérprete de comandos (por ejemplo, dentro de */bin/bash* o */usr/bin/zsh*).
     - No tienen una ruta propia en el sistema de archivos.
2. **Comandos externos:**
   - Son programas ejecutables almacenados en el sistema de archivos.
   - Cuando el usuario los invoca, el *shell* los busca en los directorios definidos en la variable de entorno $PATH.
   - Ubicaciones comunes:
     - */bin* ⟶ comandos básicos (por ejemplo, *ls*, *cp*, *mv*, *cat*).
     - */usr/bin* ⟶ aplicaciones de usuario (por ejemplo, *grep*, *nano*, *tar*).
     - */sbin* ⟶ comandos administrativos del sistema (por ejemplo, *reboot*, *ifconfig*).
     - */usr/sbin* ⟶ herramientas avanzadas para administración del sistema.

En resumen, los comandos internos están incorporados en el *shell*, mientras que los externos son programas independientes ubicados en los directorios del PATH del sistema.

(e) *¿Por qué se considera que el Shell no es parte del Kernel de GNU/Linux?*

Se considera que el Shell no es parte del Kernel de GNU/Linux porque:

- No gestiona hardware ni recursos del sistema.
- Opera en modo usuario, no en modo núcleo.
- Puede ser reemplazado sin alterar el sistema operativo.
- Actúa como interfaz entre el usuario y el Kernel, no como parte de él.

(f) *¿Es posible definir un intérprete de comandos distinto para cada usuario? ¿Desde dónde se define? ¿Cualquier usuario puede realizar dicha tarea?*

Sí, es posible definir un intérprete de comando distinto para cada usuario. En GNU/Linux, cada usuario puede tener asignado un *shell* diferente como su intérprete de comandos predeterminado. Esto significa que un usuario puede usar, por ejemplo, Bash, otro Zsh y otro Fish, según sus preferencias.

El *shell* predeterminado de cada usuario se define en el archivo */etc/passwd*.

Cada usuario puede cambiar su propio *shell*, siempre que:

- El nuevo intérprete esté instalado en el sistema.
- Esté listado en el archivo */etc/shells*, que contiene los *shells* válidos.

Sólo el usuario *root* (administrador) puede:

- Cambiar el *shell* de otros usuarios.
- Agregar nuevos intérpretes de comandos al sistema.

Ejercicio 6: El Sistema de Archivos (*File System*) en Linux.

(a) *¿Qué es?*

El sistema de archivos (*File System*) en Linux es la estructura lógica y jerárquica mediante la cual el sistema operativo organiza, almacena y administra los datos en los dispositivos de almacenamiento (como discos duros, SSD, memorias USB, etc.). En otras palabras, es la forma en que Linux ve y maneja los archivos y los directorios.

Funciones principales del *File System*:

- Organización: Estructura los datos en archivos y directorios dentro de una jerarquía.
- Gestión de acceso: Controla quién puede leer, escribir o ejecutar un archivo mediante permisos.
- Identificación: Cada archivo tiene nombre, ruta y atributos (tamaño, propietario, fecha de modificación, etc.).
- Abstracción del *hardware*: Permite al usuario trabajar con archivos sin preocuparse por cómo se almacenan, físicamente, en el disco.
- Montaje de dispositivos: Linux unifica todos los dispositivos bajo una única estructura de directorios, comenzando desde "/" (la raíz).

(b) *¿Cuál es la estructura básica de los File System en GNU/Linux? Mencionar los directorios más importantes e indicar qué tipo de información se encuentra en ellos. ¿A qué hace referencia la sigla FHS?*

El sistema de archivos (*File System*) de GNU/Linux tiene una estructura jerárquica en forma de árbol invertido, cuyo punto de partida es el directorio raíz /. A partir de él, se ramifican todos los demás directorios, que contienen archivos del sistema, configuraciones, programas y datos de los usuarios. Todo en Linux es un archivo: programas, dispositivos, directorios y procesos se representan como archivos dentro de esta estructura.

Directorios más importantes:

- /: Raíz del sistema de archivos. Contiene todos los demás directorios.
- */bin*: Programas básicos del sistema (comandos esenciales como *ls*, *cp*, *mv*, *cat*).
- */boot*: Archivos necesarios para el arranque del sistema, como el *kernel* (*vmlinuz*) y el GRUB.
- */dev*: Archivos que representan dispositivos del *hardware* (por ejemplo, */dev/sda* para discos).
- */etc*: Archivos de configuración del sistema y de los programas instalados.
- */home*: Directorios personales de los usuarios (por ejemplo, */home/jmenduina*).
- */lib*: Bibliotecas compartidas necesarias para que funcionen los programas del sistema.

- */media*: Puntos de montaje automático para dispositivos externos (pendrives, CDs).
- */mnt*: Punto de montaje temporal para dispositivos o particiones.
- */opt*: Aplicaciones opcionales instaladas, manualmente, por el usuario.
- */proc*: Archivos virtuales con información del sistema y procesos en ejecución.
- */root*: Directorio personal del usuario administrador (*root*).
- */run*: Información temporal del sistema y procesos activos desde el arranque.
- */sbin*: Programas del sistema usados por el administrador (por ejemplo, *reboot*, *ifconfig*).
- */srv*: Datos de servicios del sistema (por ejemplo, archivos servidos por un servidor *web*).
- */tmp*: Archivos temporales. Se eliminan, automáticamente, al reiniciar.
- */usr*: Contiene la mayoría de los programas y las utilidades de usuario. Incluye */usr/bin*, */usr/lib*, */usr/share*.
- */var*: Archivos variables (logs del sistema, colas de impresión, correos, etc.).

La sigla FHS hace referencia a "*Filesystem Hierarchy Standard*", es decir, Estándar de Jerarquía del Sistema de Archivos. Es una norma que define la estructura y el propósito de cada directorio en sistemas GNU/Linux y similares a Unix. Su objetivo es garantizar que todas las distribuciones de Linux mantengan una organización coherente y que los programas puedan encontrar archivos y directorios en las mismas rutas, independientemente de la distribución (Debian, Ubuntu, Fedora, etc.).

(c) *Mencionar sistemas de archivos soportados por GNU/Linux.*

GNU/Linux es compatible con una gran cantidad de sistemas de archivos, tanto nativos como de otros sistemas operativos. Algunos de los más importantes son:

- ext2 (*Second Extended File System*): Fue el sistema estándar en las primeras distribuciones. No posee *journaling* (registro de cambios), lo que lo hace más simple, pero menos seguro ante fallos.
- ext3 (*Third Extended File System*): Evolución de ext2, agrega *journaling*, mejorando la recuperación ante apagados o errores inesperados.
- ext4 (*Fourth Extended File System*): Es el más usado actualmente. Ofrece soporte para archivos muy grandes, mejor rendimiento, menor fragmentación y *journaling* más eficiente.
- Btrfs (*B-tree File System*): Sistema moderno diseñado para reemplazar a ext4. Soporta *snapshots*, compresión, verificación de integridad y administración avanzada de volúmenes.
- XFS: Desarrollado por *Silicon Graphics*, es ideal para sistemas con archivos muy grandes y de alto rendimiento (servidores, bases de datos).
- ReiserFS/Reiser4: Destacado por su eficiencia en manejo de archivos pequeños, aunque, actualmente, está en desuso.

(d) *¿Es posible visualizar particiones del tipo FAT y NTFS (que son de Windows) en GNU/Linux?*

Sí, es posible visualizar particiones del tipo FAT y NTFS (que son de Windows) en GNU/Linux. El *kernel* de Linux incluye soporte para leer y escribir en varios sistemas de archivos, entre ellos los usados por Windows.

Ejercicio 7: Particiones.

(a) *Definición. Tipos de particiones. Ventajas y desventajas.*

Una partición es una división lógica dentro de un disco duro físico. Cada partición se comporta como una unidad de almacenamiento independiente, lo que permite organizar los datos o instalar varios sistemas operativos en un mismo disco.

Tipos de particiones:

1. **Partición primaria:**
   - Es la partición principal del disco.
   - Puede haber hasta 4 particiones primarias en un disco (según el esquema MBR).
   - Una de ellas puede marcarse como "activa" para arrancar el sistema operativo.
2. **Partición extendida:**
   - Es un tipo especial de partición que no almacena datos directamente, sino que contiene particiones lógicas.
   - Sólo puede existir una extendida por disco, pero permite superar el límite de 4 particiones del MBR.
3. **Particiones lógicas:**
   - Se crean dentro de la partición extendida.
   - Permiten tener más de 4 particiones en total.
   - Se comportan igual que las primarias para el usuario y el sistema operativo.
4. **Particiones de arranque, sistema y *swap* (en Linux):**
   - */boot*: contiene los archivos necesarios para iniciar el sistema.
   - */(root)*: contiene el sistema principal.
   - *swap*: espacio en disco usado como memoria virtual.

Ventajas de particionar un disco:

- Permite instalar varios sistemas operativos en el mismo equipo (por ejemplo, Windows y Linux).
- Facilita organizar los datos (sistema, usuarios, *backups*, etc.).
- Mejora la seguridad y recuperación de datos: si una partición se daña, las otras pueden mantenerse intactas.
- Posibilita distintos sistemas de archivos en un mismo disco (ext4, NTFS, FAT32, etc.).

Desventajas de particionar un disco:

- Una partición puede quedarse sin espacio mientras otra tiene de sobra.
- La modificación del tamaño de las particiones puede ser riesgosa si no se realiza correctamente.
- Una mala configuración de particiones puede impedir el arranque del sistema.

(b) *¿Cómo se identifican las particiones en GNU/Linux? (Considerar discos IDE, SCSI y SATA).*

En GNU/Linux, las particiones se identifican mediante nombres asignados a los dispositivos de bloque que representan los discos y sus divisiones. Cada disco y partición se muestra como un archivo dentro del directorio */dev*.

- Discos IDE (antiguos): Se identifican con el prefijo *hd* (*hard disk*).
- Discos SCSI y SATA (actuales): Se identifican con el prefijo *sd* (*SCSI disk*).

(c) *¿Cuántas particiones son necesarias, como mínimo, para instalar GNU/Linux? Nombrarlas, indicando tipo de partición, identificación, tipo de File System y punto de montaje.*

Como mínimo, para instalar GNU/Linux, son necesarias dos particiones: una para el sistema principal (raíz */*) y otra para la memoria de intercambio (*swap*). Aunque es posible instalar todo en una sola partición (usando un archivo de intercambio en lugar de *swap*), la práctica recomendada es usar dos particiones separadas.

1. **Partición raíz (*/*):**
   - Tipo de partición: primaria (o lógica).
   - Identificación típica: */dev/sda1*.
   - Tipo de *File System*: *ext4* (el más común, aunque puede ser *btrfs*, *xfs*, etc.).
   - Punto de montaje: */*.
2. **Partición de intercambio (*swap*):**
   - Tipo de partición: primaria (o lógica).
   - Identificación típica: */dev/sda2*.
   - Tipo de *File System*: *swap*.
   - Punto de montaje: no tiene; el sistema la utiliza, directamente, como memoria virtual.

(d) *Dar ejemplos de diversos casos de particionamiento, dependiendo del tipo de tarea que se deba realizar en el sistema operativo.*

El esquema de particiones en GNU/Linux puede variar según el tipo de usuario o servidor, ya que diferentes tareas requieren diferentes niveles de organización, seguridad y rendimiento. Algunos casos típicos son:

- Instalación básica o de escritorio personal: Ideal para usuarios comunes o principiantes.
- Servidor *web* o de bases de datos: En servidores, se recomienda separar directorios críticos por seguridad, rendimiento y administración.
- Estación de trabajo o entorno de desarrollo: Pensado para programadores o usuarios avanzados que compilan *software*.

- Servidor de archivos o NAS: Pensado para almacenamiento masivo de datos.

(e) *¿Qué tipo de software para particionar existe? Mencionarlos y comparar.*

El *software* de particionado permite crear, modificar, redimensionar, eliminar o formatear particiones en un disco. Estos programas pueden ejecutarse desde el propio sistema operativo o desde un medio externo (por ejemplo, un *Live CD/USB*).

Herramientas de particionado en GNU/Linux:

- *fdisk*: Herramienta clásica para gestionar particiones en discos MBR. Permite crear, borrar y listar particiones. No soporta GPT.
- *parted*: Soporta discos con tablas de partición MBR y GPT. Permite redimensionar particiones y modificar el esquema del disco sin reiniciar.
- *cddisk*: Interfaz más amigable que *fdisk*. Ideal para usuarios que prefieren una vista de menú en consola.
- *gparted*: Versión con interfaz visual de *parted*. Permite mover, redimensionar y crear particiones fácilmente. Incluye soporte para FAT, NTFS, ext4, etc.
- *KDE Partition Manager*: Similar a *GParted*, pero integrado en entornos KDE (como Kubuntu).

Herramientas de particionado en otros sistemas operativos:

- *Disk Management* (Administración de discos): Permite crear y formatear particiones básicas (NTFS, FAT32). Limitado para operaciones avanzadas.
- *Disk Utility* (Utilidad de discos): Permite gestionar volúmenes HFS+, APFS y FAT. Posee funciones de reparación y formateo, pero con poca compatibilidad fuera de macOS.
- *EaseUS Partition Master/AOMEI/MiniTool*: Programas de terceros, con interfaz gráfica avanzada. Permiten redimensionar particiones sin pérdida de datos, clonar discos, etc.

Ejercicio 8: Arranque (*Bootstrap*) de un Sistema Operativo.

(a) *¿Qué es el BIOS? ¿Qué tarea realiza?*

BIOS significa *Basic Input/Output System* (Sistema Básico de Entrada/Salida). Es un *firmware* (*software* grabado en un chip de la placa madre) que se ejecuta al encender la computadora.

El BIOS se encarga de inicializar y probar el *hardware* antes de que arranque el sistema operativo. Esta fase se conoce como POST (*Power-On Self Test*). Durante el POST, el BIOS: verifica la memoria RAM; detecta dispositivos conectados (discos, teclado, USB, tarjetas de video, etc.); inicializa controladores básicos del *hardware*.

Las tareas principales del BIOS son:

- *Autotest* del *hardware* (POST): Comprueba que todos los componentes esenciales funcionen correctamente.
- Detección de dispositivos de arranque: Busca discos duros, CD/DVD, USB u otros medios desde donde pueda cargar el sistema operativo.
- Cargar el *bootstrap/bootloader*: Una vez localizado el dispositivo de arranque, el BIOS carga el primer sector (MBR) del disco en la memoria y transfiere el control al *bootloader* (por ejemplo, GRUB en Linux).
- Proporcionar servicios básicos de E/S: Permite que el sistema operativo y los programas interactúen con dispositivos básicos (teclado, pantalla, discos) antes de que existan controladores propios del sistema operativo.

(b) *¿Qué es UEFI? ¿Cuál es su función?*

UEFI significa *Unified Extensible Firmware Interface* (Interfaz de *Firmware* Extensible Unificada). Es un reemplazo moderno del BIOS, que proporciona una interfaz más avanzada entre el *firmware* de la placa madre y el sistema operativo. UEFI no sólo inicializa el *hardware*, sino que también soporta discos grandes, arranque seguro y un entorno más flexible que el BIOS clásico.

El UEFI se encarga de:

1. Inicializar y probar el *hardware* al encender la computadora, igual que el BIOS (POST).
2. Detectar y gestionar dispositivos de arranque (discos, USB, red, CD/DVD).
3. Cargar el *bootloader* o el cargador de arranque del sistema operativo.
4. Proporcionar funciones avanzadas de *firmware*, como:
   - Arranque seguro (*Secure Boot*): Garantiza que sólo se ejecute *software* firmado y confiable.
   - Soporte para discos grandes (> 2 TB) mediante GPT (*GUID Partition Table*).
   - Interfaz gráfica y compatibilidad con *mouse*.
   - Capacidad de ejecutar aplicaciones pequeñas antes de cargar el sistema operativo.

(c) *¿Qué es el MBR? ¿Qué es el MBC?*

MBR significa *Master Boot Record* (Registro de Arranque Maestro). Es el primer sector de un disco duro (sector 0, generalmente 512 bytes). Contiene información esencial para arrancar el sistema operativo y la tabla de particiones del disco.

MBC significa *Master Boot Code* (Código de Arranque Maestro). Es la parte del MBR que contiene el código ejecutable, es decir, el programa que inicia el arranque del sistema operativo. Cuando el BIOS termina el POST y transfiere el control al MBR, el MBC se ejecuta, decide qué partición es arrancable y carga el *bootloader* de esa partición.

(d) *¿A qué hacen referencia las siglas GPT? ¿Qué sustituye? Indicar cuál es su formato.*

GPT significa *GUID Partition Table* (Tabla de Particiones con Identificador Global Único). Es un nuevo esquema de particionamiento que sustituye al MBR en discos modernos. Se utiliza, principalmente, en sistemas con UEFI, aunque también puede usarse en BIOS *Legacy* con compatibilidad.

(e) *¿Cuál es la funcionalidad de un "Gestor de Arranque"? ¿Qué tipos existen? ¿Dónde se instalan? Citar gestores de arranque conocidos.*

Un "Gestor de Arranque" (*Bootloader*) es un programa que se ejecuta al inicio del sistema para cargar el sistema operativo en memoria y pasarle el control al *kernel*. Su funcionalidad es permitir que el sistema operativo se inicie correctamente y, en muchos casos, elegir entre varios sistemas operativos instalados (*dual boot*). Se instalan en MBR (BIOS) o en EFI *System Partition* (UEFI).

Tipos de gestores de arranque:

- Primera etapa (*Stage* 1): Programa mínimo ubicado en el primer sector del disco (MBR o EFI *partition*). Su tarea principal es localizar el *Stage* 2 o el *kernel*.
- Segunda etapa (*Stage* 2): Programa más completo que muestra menú de arranque, carga módulos y pasa parámetros al *kernel*.
- Gestores de arranque por BIOS vs. UEFI.

Gestores de arranque conocidos:

- GRUB (*GRand Unified Bootloader*): El más común en GNU/Linux. Soporta *multi-boot* y UEFI/BIOS.
- LILO (Linux *Loader*): Más antiguo, sólo BIOS, ya en desuso.
- *systemd-boot*: Ligero, para sistemas UEFI, integrado con *systemd*.
- *rEFInd*: Especialmente útil en sistemas UEFI con varios sistemas operativos.

- Windows *Boot Manager*: Arranca Windows y puede integrarse con otros sistemas operativos vía UEFI.

(f) *¿Cuáles son los pasos que se suceden desde que se prende una computadora hasta que el sistema operativo es cargado (proceso de bootstrap)?*

El *bootstrap* es el proceso mediante el cual la computadora inicia desde cero y carga el sistema operativo en memoria.

1. **Encendido y energía:**
   - Al presionar el botón de encendido, la fuente de alimentación entrega energía a todos los componentes del sistema.
2. **POST (*Power-On Self Test*):**
   - Ejecutado por el BIOS o UEFI.
   - Comprueba que RAM, CPU, teclado, tarjetas de video y discos funcionen correctamente.
   - Si hay errores críticos, se detiene y puede emitir *beeps* o mensajes de error.
3. **Inicialización del *hardware*:**
   - BIOS/UEFI configura los dispositivos básicos y prepara el entorno para cargar el sistema operativo.
4. **Búsqueda del dispositivo de arranque:**
   - BIOS/UEFI revisa la secuencia de arranque (disco duro, USB, CD/DVD, red).
   - Encuentra un sector de arranque válido (MBR o EFI *System Partition*).
5. **Carga del *bootloader* (gestor de arranque):**
   - BIOS/UEFI transfiere el control al bootloader, que puede estar en MBR (para BIOS) EFI *Partition* (para UEFI).
6. **Ejecución del *bootloader*:**
   - El *bootloader* muestra un menú de sistemas operativos si hay más de uno.
   - Carga el *kernel* del sistema operativo en memoria.
   - Pasa parámetros al *kernel* (por ejemplo, modo de arranque o resolución de pantalla).
7. **Inicialización del *kernel*:**
   - El *kernel* detecta el *hardware*, monta la partición raíz (*/*) y configura los controladores y servicios esenciales.
8. **Carga del sistema de inicio (*init* o *systemd*):**
   - El *kernel* transfiere el control al sistema de inicio, que inicia los servicios y *deamons*.
   - Configura la interfaz gráfica o de consola según el sistema.
9. ***Login* del usuario:**
   - Finalmente, se muestra la pantalla de *login* o el *prompt* de terminal, listo para usar el sistema operativo.

(g) *Analizar el proceso de arranque en GNU/Linux y describir sus principales pasos.*

El arranque de GNU/Linux sigue los pasos generales del *bootstrap*, pero tiene particularidades propias debido a su *kernel* y sistema de inicio.

(h) *¿Cuáles son los pasos que se suceden en el proceso de parada (shutdown) de GNU/Linux?*

El *shutdown* es el proceso mediante el cual GNU/Linux termina la ejecución del sistema operativo de manera ordenada, asegurando la integridad de los datos y del *hardware*.

Los pasos que se suceden en el proceso de parada (*shutdown*) de GNU/Linux son:

1. **Iniciar el proceso de apagado:**
   - Se ejecuta un comando como *shutdown -h now* o mediante el menú gráfico.
   - El sistema avisa a todos los usuarios conectados que el sistema se apagará.
2. **Notificar a los procesos:**
   - El sistema de inicio (*systemd* o *init*) envía señales SIGTERM a todos los procesos en ejecución.
   - Los procesos deben terminar ordenadamente, cerrando archivos abiertos y guardando datos.
3. **Finalización de los procesos:**
   - Después de un tiempo de espera, el sistema envía señales SIGKILL a los procesos que no respondieron, forzando su cierre.
4. **Detener servicios y *deamons*:**
   - *systemd* o *init* detienen todos los servicios activos (redes, servidores gráficos, servidores de impresión, cron, etc.).
5. **Desmontar sistemas de archivos:**
   - Se desmontan todas las particiones montadas, garantizando que no haya pérdida de datos.
   - Se vacían *buffers* de escritura en disco.
6. **Sincronizar y apagar *hardware*:**
   - Se sincroniza el disco duro para asegurar que todos los datos se hayan escrito.
   - Se apaga el *hardware* (CPU, ventiladores, dispositivos) o se reinicia si se solicitó.

(i) *¿Es posible tener, en una PC, GNU/Linux y otro sistema operativo instalado? Justificar.*

Sí, es posible tener, en una PC, GNU/Linux y otro sistema operativo instalado. Este esquema se conoce como *dual boot* y permite elegir qué sistema operativo iniciar al encenderla computadora.

- Cada sistema operativo necesita su propio entorno de archivos y controladores.
- Con particiones separadas y un gestor de arranque compatible, no hay conflictos entre los sistemas.

- Es una práctica común para usuarios que necesitan Linux para desarrollo y Windows para aplicaciones específicas.

Ejercicio 9: Archivos y Editores.

(a) *¿Cómo se identifican los archivos en GNU/Linux?*

En GNU/Linux, los archivos se identifican, principalmente, por su nombre y ubicación en el sistema de archivos.

(b) *Investigar el funcionamiento de los editores vim, nano y mcedit, y de los comandos cat, more y less.*

Editores:

- *vim*: Editor de texto avanzado en terminal.
- *nano*: Editor de texto sencillo en terminal.
- *mcedit*: Editor de texto incluido en *Midnight Commander* (MC), un gestor de archivos en terminal.

Comandos:

- *cat*: Muestra el contenido completo de un archivo en pantalla.
- *more*: Muestra contenido de un archivo página por página.
- *less*: Similar a *more*, pero permite avanzar y retroceder en el archivo.

(c) *Crear un archivo llamado "prueba.exe" en el directorio personal usando el vim. El mismo debe contener el número de alumno y el nombre.*

vim /home/jmenduina/prueba.exe
12345 - Juan Menduiña
:exit
cat /home/jmenduina/prueba.exe

(d) *Investigar el funcionamiento del comando file. Probarlo con diferentes archivos. ¿Qué diferencia se nota?*

El comando *file* determina el tipo de un archivo, examinando su contenido (no se fija sólo en la extensión). Lo hace con una base de datos de "*magic numbers*" (patrones binarios, firmas, cabeceras) y reglas, usando la librería *libmagic*. Puede decir si un archivo es texto, *script*, ejecutable ELF, imagen, tar gzip, dispositivo, enlace simbólico, etc.

(e) *Investigar la funcionalidad y los parámetros de los siguientes comandos relacionados con el uso de archivos:*

- *cd*: Cambia el directorio de trabajo actual.
- *mkdir*: Crea directorios.
- *rmdir*: Elimina directorios vacíos.
- *ln*: Crea enlaces (*links*) entre archivos (*hard links* o *symbolic links*).
- *tail*: Muestra las últimas líneas de un archivo.
- *locate*: Busca archivos por nombre usando una base de datos indexada.
- *ls*: Lista el contenido de directorios.
- *pwd*: Imprime el directorio de trabajo actual.
- *cp*: Copia archivos y directorios.
- *mv*: Mueve o renombra archivos/directorios.
- *find*: Busca archivos y directorios en tiempo real recursivamente, con criterios muy flexibles. Es más lento que *locate*, pero preciso y poderoso.

Ver la ayuda/guía de cada comando:

- *man comando*.
- *comando --help*.

Ejercicio 10.

Indicar qué comando es necesario utilizar para realizar cada una de las siguientes acciones. Investigar el funcionamiento y los parámetros más importantes:

(a) *Crear la carpeta ISOCSO.*

mkdir /home/jmenduina/ISOCSO

(b) *Acceder a la carpeta.*

cd /home/jmenduina/ISOCSO

(c) *Crear dos archivos con los nombres isocso.txt e isocso.csv.*

touch /home/jmenduina/ISOCSO/isocso.txt /home/jmenduina/ISOCSO/isocso.csv
ls /home/jmenduina/ISOCSO

(d) *Listar el contenido del directorio actual.*

ls

(e) *Visualizar la ruta donde se está situado.*

pwd

(f) *Buscar todos los archivos en los que su nombre contiene la cadena "iso*".*

find . -type f -name "iso"*

(g) *Informar la cantidad de espacio libre en disco.*

df -h

(h) *Verificar los usuarios conectados al sistema.*

w

(i) *Editar el archivo isocso.txt e ingresar Nombre y Apellido.*

```
echo "Juan Menduina" > /home/jmenduina/ISOCSO/isocso.txt
cat /home/jmenduina/ISOCSO/isocso.txt
```

(j) *Mostrar en pantalla las últimas líneas de un archivo.*

```
tail -n 5 /home/jmenduina/ISOCSO/isocso.txt
```

Ejercicio 11.

Investigar funcionamiento, parámetros y ubicación (directorio) de los siguientes comandos:

(a) *man.*

Muestra la página de manual de otros comandos y utilidades (documentación local).

Ubicación típica: */bin/man* o */usr/bin/man*.

(b) *shutdown.*

Inicia el apagado ordenado del sistema o permite programarlo para más tarde; notifica a usuarios y detiene servicios.

Ubicación típica: */sbin/shutdown* o */usr/sbin/shutdown*.

(c) *reboot.*

Reinicia el sistema (sincroniza, detiene servicios y apaga para, luego, volver a arrancar).

Ubicación típica: */sbin/reboot* o */usr/sbin/reboot*.

(d) *halt.*

Detiene (para) el sistema; en muchos sistemas, *halt* deja el *hardware* en estado detenido.

Ubicación típica: */sbin/halt* o */usr/sbin/halt*.

(e) *uname.*

Muestra información del sistema operativo/*kernel* (nombre, versión, arquitectura).

Ubicación típica: */bin/uname* o */usr/bin/uname*.

(f) *dmesg.*

Muestra el *buffer* de mensajes del *kernel* (mensajes de arranque y eventos del *kernel*: detección de *hardware*, *drivers*, errores). Muy útil para diagnóstico *hardware*/arranque.

Ubicación típica: */bin/dmesg* o */usr/bin/dmesg*.

(g) *lspci.*

Lista los dispositivos PCI detectados (tarjetas gráficas, controladoras, etc.). Forma parte del paquete *pciutils*.

Ubicación típica: */usr/bin/lspci*.

(h) *at.*

Programa la ejecución de comandos una sola vez en un momento futuro (*job scheduler* simple). Interactúa con el demonio *atd*.

Ubicación típica: */usr/bin/at*.

(i) *netstat.*

Muestra conexiones de red, tablas de *routing*, estadísticas de interfaces, puertos abiertos y *sockets*. Forma parte del paquete *net-tools* (antiguo).

Ubicación típica: */bin/netstat* o */usr/bin/netstat*.

(j) *head.*

Muestra las primeras líneas de uno o varios archivos. Muy útil para echar un vistazo rápido.

Ubicación típica: */bin/head* o */usr/bin/head*.

(k) *tail.*

Muestra las últimas líneas de un archivo; tiene modo seguimiento para ver nuevas líneas en tiempo real (ideal para logs).

Ubicación típica: */bin/tail* o */usr/bin/tail*.

Ejercicio 12: Procesos.

(a) *¿Qué es un proceso? ¿A qué hacen referencia las siglas PID y PPID? ¿Todos los procesos tienen estos atributos en GNU/Linux? Justificar. Indicar qué otros atributos tiene un proceso.*

Un proceso es un programa en ejecución, es decir, una instancia activa de un programa que está siendo ejecutada por el sistema operativo. Incluye:

- El código del programa (instrucciones ejecutables).
- Los datos del programa (variables, buffers, etc.).
- El estado de ejecución, que permite al sistema operativo retomar la ejecución donde se quedó.
- Recursos asignados, como memoria, archivos abiertos, y dispositivos.

En GNU/Linux, cada tarea que el *kernel* ejecuta se considera un proceso (incluyendo *deamons* y *threads*).

- PID significa *Process ID* y es el identificador único que el sistema operativo asigna a cada proceso.
- PPID significa *Parent Process ID* y es el identificador del proceso padre, es decir, el que creó o generó este proceso.

En GNU/Linux, todos los procesos tienen PID y PPID. Esto es esencial para la jerarquía de procesos:

- PID único para identificar el proceso en el *kernel*.
- PPID para establecer la relación padre-hijo, lo que permite terminar procesos hijos al finalizar el padre o que los procesos *zombies* sean adoptados por *init/systemd*.

Los procesos *kernel threads* pueden tener peculiaridades en su PPID, pero, incluso ellos, tienen un PID para la gestión por el *kernel*.

Además de PID y PPID, un proceso tiene otros atributos:

- UID/GID: Identificador de usuario y grupo propietario.
- Estado del proceso: *Running*, *Sleeping*, *Stopped*, *Zombie*.
- Prioridad/*Nice Value*: Controla el tiempo de CPU asignado.
- Memoria asignada: Tamaño de *stack*, *heap*, código y memoria residente.
- Archivos abiertos: Lista de descriptores de archivos y dispositivos asociados.
- Señales pendientes/máscara de señales: Para comunicación entre procesos.
- Tiempo de CPU usado: Usuario y sistema.
- Comando ejecutado: Nombre del programa o ruta.

(b) *Investigar funcionamiento, parámetros y ubicación (directorio) de los siguientes comandos relacionados a procesos. En caso de que algún comando no venga por defecto en la distribución que se utiliza, se deberá proceder a instalarlo:*

- *top*: Muestra, en tiempo real, los procesos activos y el uso de recursos del sistema (CPU, memoria, etc.). Ubicación típica: */usr/bin/top*.
- *htop*: Versión más amigable de *top* con interfaz colorida e interactiva, permite usar el cursor y menú para matar procesos, cambiar prioridad, etc. Ubicación típica: */usr/bin/htop*.
- *ps*: Muestra un *snapshot* de los procesos activos (no en tiempo real). Ubicación típica: */bin/ps* o */usr/bin/ps*.
- *pstree*: Muestra la jerarquía de procesos en forma de árbol, indicando padres e hijos. Ubicación típica: */usr/bin/pstree*.
- *kill*: Envía señales a un proceso (por defecto, SIGTERM para terminarlo). Ubicación típica: */bin/kill* o */usr/bin/kill*.
- *pgrep*: Busca procesos por nombre o patrón, devuelve PID(s). Ubicación típica: */usr/bin/pgrep*.
- *pkill*: Envía señales a procesos por nombre (como *kill*, pero buscando por nombre). Ubicación típica: */usr/bin/pkill*.
- *killall*: Envía señal a todos los procesos con un nombre exacto. Similar a *pkill*. Ubicación típica: */usr/bin/killall*.
- *renice*: Cambia la prioridad de ejecución (*nice value*) de un proceso activo. Ubicación típica: */usr/bin/renice*.
- *xkill*: Permite cerrar ventanas gráficas con el *mouse* (interfaz X11). Ubicación típica: */usr/bin/xkill*.
- *atop*: Monitor de procesos y recursos en tiempo real y más detallado que *top*. Puede registrar histórico de uso. Ubicación típica: */usr/bin/atop*.
- *nice*: Ejecuta un comando con un valor de prioridad distinto (*nice value*). Ubicación típica: */usr/bin/nice*.

Ver la ayuda/guía de cada comando:

- *man comando*.
- *comando --help*.

Ejercicio 13: Proceso de Arranque *SystemV* (https://github.com/systeminit/si/).

(a) *Enumerar los pasos del proceso de inicio de un sistema GNU/Linux, desde que se prende la PC hasta que se logra obtener el login en el sistema.*

Pasos del proceso de inicio de un sistema GNU/Linux:

1. **POST (*Power-On Self Test*):**
   - Cuando se prende la PC, la BIOS (o UEFI en sistemas modernos) realiza un POST, verificando *hardware* básico (memoria RAM, teclado, CPU, tarjetas de video, discos, etc.).
   - Se inicializan los dispositivos esenciales y se preparan para que el sistema operativo arranque.
2. **Localización del gestor de arranque:**
   - La BIOS busca el Master Boot Record (MBR) del primer disco de arranque.
   - En sistemas con UEFI, busca el EFI *System Partition* (ESP) para localizar un *bootloader* compatible.
   - El gestor de arranque (*bootloader*), como GRUB, se carga en memoria y se ejecuta.
3. **Carga del *kernel*:**
   - El *bootloader* muestra un menú (opcional) y carga el *kernel* de Linux en memoria junto con la *initramfs* (*filesystem* temporal).
   - La *initramfs* contiene los controladores esenciales y los *scripts* necesarios para montar el sistema de archivos raíz.
4. **Inicialización del *kernel*:**
   - El *kernel* inicializa gestión de memoria (RAM), dispositivos y controladores y subsistemas del *kernel* (*scheduler*, manejo de permisos, sistemas de archivos, etc.).
   -
5. **Ejecución del proceso *init*:**
   - Una vez que el *kernel* está listo, ejecuta */sbin/init*, que es el primer proceso usuario (PID= 1).
   - En *SystemV*, *init* es responsable de configurar la consola e iniciar *runlevel* correspondiente.
6. ***Runlevels* (niveles de ejecución):**
   - *SystemV* organiza servicios en *runlevels*.
   - Según el *runlevel* configurado, *init* ejecuta los *scripts* de inicio en */etc/rc.d/* o */etc/rc#.d/*.
7. **Inicialización de servicios:**
   - Se ejecutan servicios esenciales:
     - Red (*networking*).
     - *Daemons* del sistema (*cron*, *syslog*, etc.).
     - Montaje de sistemas de archivos adicionales (*/home*, */var*, etc.).
     - Cualquier servicio configurado para ese *runlevel*.
8. ***Login*:**
   - Finalmente, se lanza el *getty* en las terminales virtuales (TTY) o el *display manager* (si es modo gráfico).

- El usuario ve la pantalla de *login* y puede autenticarse para empezar a usar el sistema.

(b) *Proceso INIT. ¿Quién lo ejecuta? ¿Cuál es su objetivo?*

El proceso *INIT* (en sistemas *SystemV*) es el primer proceso que ejecuta el *kernel* de Linux una vez que termina su propia inicialización. En particular:

- El *kernel* lo lanza automáticamente.
- Se encuentra en la ruta */sbin/init*.
- Su PID (*Process ID*) siempre es 1, lo que lo convierte en el padre de todos los procesos del sistema.

El objetivo del proceso *init* es inicializar el espacio de usuario (*user space*) del sistema operativo. Esto incluye:

- Ejecutar los *scripts* de arranque definidos en */etc/inittab* y en los directorios */etc/rc.d/* o */etc/init.d/*.
- Montar sistemas de archivos, iniciar servicios básicos (*red*, *syslog*, etc.).
- Cambiar al nivel de ejecución (*runlevel*) configurado por defecto (por ejemplo, modo texto o modo gráfico).
- Iniciar los procesos *getty* que muestran los *prompts* de *login* en las terminales.
- Mantener el control de los procesos del sistema (reiniciar servicios, apagar correctamente, etc.).

(c) *RunLevels. ¿Qué son? ¿Cuál es su objetivo?*

Los *RunLevels* (niveles de ejecución) son modos de operación predefinidos en los sistemas GNU/Linux basados en *SystemV* (*SysVInit*). Cada *runlevel* representa un estado del sistema que determina qué servicios y procesos deben estar activos. En otras palabras, un *runlevel* indica qué conjunto de procesos el sistema debe iniciar o detener según la fase o el propósito de uso (por ejemplo, modo monousuario, modo multiusuario, modo gráfico, etc.).

El objetivo de los *runlevels* es permitir al administrador controlar el estado operativo del sistema, facilitando tareas como:

- Iniciar el sistema con distintos niveles de servicio.
- Cambiar entre modos (por ejemplo, del modo texto al gráfico).
- Detener o reiniciar el sistema de manera controlada.

(d) *¿A qué hace referencia cada nivel de ejecución según el estándar? ¿Dónde se define qué RunLevel ejecutar al iniciar el sistema operativo? ¿Todas las distribuciones respetan estos estándares?*

En los sistemas GNU/Linux basados en *SystemV Init*, los niveles de ejecución (*RunLevels*) representan diferentes modos de operación del sistema. Cada nivel indica qué servicios y procesos deben estar activos:

RunLevel	**Nombre/Estado**	**Descripción**
0	*Halt* (Apagado)	Apaga el sistema. Todos los procesos se detienen
1	*Single-user mode*	Modo de mantenimiento o recuperación. Sólo un usuario *root*, sin servicios de red
2	*Multi-user* (sin red)	Modo multiusuario, pero sin servicios de red
3	*Multi-user* (con red)	Modo multiusuario completo, en modo texto
4	*Undefined/Custom*	No se usa por defecto; reservado para configuraciones personalizadas
5	*Graphical* (X11)	Igual que runlevel 3, pero con entorno gráfico
6	*Reboot*	Reinicia el sistema

En sistemas con *SysVInit*, el *RunLevel* a ejecutar al iniciar el sistema operativo se define en el archivo */etc/inittab*.

Si bien el estándar *SysVInit* definió esos *runlevels*, no todas las distribuciones modernas lo respetan estrictamente, ya que muchas usan *systemd* como sistema de inicialización. En *systemd*, los *runlevels* fueron reemplazados por "*targets*", que cumplen la misma función pero con mayor flexibilidad.

(e) *Archivo /etc/inittab. ¿Cuál es su finalidad? ¿Qué tipo de información se almacena en él? ¿Cuál es la estructura de la información que en él se almacena?*

El archivo */etc/inittab* pertenece al sistema de inicialización *SystemV Init* (*SysVInit*). Su función principal es definir el comportamiento del proceso *init*, que es el primer proceso del sistema operativo (PID 1). En concreto, este archivo le indica a *init*:

- Qué *runlevel* debe cargarse por defecto.
- Qué procesos o *scripts* ejecutar en cada *runlevel*.
- Qué acciones realizar cuando se producen determinados eventos del sistema (reinicio, apagado, etc.).

Entonces, dentro de */etc/inittab*, se almacena información de configuración como:

- *RunLevel* por defecto (en qué modo arranca el sistema).

- Procesos que deben iniciarse, automáticamente, en cada *runlevel*.
- Configuración de terminales (por ejemplo, consolas *tty1*, *tty2*, etc.).
- Acciones asociadas a eventos como reinicio o apagado.

Cada línea de */etc/inittab* sigue la siguiente estructura: *id:runlevels:acción:proceso*, donde:

- *id*: Identificador único de la entrada.
- *runlevels*: Nivel(es) de ejecución donde se aplica esta entrada (0-6).
- *acción*: Qué tipo de acción debe realizar *init*.
- *proceso*: Comando o *script* que se ejecutará.

(f) *Suponer que se encuentra en el runlevel <X>. Indicar qué comando(s) se deberá ejecutar para cambiar al runlevel <Y>. ¿Este cambio es permanente? ¿Por qué?*

En los sistemas GNU/Linux basados en *SystemV Init* (*SysVInit*), el comando que se deberá ejecutar para cambiar al *runlevel* <Y> es *init <Y>*. Este cambio no es permanente, sólo dura mientras el sistema esté encendido; cuando el sistema se reinicia, vuelve al *runlevel* por defecto definido en el archivo */etc/inittab* (o su equivalente moderno en *systemd*).

(g) *Scripts RC. ¿Cuál es su finalidad? ¿Dónde se almacenan? Cuando un sistema GNU/Linux arranca o se detiene, se ejecutan scripts. Indicar cómo determina qué script ejecutar ante cada acción. ¿Existe un orden para llamarlos? Justificar.*

Los *scripts* RC (*Run Commands*) son *scripts* de inicio y apagado del sistema en GNU/Linux. Su funcionalidad es automatizar la carga y la detención de servicios (como red, impresoras, *deamons*, entorno gráfico, etc.) durante el arranque y el apagado del sistema. Se almacenan en */etc/init.d/*.

Cuando el sistema cambia de *runlevel* (por ejemplo, al iniciar o apagar), el proceso *init*:

- Consulta el nuevo *runlevel* al que debe cambiar.
- Busca en el directorio correspondiente (por ejemplo, */etc/rc3.d/*).
- Ejecuta los *scripts* dentro de ese directorio en orden alfabético.

Cada archivo comienza con una letra:

- S (de *Start*): Indica que el servicio se inicia.
- K (de *Kill*): Indica que el servicio se detiene.

Existe un orden para llamar a estos *scripts*. El orden lo define el número que sigue a la letra S o K y los *scripts* se ejecutan en orden ascendente, lo cual permite controlar las dependencias.

Ejercicio 14: *SystemD* (https://github.com/systemd/systemd/).

(a) *¿Qué es SystemD?*

SystemD es un sistema de inicialización y gestión de servicios moderno para sistemas operativos GNU/Linux, que reemplaza al clásico *SystemV Init* (*SysVInit*). Su principal objetivo es inicializar el sistema más rápido y de forma más eficiente, además de gestionar los servicios y los procesos del sistema durante toda su ejecución.

(b) *¿A qué hace referencia el concepto de Unit en SystemD?*

El concepto de *Unit* en *SystemD* hace referencia al componente básico de configuración y control. Cada *unit* representa un recurso del sistema que *SystemD* puede gestionar, supervisar o controlar, como un servicio, un punto de montaje, un dispositivo, un temporizador, etc. En otras palabras, una *unit* le indica a *SystemD* qué debe iniciarse, cuándo, cómo y en qué orden.

(c) *¿Para qué sirve el comando systemctl en SystemD?*

En *SystemD*, el comando *systemctl* sirve es la herramienta principal de administración. Permite controlar, inspeccionar y administrar tanto el estado general del sistema como los servicios (*units*) que gestiona *SystemD*.

(d) *¿A qué hace referencia el concepto de target en SystemD?*

En *SystemD*, el concepto de *target* hace referencia a una unidad especial (un tipo de *unit*) que agrupa y coordina el inicio o la detención de otros servicios, *sockets*, dispositivos, montajes, etc. En otras palabras, los *targets* definen estados o etapas del sistema, equivalentes a los niveles de ejecución (*runlevels*) del sistema *SysVInit*, pero con mayor flexibilidad.

(e) *Ejecutar el comando pstree. ¿Qué es lo que se puede observar a partir de la ejecución de este comando?*

Lo que se puede observar, a partir de la ejecución del comando *pstree*, es que el sistema muestra una vista jerárquica (en forma de árbol) de todos los procesos, actualmente, en ejecución, mostrando qué procesos son padres o hijos de otros. Es muy útil para comprender cómo se estructura el sistema de procesos, detectar relaciones y diagnosticar problemas de ejecución o dependencias entre procesos.

Ejercicio 15: Usuarios.

(a) *¿Qué archivos son utilizados en un sistema GNU/Linux para guardar la información de los usuarios?*

En un sistema GNU/Linux, para guardar la información de los usuarios, los archivos que son utilizados son:

- */etc/passwd*: Contiene la información básica de todas las cuentas de usuario.
- */etc/shadow*: Guarda las contraseñas cifradas y los parámetros de seguridad.
- */etc/group*: Contiene la información de los grupos del sistema.
- */etc/gshadow*: Similar a */etc/shadow*, pero para grupos.

(b) *¿A qué hacen referencia las siglas UID y GID? ¿Pueden coexistir UIDs iguales en un sistema GNU/Linux? Justificar.*

UID significa *User Identifier* (Identificador de Usuario). Es el número entero que el sistema asigna a cada usuario para identificarlo internamente (más allá de su nombre). Se usa en lugar del nombre de usuario para asignar propiedad y permisos sobre archivos, procesos y recursos del sistema.

GID significa *Group Identifier* (Identificador de Grupo). Indica a qué grupo pertenece, por defecto, el usuario. También se usa para determinar los permisos de grupo sobre archivos y procesos.

En un sistema GNU/Linux, pueden coexistir UIDs iguales, pero no es recomendable. El sistema permite crear dos cuentas distintas con el mismo UID. Sin embargo, si esto ocurre, ambos usuarios comparten los mismos permisos y propiedad sobre los archivos (ya que el sistema identifica por UID, no por nombre). En consecuencia, se pierde la separación de privilegios, lo que puede generar problemas de seguridad y administración. Por eso, cada usuario debe tener un UID único, salvo casos muy específicos (por ejemplo, cuentas técnicas que, intencionalmente, comparten permisos).

(c) *¿Qué es el usuario root? ¿Puede existir más de un usuario con este perfil en GNU/Linux? ¿Cuál es el UID de root?*

El usuario *root* es el superusuario del sistema GNU/Linux. Tiene todos los privilegios administrativos, es decir: puede acceder, modificar o eliminar cualquier archivo del sistema; puede crear, modificar o borrar usuarios; puede instalar o eliminar *software*; puede cambiar configuraciones del sistema, *kernel*, red, etc. Este usuario es esencial para la administración y el mantenimiento del sistema operativo.

En GNU/Linux, no puede existir más de un usuario con este perfil, pero sí es posible tener varios usuarios con privilegios equivalentes al *root*.

El UID de *root* es 0. Este valor es reservado y reconocido por el *kernel* como el identificador del superusuario.

(d) *Agregar un nuevo usuario llamado isocso a la instalación de GNU/Linux, especificar que su home sea creada en /home/isocso y hacerlo miembro del grupo informatica (si no existe, se deberá crear). Luego, sin iniciar sesión como este usuario, crear un archivo en su home personal que le pertenezca. Luego de todo esto, borrar el usuario y verificar que no queden registros de él en los archivos de información de los usuarios y grupos.*

```
getent group informatica || groupadd informatica
useradd -m -d /home/isocso -g informatica isocso
touch /home/isocso/isocso.txt
userdel -r isocso
grep isocso /etc/passwd
grep isocso /etc/shadow
grep informatica /etc/group
grep informatica /etc/gshadow
```

(e) *Investigar la funcionalidad y los parámetros de los siguientes comandos:*

- *useradd* y *adduser*: Crean un nuevo usuario en el sistema.
- *usermod*: Modifica un usuario existente.
- *userdel*: Elimina un usuario del sistema.
- *su*: Cambia de usuario en la sesión actual.
- *groupadd*: Crea un nuevo grupo.
- *who*: Muestra la información de usuarios conectados al sistema.
- *groupdel*: Elimina un grupo del sistema.
- *passwd*: Cambia la contraseña de un usuario.

Ver la ayuda/guía de cada comando:

- *man comando*.
- *comando --help*.

Ejercicio 16: *File System* y Permisos.

(a) *¿Cómo son definidos los permisos sobre archivos en un sistema GNU/Linux?*

En un sistema GNU/Linux, los permisos sobre archivos definen quién puede leer, escribir o ejecutar un archivo o directorio. Estos permisos están asociados a tres tipos de usuarios y a tres tipos de acciones.

Tipos de usuarios:

Cada archivo o directorio pertenece a:

- Usuario (u): El propietario del archivo.
- Grupo (g): Un conjunto de usuarios que comparten ciertos permisos.
- Otros (o): Todos los demás usuarios del sistema.

Tipos de permisos:

- Lectura (r):
  - Archivo: Permite leer el contenido del archivo.
  - Directorio: Permite listar el contenido del directorio.
- Escritura (w):
  - Archivo: Permite modificar o borrar el archivo.
  - Directorio: Permite crear, renombrar o eliminar archivos dentro del directorio.
- Ejecución (x):
  - Archivo: Permite ejecutar el archivo (si es un programa o un *script*).
  - Directorio: Permite acceder al directorio.

(b) *Investigar la funcionalidad y los parámetros de los siguientes comandos relacionados con los permisos en GNU/Linux:*

- *chmod*: Cambia los permisos de lectura (r), escritura (w) y ejecución (x) de un archivo o directorio.
- *chown*: Permite cambiar el usuario propietario y/o el grupo al que pertenece un archivo o directorio.
- *chgrp*: Cambia, únicamente, el grupo de un archivo o directorio (sin modificar el propietario).

Ver la ayuda/guía de cada comando:

- *man comando*.
- *comando --help*.

(c) *Al utilizar el comando chmod, generalmente, se utiliza una notación octal asociada para definir permisos. ¿Qué significa esto? ¿A qué hace referencia cada valor?*

La notación octal es una forma numérica (en base 8) de representar los permisos de archivos en GNU/Linux. Cada permiso (lectura, escritura, ejecución) se asocia a un valor numérico y la suma de esos valores define los permisos para cada categoría de usuarios.

Valores numéricos de los permisos:

- Lectura (r): 4.
- Escritura (w): 2.
- Ejecución (x): 1.

Tabla resumen de valores comunes:

Valor	Permisos	Significado
0	---	Sin permisos
1	*--x*	Sólo ejecución
2	*-w-*	Sólo escritura
3	*-wx*	Escritura y ejecución
4	*r--*	Sólo lectura
5	*r-x*	Lectura y ejecución
6	*rw-*	Lectura y escritura
7	*rwx*	Lectura, escritura y ejecución

(d) *¿Existe la posibilidad de que algún usuario del sistema pueda acceder a determinado archivo para el cual no posee permisos? Indicarlo y realizar las pruebas correspondientes.*

No, en principio, no existe la posibilidad de que algún usuario del sistema pueda acceder a determinado archivo para el cual no posee permisos. Sin embargo, el usuario *root* (superusuario) puede acceder, modificar o eliminar cualquier archivo, sin importar sus permisos. Esto se debe a que *root* tiene el UID 0 y el *kernel* le otorga control total sobre el sistema.

(e) *Explicar los conceptos de "full path name" (path absoluto) y "relative path name" (path relativo). Dar ejemplos claros de cada uno de ellos.*

Full path name (*path* absoluto) es la ruta completa desde el directorio raíz (*/*) hasta el archivo o carpeta deseada. Siempre comienza con /, que representa el *root directory*. Ejemplo: */home/jmenduina/informe.txt*.

Relative path name (*path* relativo) indica la ubicación en relación con el directorio actual (el que muestra *pwd*). No empieza con / y su interpretación depende de dónde se esté ubicado. Ejemplo: Suponiendo que se está en */home/jmenduina* y se quiere acceder a */home/jmenduina/informe.txt*, se puede hacer de forma relativa con *cat informe.txt*.

(f) *¿Con qué comando se puede determinar en qué directorio se encuentra actualmente? ¿Existe alguna forma de ingresar al directorio personal sin necesidad de escribir todo el path completo? ¿Se podría utilizar la misma idea para acceder a otros directorios? ¿Cómo? Explicar con un ejemplo.*

Con el comando *pwd* se puede determinar en qué directorio se encuentra actualmente.

Existen formas de ingresar al directorio personal sin necesidad de escribir todo el *path* completo. En particular, con *cd ~* o, simplemente, *cd*. Se puede utilizar la misma idea para acceder a otros directorios. Ejemplo: Suponiendo que existe un usuario llamado *juan*, se puede acceder a su carpeta personal (si se tienen los permisos) con *cd ~juan*.

(g) *Investigar la funcionalidad y los parámetros de los siguientes comandos relacionados con el uso del File System:*

- *umount*: Desmonta un sistema de archivos previamente montado para que deje de estar accesible desde el sistema.
- *du*: Muestra el espacio ocupado por archivos y directorios.
- *df*: Muestra el espacio disponible y usado en todos los sistemas de archivos montados.
- *mount*: Permite montar un dispositivo (partición, disco, ISO, USB) en un directorio del sistema.
- *mkfs*: Crea un sistema de archivos en una partición o disco.
- *fdisk* (con cuidado): Permite crear, eliminar o modificar particiones en discos duros.
- *write*: Envía mensajes a otro usuario conectado en el mismo sistema.
- *losetup*: Permite tratar un archivo como un dispositivo de bloque (útil para imágenes de disco o ISO).
- *stat*: Muestra información completa de un archivo o directorio (permisos, propietario, tamaño, fechas, inodo, etc.).

Ver la ayuda/guía de cada comando:

- *man comando*.
- *comando --help*.

Ejercicio 17: Procesos.

(a) *¿Qué significa que un proceso se está ejecutando en Background? ¿Y en Foreground?*

Que un proceso se está ejecutando en *Background* significa que se ejecuta "detrás de la terminal", permitiendo seguir usando la terminal, por lo que no bloquea la entrada de comandos.

Que un proceso se está ejecutando en *Foreground* significa que se ejecuta de manera interactiva en la terminal y, mientras lo hace, bloquea la terminal, es decir, no se pueden ingresar otros comandos hasta que termine.

(b) *¿Cómo se puede hacer para ejecutar un proceso en Background? ¿Cómo se puede hacer para pasar un proceso de background a foreground y viceversa?*

Para ejecutar un proceso en *Background*, simplemente, se agrega *&* al final del comando.

Para pasar un proceso de *background* a *foreground*, se presiona *Ctrl+Z*, lo cual suspende, temporalmente, el proceso y lo pone en estado detenido (*stopped*), y, luego, se envía a *background* con *bg %N*, donde *%N* es el número del *job*.

Para pasar un proceso de *foreground* a *background*, se usa *fg %N*.

(c) *Pipe (|). ¿Cuál es su finalidad? Citar ejemplos de su utilización.*

La finalidad de *Pipe* (|) es conectar la salida estándar de un comando con la entrada estándar de otro. En otras palabras, permite encadenar comandos, de modo que el resultado de uno se use como entrada del siguiente. Su principal objetivo es procesar información en una sola línea sin necesidad de crear archivos intermedios. Permite combinar comandos para filtrar, ordenar, contar o transformar datos.

La sintaxis general es: *comando1* | *comando2*. Esto significa *"Ejecutá comando1, y el resultado pásaselo, directamente, a comando2"*.

Ejemplos de su utilización:

- *ls -l* | *grep ".txt"*: Muestra sólo los archivos *.txt* del listado detallado.
- *ps aux* | *wc -l*: Muestra la cantidad de procesos en ejecución.
- *who* | *grep root*: Muestra si el usuario *root* tiene una sesión activa.
- *du -ah* | *sort -rh* | *head -n 10*: *du* calcula tamaños, *sort* ordena, *head* muestra los primeros 10.

(d) *Redirección. ¿Qué tipo de redirecciones existen? ¿Cuál es su finalidad? Citar ejemplos de su utilización.*

La redirección permite cambiar el destino o el origen de los flujos de datos de un comando. Por defecto:

- Entrada estándar (*stdin*) ⟶ viene del teclado.
- Salida estándar (*stdout*) ⟶ se muestra en pantalla.
- Salida de error (*stderr*) ⟶ también se muestra en pantalla, pero separada del *stdout*.

Con la redirección, se pueden enviar o recibir datos desde archivos u otros comandos.

Tipos de redirecciones:

Tipo	Descriptor	Símbolo	Finalidad
Entrada estándar	0	<	Leer datos desde un archivo en lugar del teclado
Salida estándar	1	>	Enviar la salida normal de un comando a un archivo (sobrescribe o lo crea si no existe)
Salida estándar (*append*)	1	>>	Agregar la salida al final de un archivo existente (si no existe, lo crea)
Salida de error estándar	2	*2>*	Redirigir sólo los errores a un archivo (sobreescribe o lo crea si no existe)
Salida de error (*append*)	2	*2>>*	Agregar errores al final de un archivo existente (si no existe, lo crea)
Combinar salida y error	1 y 2	*&>* o *2>&1*	Redirigir tanto salida como error al mismo destino

Ejemplos de su utilización:

- *sort < nombres.txt*: Lee los datos del archivo *nombres.txt* y los ordena (sin escribir en disco).
- *ls > listado.txt*: Guarda el resultado del comando *ls* en el archivo *listado.txt* (si existe, lo sobrescribe).
- *echo "Nueva línea" >> archivo.txt*: Agrega texto al final de *archivo.txt* sin borrar su contenido.

- *ls /directorio_que_no_existe 2> errores.txt*: Guarda el mensaje de error en *errores.txt*.
- *ls /home /directorio_que_no_existe &> salida_total.txt*: Guarda tanto la salida como los errores en el mismo archivo.

Ejercicio 18: Otros Comandos de Linux (Indicar Funcionalidad y Parámetros).

(a) *¿A qué hace referencia el concepto de empaquetar archivos en GNU/Linux?*

En GNU/Linux, el concepto de empaquetar archivos hace referencia al proceso de unir varios archivos y directorios en un solo contenedor, sin necesariamente comprimirlos. Esto se hace, principalmente, para facilitar su distribución, copia o respaldo.

(b) *Seleccionar 4 archivos dentro de algún directorio al que se tenga permiso y sumar el tamaño de cada uno de estos archivos. Crear un archivo empaquetado conteniendo estos 4 archivos y comparar los tamaños de los mismos. ¿Qué característica se nota?*

```
cd /home/jmenduina
touch file1.txt file2.txt file3.txt file4.txt
echo "Archivo 1" > file1.txt
echo "Archivo 2" > file2.txt
echo "Archivo 3" > file3.txt
echo "Archivo 4" > file4.txt
ls -l file*.txt
du -ch file*.txt | grep total
tar -cvf archivos.tar file*.txt
ls -l archivos.tar
```

La característica que se nota es que empaquetar no reduce el tamaño de los archivos (de hecho, lo aumenta), sólo los combina en uno solo.

(c) *¿Qué acciones se deben llevar a cabo para comprimir 4 archivos en uno solo? Indicar la secuencia de comandos ejecutados.*

Las acciones que se deben llevar a cabo para comprimir 4 archivos en uno solo son:

```
cd /home/jmenduina
touch file1.txt file2.txt file3.txt file4.txt
echo "Archivo 1" > file1.txt
echo "Archivo 2" > file2.txt
echo "Archivo 3" > file3.txt
echo "Archivo 4" > file4.txt
ls -l file*.txt
du -ch file*.txt | grep total
tar -cvf archivos.tar file*.txt
ls -l archivos.tar
tar -czvf archivos.tar.gz file*.txt
ls -l archivos.tar.gz
```

(d) *¿Pueden comprimirse un conjunto de archivos utilizando un único comando?*

Sí, un conjunto de archivos pueden comprimirse utilizando un único comando:

tar -czvf archivos.tar.gz file.txt*.

(e) *Investigar la funcionalidad de los siguientes comandos:*

- *tar*: Sirve para empaquetar (agrupar) varios archivos o directorios en uno solo, sin comprimirlos necesariamente. También puede desempaquetar o extraer archivos desde un paquete *.tar* o *.tar.gz*.
- *grep*: Busca texto o patrones dentro de archivos. Es muy usado para filtrar información en la terminal.
- *gzip*: Comprime archivos individuales usando el algoritmo *gzip* (GNU *zip*). Por defecto, reemplaza el archivo original por su versión comprimida *.gz*.
- *zgrep*: Permite buscar texto dentro de archivos comprimidos con *gzip* (*.gz*) sin tener que descomprimirlos primero. Funciona igual que *grep*, pero sobre archivos *.gz*.
- *wc*: Cuenta líneas, palabras y caracteres de un archivo o entrada estándar.

Ejercicio 19.

Indicar qué acción realiza cada uno de los comandos indicados a continuación, considerando su orden. Suponer que se ejecutan desde un usuario que no es root ni pertenece al grupo de root. (Asumir que se encuentra posicionado en el directorio de trabajo del usuario con el que se logueó). En caso de no poder ejecutarse el comando, indicar la razón:

1. *ls -l > prueba*:
   Lista el contenido detallado del directorio actual y lo guarda en el archivo *prueba* (creándolo o sobreescribiéndolo).
2. *ps > PRUEBA*:
   Guarda la salida del comando *ps* (procesos del usuario actual) en el archivo *PRUEBA*.
3. *chmod 710 prueba*:
   Establece permisos *rwx--x---* (propietario: lectura/escritura/ejecución, grupo: ejecución, otros: ninguno) sobre el archivo *prueba*.
4. *chown root:root PRUEBA*:
   Intenta cambiar propietario y grupo de *PRUEBA* a *root:root*. No se puede ejecutar desde un usuario normal, ya que sólo *root* puede cambiar la propiedad a otro usuario.
5. *chmod 777 PRUEBA*:
   Establece permisos *rwxrwxrwx* (todos pueden leer/escribir/ejecutar) sobre el archivo *PRUEBA*.
6. *chmod 700 /etc/passwd*:
   Intenta establecer permisos *rwx------* sobre */etc/passwd* (propietario: lectura/escritura/ejecución, grupo: ninguno, otros: ninguno). No se puede ejecutar desde un usuario normal, ya que sólo *root* puede cambiar los permisos del archivo */etc/passwd*.
7. *passwd root*:
   Intenta cambiar la contraseña del usuario *root*. No se puede ejecutar desde un usuario normal, ya que sólo *root* puede cambiar otra cuenta.
8. *rm PRUEBA*:
   Borra el archivo *PRUEBA*.
9. *man /etc/shadow*:
   Intenta localizar la página del manual */etc/shadow*. No se puede ejecutar desde un usuario normal, ya que sólo *root* puede acceder a él.
10. *find / -name *.conf*:
    Busca, recursivamente, desde el directorio raíz (*/*) archivos cuyo nombre coincida con la máscara **.conf*. Se puede ejecutar desde un usuario normal, pero el usuario no tiene permiso de lectura/ejecución en todos los directorios del sistema, por lo que *find* producirá muchos mensajes *"Permission denied"* en rutas restringidas. Aun así, encontrará y listará los archivos dentro de rutas donde el usuario sí tiene permiso de lectura/ejecución (por ejemplo, dentro de su *home* o áreas públicas).
11. *usermod root -d /home/ newroot -L*:
    Intenta modificar la cuenta *root* (nombre y/o directorio *home*). No se puede ejecutar desde un usuario normal, ya que sólo *root* puede cambiar otra cuenta.
12. *cd /root*:

Intenta cambiar el directorio actual al directorio */root* (*home* de *root*). No se puede ejecutar desde un usuario normal, ya que sólo *root* puede acceder a su directorio.

13. *rm **:
    Intenta borrar todos los archivos del directorio actual (según la expansión del *). Se puede ejecutar desde un usuario normal, para los archivos sobre los que éste tenga permiso de escritura.
14. *cd /etc*:
    Cambia el directorio actual al directorio */etc*.
15. *cp * /home -R*:
    Intenta copiar, recursivamente, todos los elementos del directorio actual al directorio */home*. No se puede ejecutar desde un usuario normal, ya que sólo *root* puede escribir en el directorio */home*.
16. *shutdown*:
    Ordena el apagado/reinicio del sistema. No se puede ejecutar desde un usuario normal.

Ejercicio 20.

Indicar qué comando sería necesario ejecutar para realizar cada una de las siguientes acciones:

(a) *Terminar el proceso con PID 23.*

kill 23

(b) *Terminar el proceso llamado init o systemd. ¿Qué resultados se obtuvieron?*

kill 1

No es posible terminar este proceso, ya que *systemd/init* mantiene el sistema en funcionamiento (arranca servicios, monta discos, gestiona sesiones y coordina el apagado). Matarlo equivaldría a "apagar el corazón del sistema operativo".

(c) *Buscar todos los archivos de usuarios en los que su nombre contiene la cadena ".conf".*

find / -name "∗.conf"

(d) *Guardar una lista de procesos en ejecución en el archivo* ***/home/<su nombre de usuario>/procesos****.*

ps aux > /home/jmenduina/procesos
cat /home/jmenduina/procesos

(e) *Cambiar los permisos del archivo* ***/home/<su nombre de usuario>/xxxx*** *a:*

- *Usuario: Lectura, escritura, ejecución.*
- *Grupo: Lectura, ejecución.*
- *Otros: Ejecución.*

chmod 751 /home/jmenduina/procesos
ls -l /home/jmenduina/procesos

(f) *Cambiar los permisos del archivo* ***/home/<su nombre de usuario>/yyyy*** *a:*

- *Usuario: Lectura, escritura.*

- *Grupo: Lectura, ejecución.*
- *Otros: Ninguno.*

chmod 650 /home/jmenduina/procesos
ls -l /home/jmenduina/procesos

(g) *Borrar todos los archivos del directorio **/tmp**.*

*rm -f /tmp/**
ls /tmp

(h) *Cambiar el propietario del archivo **/opt/isodata** al usuario **isocso**.*

touch /opt/isodata
chown isocso /opt/isodata
ls -l /opt/isodata

(i) *Guardar, en el archivo **/home/<su nombre de usuario>/donde**, el directorio donde me encuentro en este momento. En caso de que el archivo exista, no se debe eliminar su contenido anterior.*

pwd >> /home/jmenduina/directorio
cat /home/jmenduina/directorio

Ejercicio 21.

Indicar qué comando sería necesario ejecutar para realizar cada una de las siguientes acciones:

(a) *Ingresar al sistema como usuario "root".*

su -

(b) *Crear un usuario. Elegir, como nombre, por convención, la primera letra del nombre seguida del apellido. Asignarle una contraseña de acceso.*

useradd -m jmenduina
passwd jmenduina

(c) *¿Qué archivos fueron modificados luego de crear el usuario y qué directorios se crearon?*

Luego de crear el usuario, los archivos que fueron modificados fueron: */etc/passwd*, */etc/shadow*, */etc/group*, */etc/gshadow*. Y el directorio que se creó fue */home/jmenduina*, con sus archivos ocultos de configuración: */.bash_logout*, */.bashrc* y */.profile*.

(d) *Crear un directorio en /tmp llamado miCursada.*

mkdir /tmp/miCursada
cd /tmp/miCursada

(e) *Copiar todos los archivos de /var/log al directorio antes creado.*

cp -r /var/log/ /tmp/miCursada*
ls /tmp/miCursada

(f) *Para el directorio antes creado (y los archivos y subdirectorios contenidos en él), cambiar el propietario y grupo al usuario creado y grupo users*.

chown -r jmenduina:users /tmp/miCursada
ls -l /tmp/miCursada

(g) *Agregar permiso total al dueño, de escritura al grupo y escritura y ejecución a todos los demás usuarios para todos los archivos dentro de un directorio en forma recursiva.*

chmod -r 723 /tmp/miCursada
ls -l /tmp/miCursada

(h) *Acceder a otra terminal para loguearse con el usuario antes creado.*

su jmenduina

(i) *Una vez logueado con el usuario antes creado, averiguar cuál es el nombre de la terminal.*

tty

(j) *Verificar la cantidad de procesos activos que hay en el sistema.*

ps aux | wc -l

(k) *Verificar la cantidad de usuarios conectados al sistema.*

who | wc -l

(l) *Volver a la terminal del usuario root y enviarle un mensaje al usuario anteriormente creado, enviándole que el sistema va a ser apagado.*

write jmenduina
El sistema va a ser apagado.

(m) *Apagar el sistema.*

shutdown -h now

Ejercicio 22.

Indicar qué comando sería necesario ejecutar para realizar cada una de las siguientes acciones:

(a) *Crear un directorio cuyo nombre sea el número de legajo e ingresar a él.*

mkdir /home/jmenduina/24981
cd /home/jmenduina/24981

(b) *Crear un archivo utilizando el editor de textos vi e introducir información personal: Nombre, Apellido, Número de alumno y dirección de correo electrónico. El archivo debe llamarse "LEAME".*

vi /home/jmenduina/249819/LEAME.txt
Juan Ignacio Menduiña, 24981/9, menduinajuan@gmail.com
:exit
cat /home/jmenduina/249819/LEAME.txt

(c) *Cambiar los permisos del archivo "LEAME", de manera que se puedan ver reflejados los siguientes permisos:*

- *Dueño: Ningún permiso.*
- *Grupo: Permiso de ejecución.*
- *Otros: Todos los permisos.*

chmod 017 /home/jmenduina/249819/LEAME.txt
ls -l /home/jmenduina/249819/LEAME.txt

(d) *Ir al directorio /etc y verificar su contenido. Crear un archivo dentro del directorio personal cuyo nombre sea "leame", donde el contenido del mismo sea el listado de todos los archivos y directorios contenidos en /etc. ¿Cuál es la razón por la cual se puede crear este archivo si ya existe un archivo llamado "LEAME" en este directorio?*

cd /etc
ls
ls > /home/jmenduina/249819/leame.txt
cat /home/jmenduina/249819/leame.txt

La razón por la cual se puede crear este archivo "leame.txt" si ya existe un archivo llamado "LEAME.txt" es que ambos nombres son distintos, ya que, en los sistemas Linux (y, en general, en todos los sistemas de archivos *Unix-Like*), los nombres de archivos distinguen entre mayúsculas y minúsculas.

(e) *¿Qué comando se utilizaría y de qué manera si se tuviera que localizar un archivo dentro del file system? ¿Y si se tuvieran que localizar varios archivos con características similares? Explicar el concepto teórico y ejemplificar.*

Si se tuviera que localizar un archivo dentro del *file system*, el comando que se utilizaría sería *find*:

find / -name "archivo.txt", que busca, en todo el sistema (*/*), un archivo llamado, exactamente, *archivo.txt*.

Y, si se tuvieran que localizar varios archivos con características similares, el comando que se utilizaría también sería *find*:

find / -name ".conf"*, que busca, en todo el sistema (*/*), todos los archivos terminados en *.conf*.

(f) *Utilizando los conceptos aprendidos en el inciso anterior, buscar todos los archivos cuya extensión sea .so y almacenar el resultado de esta búsqueda en un archivo dentro del directorio creado en el primer inciso. El archivo deberá llamarse ejercicioF*.

```
find / -type f -name "*.so" > /home/jmenduina/249819/ejercicioF
cat /home/jmenduina/249819/ejercicioF
```

Ejercicio 23.

Indicar qué acción realiza cada uno de los comandos indicados a continuación, considerando su orden. Suponer que se ejecutan desde un usuario que no es root ni pertenece al grupo de root. (Asumir que se encuentra posicionado en el directorio de trabajo del usuario con el que se logueó). En caso de no poder ejecutarse el comando, indicar la razón:

(a) *Iniciar 2 sesiones utilizando nombre de usuario y contraseña. En una sesión, ir siguiendo, paso a paso, las órdenes que se encuentran escritas en el cuadro superior. En la otra sesión, crear, utilizando algún editor de textos, un archivo que se llame "explicacion_de_ejercicio" dentro del directorio creado en el ejercicio 22 y, para cada uno de los comandos que se ejecute en la otra sesión, realizar una breve explicación de los resultados obtenidos.*

(b) *Completar los comandos 19 y 20, de manera tal que realicen la siguiente acción:*

- *19: Copiar el directorio iso y todo su contenido al directorio creado en el ejercicio 23.a.*
- *20: Copiar el resto de los archivos y directorios que se crearon en este ejercicio al directorio creado en el ejercicio 23.a.*

(c) *Ejecutar las órdenes 19 y 20 y comentarlas en el archivo creado en el inciso (a).*

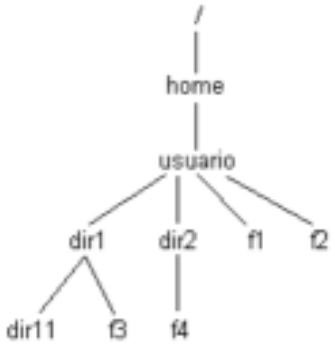
1. *mkdir iso*:
   Crea, en el directorio actual (*/home*), un directorio llamado *iso*.
2. *cd ./iso; ps > f0*:
   Cambia el directorio actual al directorio */home/iso*. Y, además, guarda la salida del comando *ps* (procesos del usuario actual) en el archivo *f0* (creándolo o sobreescribiéndolo).
3. *ls > f1*:
   Lista el contenido del directorio *iso* y lo guarda en el archivo *f1* (creándolo o sobreescribiéndolo).
4. *cd /*:
   Cambia el directorio actual al directorio raíz (*/*).
5. *echo $HOME*:
   Muestra la ruta del directorio *home* del usuario actual.
6. *ls -l > $HOME/iso/ls*:
   Lista el contenido detallado del directorio *iso* y lo guarda en el archivo *ls* (creándolo o sobreescribiéndolo).
7. *cd $HOME; mkdir f2*:
   Cambia el directorio actual al directorio *home* del usuario y crea, allí, un directorio llamado *f2*.
8. *ls −ld f2*:
   Lista el contenido detallado del directorio *f2*.
9. *chmod 341 f2*:
   Establece permisos *-wxr----x* (propietario: escritura/ejecución, grupo: escritura, otros: ejecución) sobre el directorio *f2*.
10. *touch dir*:

Crea un archivo llamado *dir* en el directorio *home*.

11. *cd f2*:
    Cambia el directorio actual al directorio *f2*.
12. *cd ~/iso*:
    Cambia el directorio actual al directorio */home/iso*.
13. *pwd > f3*:
    Guarda la ruta completa del directorio actual (*/home/iso*) en el archivo *f3* (creándolo o sobreescribiéndolo).
14. *ps | grep 'ps' | wc -l >> ../f2/f3*:
    Cuenta cuántos procesos *ps* están corriendo y guarda el resultado al final del archivo *f3* ubicado en el directorio */home/f2* (si no existe, lo crea). .
15. *chmod 700 ../f2 ; cd ..*:
    Establece permisos *rwx*------ (propietario: lectura/escritura/ejecución, grupo: ninguno, otros: ninguno) sobre el directorio *f2*. Y, además, cambia el directorio actual al directorio (*/home*).
16. *find . -name etc/passwd*:
    Intenta buscar, recursivamente, desde el directorio actual (*/home*), pero el parámetro *-name* en *find* sólo busca por el nombre de un archivo, no por rutas completas.
17. *find / -name etc/passwd*:
    Intenta buscar, recursivamente, desde el directorio raíz (*/*), pero el parámetro *-name* en *find* sólo busca por el nombre de un archivo, no por rutas completas.
18. *mkdir ejercicio5*:
    Crea, en el directorio actual (*/home*), un directorio llamado *ejercicio5*.
19. *cp -r /home/iso /home/ejercicio5.*
20. *cp -r /home/f2 /home/dir /home/ejercicio5.*

Ejercicio 24.

Crear una estructura desde el directorio /home que incluya varios directorios, subdirectorios y archivos, según el esquema siguiente:



Creación de estructura:

mkdir /home/jmenduina/dir1
mkdir /home/jmenduina/dir2
touch /home/jmenduina/f1 /home/jmenduina/f2
mkdir /home/jmenduina/dir1/dir11
touch /home/jmenduina/dir1/f3
touch /home/jmenduina/dir2/f4

Asumir que "usuario" indica cuál es el nombre de usuario. Además, se deberá tener en cuenta que dirX hace referencia a directorios y fX hace referencia a archivos. Utilizando la estructura de directorios anteriormente creada, indicar qué comandos son necesarios para realizar las siguientes acciones:

(a) *Mover el archivo "f3" al directorio de trabajo /home/usuario.*

mv /home/jmenduina/dir1/f3 /home/jmenduina
ls /home/jmenduina

(b) *Copiar el archivo "f4" en el directorio "dir11".*

cp /home/jmenduina/dir2/f4 /home/jmenduina/dir1/dir11
ls /home/jmenduina/dir1/dir11

(c) *Hacer lo mismo que en el inciso anterior, pero el archivo de destino se debe llamar "f7".*

cp /home/jmenduina/dir2/f4 /home/jmenduina/dir1/dir11/f7
ls /home/jmenduina/dir1/dir11

(d) *Crear el directorio "copia" dentro del directorio usuario y copiar, en él, el contenido de "dir1".*

mkdir /home/jmenduina/copia
cp -r /home/jmenduina/dir1/ /home/jmenduina/copia*
ls /home/jmenduina/copia

(e) *Renombrar el archivo "f1" por el nombre "archivo" y ver los permisos del mismo.*

mv /home/jmenduina/f1 /home/jmenduina/archivo
ls -l /home/jmenduina

(f) *Cambiar los permisos del archivo llamado "archivo", de manera de reflejar lo siguiente:*

- *Usuario: Permisos de lectura y escritura.*
- *Grupo: Permisos de ejecución.*
- *Otros: Todos los permisos.*

chmod 617 /home/jmenduina/archivo
ls -l /home/jmenduina/archivo

(g) *Renombrar los archivos "f3" y "f4" de manera que se llamen "f3.exe" y "f4.exe", respectivamente.*

mv /home/jmenduina/f3 /home/jmenduina/f3.exe
mv /home/jmenduina/dir2/f4 /home/jmenduina/dir2/f4.exe
ls /home/jmenduina /home/jmenduina/dir2

(h) *Utilizando un único comando, cambiar los permisos de los dos archivos renombrados en el inciso anterior, de manera de reflejar lo siguiente:*

- *Usuario: Ningún permiso.*
- *Grupo: Permisos de escritura.*
- *Otros: Permisos de escritura y ejecución.*

chmod 023 /home/jmenduina/f3.exe /home/jmenduina/dir2/f4.exe
ls -l /home/jmenduina/f3.exe /home/jmenduina/dir2/f4.exe

Ejercicio 25.

Indicar qué comando/s es/son necesario/s para realizar cada una de las acciones de la siguiente secuencia de pasos (considerando su orden de aparición):

(a) *Crear un directorio llamado logs en el directorio /tmp.*

```
mkdir /tmp/logs
cd /tmp/logs
```

(b) *Copiar todo el contenido del directorio /var/log en el directorio creado en el inciso anterior.*

```
cp -r /var/log/* /tmp/logs
ls /tmp/logs
```

(c) *Empaquetar el directorio creado en (a); el archivo resultante se debe llamar "misLogs.tar".*

```
tar -cvf /tmp/misLogs.tar -C /tmp logs
ls /tmp
```

(d) *Empaquetar y comprimir el directorio creado en (a); el archivo resultante se debe llamar "misLogs.tar.gz".*

```
tar -czvf /tmp/misLogs.tar.gz -C /tmp logs
ls /tmp
```

(e) *Copiar los archivos creados en (c) y (d) al directorio de trabajo del usuario.*

```
cp /tmp/misLogs.tar /tmp/misLogs.tar.gz /home/jmenduina
ls /home/jmenduina
```

(f) *Eliminar el directorio creado en (a), logs.*

```
rm -rf /tmp/logs
ls /tmp/logs
```

(g) *Desempaquetar los archivos creados en (c) y (d) en 2 directorios diferentes.*

```
mkdir /tmp/desempaquetado_tar
mkdir /tmp/desempaquetado_targz
tar -xvf /home/jmenduina/misLogs.tar -C /tmp/desempaquetado_tar
tar -xzvf /home/jmenduina/misLogs.tar.gz -C /tmp/desempaquetado_targz
ls /tmp/desempaquetado_tar /tmp/desempaquetado_targz
```

Juan Menduiña

Trabajo Práctico N° 2:

Definición de SO. Componentes de un SO. Apoyo del *Hardware*: Modos de Ejecución, Interrupciones. Llamadas al Sistema. Programa y Proceso. Planificación de CPU. Colas de Planificación. *Context Switch*. Creación de Procesos. Espacio de Direcciones. Direcciones Lógicas y Físicas. Particiones Fijas y Dinámicas. Paginación y Segmentación. Fragmentación. MMU. Algoritmos de Administración.

Ejercicio 1.

Ejercicio 2.

Ejercicio 3.

Ejercicio 4.

Ejercicio 5.

Ejercicio 6.

Ejercicio 7.

Ejercicio 8.

Ejercicio 9.

Ejercicio 10.

Ejercicio 11.

Ejercicio 12.

Ejercicio 13.

Ejercicio 14.

Ejercicio 15.

Ejercicio 16.

Ejercicio 17.

Ejercicio 18.

Ejercicio 19.

Ejercicio 20.

Ejercicio 21.

Ejercicio 22.

Ejercicio 23.

Ejercicio 24.

Ejercicio 25.

Ejercicio 26.

Ejercicio 27.

Ejercicio 28.

Ejercicio 29.

Ejercicio 30.

Ejercicio 31.

Ejercicio 32.

Ejercicio 33.

Ejercicio 34.

Ejercicio 35.

Ejercicio 36.

Ejercicio 37.

Ejercicio 38.

Juan Menduiña

Trabajo Práctico N° 3:
Bash. *Script*. Sintaxis. GNU/Linux.

Ejercicio 1.

¿Qué es el Shell Scripting? ¿A qué tipos de tareas están orientados los script? ¿Los scripts deben compilarse? ¿Por qué?

El *Shell Scripting* es el proceso de escribir programas o secuencias de comandos (*scripts*) que se ejecutan en un intérprete de comandos del sistema operativo, como Bash, Zsh, Sh, Ksh, etc. Un *shell script* es, básicamente, un archivo de texto con instrucciones que el sistema interpreta línea por línea, igual que si se escribieran, manualmente, en la terminal. Generalmente, su extensión es .sh.

Los *scripts* de *shell* están orientados, principalmente, a tareas repetitivas o administrativas en sistemas GNU/Linux (aunque también en Windows con PowerShell).

Los *scripts* no deben compilarse, se interpretan. Esto significa que no se traducen a código máquina antes de ejecutarse, como ocurre con los programas en C o Java. En cambio, el intérprete del *shell* lee y ejecuta cada línea del *script*, directamente, en tiempo de ejecución.

No se compilan porque el *shell* es un intérprete, no un compilador. Su diseño busca flexibilidad y rapidez en la escritura y la ejecución, no velocidad de ejecución máxima. La idea es poder modificar y probar *scripts* fácilmente, sin necesidad de un proceso de compilación intermedio.

Ejercicio 2.

(a) *Investigar la funcionalidad de los comandos echo y read.*

El comando *echo* muestra texto o el valor de variables por pantalla (en la terminal). Es útil para dar mensajes al usuario o mostrar resultados. Ejemplo:

```
echo "Hola mundo"
echo "Tu nombre es $nombre"
```

El comando *read* permite leer datos ingresados por el usuario desde el teclado y guardarlos en una variable. Se usa para hacer *scripts* interactivos. Ejemplo:

```
echo "Ingresar nombre:"
read nombre
echo "Hola $nombre, ¡bienvenido!"
```

(b) *¿Cómo se indican los comentarios dentro de un script?*

Los comentarios en *shell script* se indican con el símbolo #. Todo lo que sigue después del # en esa línea no se ejecuta. Ejemplo:

```
# Éste es un comentario
echo "Hola mundo" # Esto también es un comentario al final de una línea
```

(c) *¿Cómo se declaran y se hace referencia a variables dentro de un script?*

Las variables se declaran sin espacios entre el nombre, el signo = y el valor. Ejemplo:

```
nombre="Juan"
edad=25
```

Para usar o mostrar el valor de una variable, se antepone el signo $ al nombre de la variable. Ejemplo:

```
echo "Mi nombre es $nombre y tengo $edad años."
```

Ejercicio 3.

Crear, dentro del directorio personal del usuario logueado, un directorio llamado practica-shell-script y, dentro de él, un archivo llamado mostrar.sh cuyo contenido sea el siguiente:

```
#!/bin/bash
# Comentarios acerca de lo que hace el script
# Siempre comento mis scripts, si no lo hago hoy,
# mañana ya no me acuerdo de lo que quise hacer
echo "Introduzca su nombre y apellido:"
read nombre apellido
echo "Fecha y hora actual:"
date
echo "Su apellido y nombre es:"
echo "$apellido $nombre"
echo "Su usuario es: `whoami`"
echo "Su directorio actual es:"
```

(a) *Asignar al archivo creado los permisos necesarios de manera que se pueda ejecutar.*

```
mkdir /home/jmenduina/practica-shell-script
cd /home/jmenduina/practica-shell-script
touch /home/jmenduina/practica-shell-script/mostrar.sh
ls -l /home/jmenduina/practica-shell-script

nano /home/jmenduina/practica-shell-script/mostrar.sh

chmod +x /home/jmenduina/practica-shell-script/mostrar.sh
ls -l /home/jmenduina/practica-shell-script
```

(b) *Ejecutar el archivo creado de la siguiente manera: ./mostrar.sh.*

```
./mostrar.sh
```

(c) *¿Qué resultado visualiza?*

El resultado que se visualiza es:

```
Introduzca su nombre y apellido:
Juan Menduiña
Fecha y hora actual:
Sun Nov 9 13:32:34 -03 2025
Su apellido y nombre es:
Menduiña Juan
```

Su usuario es: root
Su directorio actual es:

(d) *Las backquotes (`) entre el comando whoami ilustran el uso de la sustitución de comandos. ¿Qué significa esto?*

La sustitución de comandos en *shell* significa que el resultado (salida) de un comando se reemplaza, directamente, en la línea donde aparece. El *shell* ejecuta el comando que está entre las comillas invertidas o dentro de *$( )*, toma su salida y la sustituye en el lugar donde estaba el comando.

(e) *Realizar modificaciones al script anteriormente creado de manera de poder mostrar distintos resultados (cuál es su directorio personal, el contenido de un directorio en particular, el espacio libre en disco, etc.). Pedir que se introduzcan por teclado (entrada estándar) otros datos.*

Sólo agrego lo siguiente:

```
echo "Su directorio actual es:"
pwd
```

Ahora, el resultado que se visualiza es:

```
Introduzca su nombre y apellido:
Juan Menduiña
Fecha y hora actual:
Sun Nov 9 13:32:34 -03 2025
Su apellido y nombre es:
Menduiña Juan
Su usuario es: root
Su directorio actual es:
/home/jmenduina/practica-shell-script
```

Ejercicio 4.

Parametrización: ¿Cómo se acceden a los parámetros enviados al script al momento de su invocación? ¿Qué información contienen las variables $#, $, $? y $HOME dentro de un script?*

Cuando se ejecuta un *script* en Bash, se le pueden pasar parámetros o argumentos al invocarlo desde la línea de comandos, separados por espacios: *./mi_script.sh juan menduiña 30*. A los parámetros enviados al *script* al momento de su invocación se accede mediante variables especiales numeradas: *$0* (nombre del *script*), *$1* (primer parámetro), *$2* (segundo parámetro), …

Existen las siguientes variables especiales en Bash:

- *$#*: Cantidad de parámetros pasados al *script*.
- *$**: Todos los parámetros en una sola cadena (separados por espacios).
- *$?*: Código de salida del último comando ejecutado (0 = éxito, distinto de 0 = error).
- *$HOME*: Directorio personal del usuario actual.

Ejercicio 5.

¿Cuál es la funcionalidad del comando exit? ¿Qué valores recibe como parámetro y cuál es su significado?

La funcionalidad del comando *exit* es detener, inmediatamente, la ejecución del *script* y devuelve un valor numérico (el código de salida o *exit status*) al sistema.

El valor que recibe como parámetro *n* opcional numérico puede ser entre 0 y 255, que representa el estado de finalización del *script* (0 = éxito, distinto de 0 = error). Si no se especifica ningún número, Bash usa el código de salida del último comando ejecutado.

Ejercicio 6.

El comando expr permite la evaluación de expresiones. Su sintaxis es: expr arg1 op arg2, donde arg1 y arg2 representan argumentos y op la operación de la expresión. Investigar qué tipo de operaciones se pueden utilizar.

El comando *expr* (abreviatura de *expression*) sirve para evaluar expresiones y mostrar su resultado por pantalla. Se usa, principalmente, en *scripts* antiguos o cuando no se dispone de la expansión aritmética *$(( ))*.

El tipo de operaciones que se pueden utilizar son:

1. Operaciones aritméticas: + (suma), - (resta), * (multiplicación), / (división entera), % (módulo).
2. Operaciones relacionales o de comparación: = (igualdad de cadenas), *!=* (desigualdad de cadenas), > (mayor que), < (menor que), >= (mayor o igual que), <= (menor o igual que).
3. Operaciones lógicas: | (OR, o lógico), *&* (AND, y lógico).

Ejercicio 7.

El comando test expresion permite evaluar expresiones y generar un valor de retorno, true o false. Este comando puede ser reemplazado por el uso de corchetes de la siguiente manera: [expresión]. Investigar qué tipo de expresiones pueden ser usadas con el comando test. Tener en cuenta operaciones para: evaluación de archivos, evaluación de cadenas de caracteres y evaluaciones numéricas.

El comando *test* (y su forma equivalente con corchetes *[expresión]*) sirve para evaluar condiciones en un *script*. Su salida es un valor de retorno: 0 (verdadero), 1 (falso). Se usa, normalmente, en estructuras *if*, *while*, etc.

El tipo de expresiones que pueden ser usadas con el comando *test* son:

1. Evaluación de archivos: Sirven para comprobar si un archivo o directorio existe o cumple ciertas condiciones.
   Ejemplos:
   - *-e archivo* (verdadero si existe el archivo).
   - *-f archivo* (verdadero si existe y es un archivo regular).
   - *-d archivo* (verdadero si existe y es un directorio).
   - *-r archivo* (verdadero si el archivo es legible).
   - *-w archivo* (verdadero si el archivo es escribible).
   - *-x archivo* (verdadero si el archivo es ejecutable).
   - *-s archivo* (verdadero si el archivo no está vacío).
   - *archivo1 -nt archivo2* (verdadero si archivo1 es más nuevo que archivo2).
   - *archivo1 -ot archivo2* (verdadero si archivo1 es más viejo que archivo2).
2. Evaluación de cadenas de caracteres: Sirven para comparar o verificar cadenas.
   Ejemplos:
   - *-z cadena* (verdadero si la longitud es cero).
   - *-n cadena* (verdadero si la longitud no es cero).
   - *cadena1 = cadena2* (verdadero si son iguales).
   - *cadena1 != cadena2* (verdadero si son distintas).
3. Evaluaciones numéricas: Sirven para comparar valores enteros (no cadenas).
   Ejemplos:
   - *-eq* (igual).
   - *-ne* (distinto).
   - *-gt* (mayor que).
   - *-lt* (menor que).
   - *-ge* (mayor o igual que).
   - *-le* (menor o igual que).

Ejercicio 8.

Estructuras de control. Investigar la sintaxis de las siguientes estructuras de control incluídas en shell scripting:

(a) *if.*

Se usa para evaluar una condición y ejecutar comandos según sea verdadera o falsa.

```
if [ condición ]; then
   comandos_si_verdadero
elif [ otra_condición ]; then
   comandos_si_se_cumple_elif
else
   comandos_si_falso
fi
```

(b) *case.*

Se usa para evaluar una variable frente a varios posibles valores (como un *switch* en otros lenguajes).

```
case variable in
   valor1)
      comandos
      ;;
   valor2|valor3)
      otros_comandos
      ;;
   *)
      comandos_por_defecto
      ;;
esac
```

(c) *while.*

Ejecuta un bloque de comandos mientras se cumpla una condición.

```
while [ condición ]; do
   comandos
done
```

(d) *for.*

Permite recorrer una lista de elementos o un rango.

```
for variable in lista; do
   comandos
done
```

(e) *select.*

Se usa para crear menús interactivos (útil en *scripts* de usuario).

```
select variable in lista; do
   comandos
done
```

Ejercicio 9.

¿Qué acciones realizan las sentencias break y continue dentro de un bucle? ¿Qué parámetros reciben?

Las acciones que realizan las sentencias *break* y *continue* dentro de un bucle y los parámetros que reciben son:

- *break*: El programa sale del ciclo y continúa ejecutando las instrucciones que siguen después del bucle.
- *continue*: Salta a la siguiente iteración del bucle, sin ejecutar las instrucciones restantes de la iteración actual.

Ambas sentencias pueden recibir un parámetro *n* opcional numérico, que indica cuántos niveles de bucles anidados deben afectarse:

- *break*: Sale del bucle actual (por defecto).
- *break n*: Sale de *n* niveles de bucles anidados.
- *continue*: Salta a la siguiente iteración del bucle actual (por defecto).
- *continue n*: Salta a la siguiente iteración del n-ésimo bucle exterior.

Ejercicio 10.

¿Qué tipo de variables existen? ¿Es shell script fuertemente tipado? ¿Se pueden definir arreglos? ¿Cómo?

Existen los siguientes tipo de variables:

1. Variables del usuario (o locales): Son las que se definen dentro del *script*. Sólo existen mientras el *script* se está ejecutando.
2. Variables de entorno: Son variables del sistema o del entorno del usuario. Están disponibles para todos los procesos y los programas.

Shell Script no es fuertemente tipado, lo cual significa que todas las variables se manejan como cadenas de texto, aunque contengan números.

Se pueden definir arreglos en Bash (no en todos los *shell* antiguos). Ejemplo: *numeros=(10 20 30 40)*.

Ejercicio 11.

¿Pueden definirse funciones dentro de un script? ¿Cómo? ¿Cómo se maneja el pasaje de parámetros de una función a la otra?

Sí, dentro de un *script*, pueden definirse funciones. Hay dos formas válidas de definir funciones:

```
nombre_funcion() {
   comandos
}

function nombre_funcion {
   comandos
}
```

El pasaje de parámetros de una función a la otra se maneja igual que en un *script*, se pasan al invocar la función, separados por espacios. Dentro de la función, se accede a ellos con las variables posicionales: *$1* (primer parámetro), *$2* (segundo parámetro), …

Bash no devuelve valores "numéricos" directamente, como otros lenguajes. Hay dos formas de devolver resultados:

1. Usando *echo*: Permite capturar el valor en una variable.
2. Usando *return*: Sólo para códigos de estado 0-255.

Ejercicio 12: Evaluación de Expresiones.

(a) *Realizar un script que le solicite al usuario 2 números, los lea de la entrada Standard e imprima la multiplicación, suma, resta y cuál es el mayor de los números leídos.*

```
#!/bin/bash
# Script: Ejercicio12a.sh
# Uso: ./Ejercicio12a.sh

# Solicitud de números al usuario
echo -n "Introducir primer número: "
read num1
echo -n "Introducir segundo número: "
read num2

# Operaciones aritméticas
suma=$((num1 + num2))
resta=$((num1 - num2))
multiplicacion=$((num1 * num2))

# ¿Cuál es el mayor de los números leídos?
if [ $num1 -gt $num2 ]; then
   mayor=$num1
elif [ $num1 -lt $num2 ]; then
   mayor=$num2
else
   mayor="Son iguales"
fi

# Resultados
echo "------------------------------"
echo "RESULTADOS:"
echo " - La suma es: $suma"
echo " - La resta es: $resta"
echo " - La multiplicación es: $multiplicacion"
echo " - El mayor es: $mayor"
echo "------------------------------"
```

(b) *Modificar el script creado en el inciso anterior para que los números sean recibidos como parámetros. El script debe controlar que los dos parámetros sean enviados.*

```
#!/bin/bash
# Script: Ejercicio12b.sh
# Uso: ./Ejercicio12b.sh <num1> <num2>

# Control de cantidad de parámetros
if [ $# -ne 2 ]; then
```

```
    echo "Error: Se deben ingresar, exactamente, 2 parámetros"
    echo "Uso: $0 <num1> <num2>"
    exit 1
fi

# Asignación de parámetros
num1=$1
num2=$2

# Operaciones aritméticas
suma=$((num1 + num2))
resta=$((num1 - num2))
multiplicacion=$((num1 * num2))

# ¿Cuál es el mayor de los números leídos?
if [ $num1 -gt $num2 ]; then
    mayor=$num1
elif [ $num1 -lt $num2 ]; then
    mayor=$num2
else
    mayor="Son iguales"
fi

# Resultados
echo "------------------------------"
echo "RESULTADOS:"
echo " - La suma es: $suma"
echo " - La resta es: $resta"
echo " - La multiplicación es: $multiplicacion"
echo " - El mayor es: $mayor"
echo "------------------------------"
```

(c) *Realizar una calculadora que ejecute las 4 operaciones básicas: +, -, *, %. Esta calculadora debe funcionar recibiendo la operación y los números como parámetros.*

```
#!/bin/bash
# Script: Ejercicio12c.sh
# Uso: ./Ejercicio12c.sh <operación> <num1> <num2>

# Control de cantidad de parámetros
if [ $# -ne 3 ]; then
    echo "Error: Se deben ingresar, exactamente, 3 parámetros"
    echo "Uso: $0 <operación> <num1> <num2>"
    echo "Operaciones válidas: + - * %"
    exit 1
fi

# Asignación de parámetros
```

```
operacion=$1
num1=$2
num2=$3

# Validación de números
if ! [[ $num1 =~ ^-?[0-9]+$ && $num2 =~ ^-?[0-9]+$ ]]; then
    echo "Error: Los operandos deben ser números enteros"
    exit 1
fi

# Calculadora
case $operacion in
    "+")
        resultado=$((num1 + num2))
        ;;
    "-")
        resultado=$((num1 - num2))
        ;;
    "*")
        resultado=$((num1 * num2))
        ;;
    "%")
        if [ $num2 -eq 0 ]; then
            echo "Error: No se puede dividir por cero"
            exit 1
        fi
        resultado=$((num1 % num2))
        ;;
    *)
        echo "Operación inválida. Se debe usar: + - * %"
        exit 1
        ;;
esac
echo "Resultado: $resultado"
```

Ejercicio 13: Uso de las Estructuras de Control.

(a) *Realizar un script que visualice por pantalla los números del 1 al 100, así como sus cuadrados.*

```
#!/bin/bash
# Script: Ejercicio13a.sh
# Uso: ./Ejercicio13a.sh

# Algoritmo
for ((i = 1; i <= 100; i++)); do
   echo "Número: $i - Cuadrado: $((i * i))"
done
```

(b) *Crear un script que muestre 3 opciones al usuario: Listar, DondeEstoy y QuienEsta. Según la opción elegida, se le debe mostrar:*

- *Listar: Lista el contenido del directorio actual.*
- *DondeEstoy: Muestra la ruta del directorio donde me encuentro ubicado.*
- *QuienEsta: Muestra los usuarios conectados al sistema.*

```
#!/bin/bash
# Script: Ejercicio13b.sh
# Uso: ./Ejercicio13b.sh

# Selección de opción por parte del usuario
echo "Seleccionar una opción:"
echo "1) Listar"
echo "2) DondeEstoy"
echo "3) QuienEsta"
read -p "Opción: " opcion

# Algoritmo
case $opcion in
   1|Listar|listar)
      echo "Contenido del directorio actual:"
      ls
      ;;
   2|DondeEstoy|dondeestoy)
      echo "Ruta actual:"
      pwd
      ;;
   3|QuienEsta|quienesta)
      echo "Usuarios conectados:"
      who
      ;;
   *)
```

```
        echo "Opción inválida"
        ;;
esac
```

(c) *Crear un script que reciba como parámetro el nombre de un archivo e informe si el mismo existe o no y, en caso afirmativo, indique si es un directorio o un archivo. En caso de que no exista el archivo/directorio, cree un directorio con el nombre recibido como parámetro.*

```
#!/bin/bash
# Script: Ejercicio13c.sh
# Uso: ./Ejercicio13c.sh

# Control de cantidad de parámetros
if [ $# -ne 1 ]; then
   echo "Error: Se debe ingresar, exactamente, 1 parámetro"
   echo "Uso: $0 <nombre_archivo>"
   exit 1
fi

# Asignación de parámetros
nombre=$1

# Algoritmo
if [ -e "$nombre" ]; then
   if [ -d "$nombre" ]; then
      echo "'$nombre' existe y es un directorio"
   elif [ -f "$nombre" ]; then
      echo "'$nombre' existe y es un archivo regular"
   else
      echo "'$nombre' existe, pero no es ni archivo ni directorio"
   fi
else
   echo "'$nombre' no existe. Se creará un directorio con ese nombre..."
   mkdir "$nombre"
   echo "Directorio '$nombre' creado correctamente"
fi
```

Ejercicio 14: Renombrando Archivos.

Hacer un script que renombre sólo archivos de un directorio pasado como parámetro, agregándole una CADENA, contemplando las opciones:

- *"-a CADENA": Renombra el fichero concatenando CADENA al final del nombre del archivo.*
- *"-b CADENA": Renombra el fichero concatenando CADENA al comienzo del nombre del archivo.*

Ejemplos: Si se tienen los siguientes archivos /tmp/a y /tmp/b, al ejecutar ./renombra /tmp/ -a EJ, se obtendrá, como resultado, /tmp/aEJ y /tmp/bEJ. Y, si se ejecuta ./renombra /tmp/ -b EJ, el resultado será /tmp/EJa /tmp/EJb.

```
#!/bin/bash
# Script: Ejercicio14.sh
# Uso: ./Ejercicio14.sh

# Control de cantidad de parámetros
if [ $# -ne 3 ]; then
   echo "Error: Se deben ingresar, exactamente, 3 parámetros"
   echo "Uso: $0 <directorio> -a|-b <CADENA>"
   exit 1
fi

# Asignación de parámetros
directorio=$1
opcion=$2
cadena=$3

# Verificación de que el directorio exista
if [ ! -d "$directorio" ]; then
   echo "Error: '$directorio' no es un directorio válido"
   exit 1
fi

# Algoritmo
case $opcion in
   -a)
      for archivo in "$directorio"/*; do
         if [ -f "$archivo" ]; then
            nombre=$(basename "$archivo")
            mv "$archivo" "$directorio/${nombre}${cadena}"
         fi
      done
      echo "Archivos renombrados agregando '$cadena' al final"
      ;;
   -b)
      for archivo in "$directorio"/*; do
```

```
        if [ -f "$archivo" ]; then
            nombre=$(basename "$archivo")
            mv "$archivo" "$directorio/${cadena}${nombre}"
        fi
    done
    echo "Archivos renombrados agregando '$cadena' al comienzo"
    ;;
  *)
    echo "Opción inválida. Se debe usar: -a o -b"
    exit 1
    ;;
esac
```

Ejercicio 15.

El comando cut nos permite procesar las líneas de la entrada que reciba (archivo, entrada estándar, resultado de otro comando, etc.) y cortar columnas o campos, siendo posible indicar cuál es el delimitador de las mismas. Investigar los parámetros que puede recibir este comando y citar ejemplos de uso.

El comando *cut* en *Shell* sirve para extraer secciones específicas de cada línea de texto en un archivo o entrada estándar (*stdin*). Se puede pensar como una forma de "recortar columnas" o "campos" de texto según su posición o delimitador.

La sintaxis general es: *cut [opciones] [archivo]*. Si no se indica archivo, *cut* lee de la entrada estándar (por ejemplo, se puede usar con *pipe*).

Los parámetros que puede recibir este comando son:

- *-b LISTA*: Selecciona bytes específicos de cada línea (por posición).
- *-c LISTA*: Selecciona caracteres específicos de cada línea.
- *-f LISTA*: Selecciona campos (*fields*), separados por un delimitador.
- *-d 'DELIM'*: Define el delimitador de campos (por defecto es tabulación).
- *--complement*: Muestra todo menos los bytes/campos/caracteres indicados.
- *-s*: Suprime líneas que no contienen el delimitador (útil con *-f*).

Ejemplos de uso:

- *cut -b 1-3 archivo.txt*: Muestra los primeros 3 bytes (útil para datos binarios o sin acentos).
- *cut -c 1-5 archivo.txt*: Muestra los primeros 5 caracteres de cada línea del archivo.
- *cut -d ',' -f 1 datos.csv*: Muestra sólo la primera columna (nombre).
- *cut -d ',' -f 1 --complement datos.csv*: Muestra todos los campos excepto el primero.

Ejercicio 16.

Realizar un script que reciba como parámetro una extensión y haga un reporte con 2 columnas, el nombre de usuario y la cantidad de archivos que posee con esa extensión. Se debe guardar el resultado en un archivo llamado reporte.txt.

```
#!/bin/bash
# Script: Ejercicio16.sh
# Uso: ./Ejercicio16.sh

# Control de cantidad de parámetros
if [ $# -ne 1 ]; then
   echo "Error: Se debe ingresar, exactamente, 1 parámetro"
   echo "Uso: $0 <extension>"
   exit 1
fi

# Asignación de parámetros
extension=$1
salida="reporte.txt"

# Limpieza del archivo de salida
> "$salida"

# Algoritmo
echo "Nombre de usuario | Cantidad de archivos con extensión .$extension" >> "$salida"
echo "------------------------------------------------------------" >> "$salida"
for dir in /home/*; do
   if [ -d "$dir" ]; then
      usuario=$(basename "$dir")
      cantidad=$(find "$dir" -type f -name "*.$extension" 2>/dev/null | wc -l)
      echo "$usuario | $cantidad" >> "$salida"
   fi
done
echo "Resultado guardado en $salida"
```

Ejercicio 17.

Escribir un script que, al ejecutarse, imprima en pantalla los nombres de los archivos que se encuentran en el directorio actual, intercambiando minúsculas por mayúsculas, además de eliminar la letra a (mayúscula o minúscula).

Por ejemplo, si, en el directorio actual, están los siguientes archivos:

- *IsO.*
- *pepE.*
- *Maria.*

y se ejecutó ./ejercicio17 , se obtendrá, como resultado:

- *iSo.*
- *PEPe.*
- *mRI.*

Ayuda: Investigar el comando tr.

```
#!/bin/bash
# Script: Ejercicio17.sh
# Uso: ./Ejercicio17.sh

# Algoritmo
for archivo in *; do
    # Transformación del nombre:
    # - Intercambia minúsculas y mayúsculas: 'tr a-zA-Z A-Za-z'
    # - Elimina todas las 'a' o 'A': 'tr -d'
    if [ -f "$archivo" ]; then
        nuevo_nombre=$(echo "$archivo" | tr 'a-zA-Z' 'A-Za-z' | tr -d 'aA')
        echo "$nuevo_nombre"
    fi
done
```

Ejercicio 18.

Crear un script que verifique cada 10 segundos si un usuario se ha logueado en el sistema (el nombre del usuario será pasado por parámetro). Cuando el usuario, finalmente, se logueé, el programa deberá mostrar el mensaje "Usuario XXX logueado en el sistema" y salir.

```
#!/bin/bash
# Script: Ejercicio18.sh
# Uso: ./Ejercicio18.sh

# Control de cantidad de parámetros
if [ $# -ne 1 ]; then
   echo "Error: Se debe ingresar, exactamente, 1 parámetro"
   echo "Uso: $0 <nombre_usuario>"
   exit 1
fi

# Asignación de parámetros
usuario=$1

# Algoritmo
while true; do
   if who | grep -wq "$usuario"; then
      echo "Usuario $usuario logueado en el sistema"
      exit 0
   else
      echo "Esperando que $usuario se loguee..."
      sleep 10
   fi
done
```

Ejercicio 19.

Escribir un Programa de "Menú de Comandos Amigable con el Usuario" llamado menú, el cual, al ser invocado, mostrará un menú con la selección para cada uno de los scripts creados en esta práctica. Las instrucciones de cómo proceder deben mostrarse junto con el menú. El menú deberá iniciarse y permanecer activo hasta que se seleccione "Salir". Por ejemplo:

MENU DE COMANDOS
03. Ejercicio 3
12. Evaluar Expresiones
13. Probar estructuras de control
...
Ingrese la opción a ejecutar: 03

```
#!/bin/bash
# Script: Ejercicio19.sh
# Uso: ./Ejercicio19.sh

# Ruta donde están los scripts creados en esta práctica
SCRIPTS_DIR="/home/jmenduina/practica-shell-script"

# Función "mostrar_menu"
mostrar_menu() {
   clear
   echo "================================"
   echo       "MENÚ DE COMANDOS"
   echo "================================"
   echo "3. Ejercicio 3"
   echo "12a. Ejercicio 12a"
   echo "12b. Ejercicio 12b"
   echo "12c. Ejercicio 12c"
   echo "13a. Ejercicio 13a"
   echo "13b. Ejercicio 13b"
   echo "13c. Ejercicio 13c"
   echo "17. Ejercicio 17"
   echo "19. Ejercicio 19"
   echo "20. Ejercicio 20"
   echo "21. Ejercicio 21"
   echo "22. Ejercicio 22"
   echo "23. Ejercicio 23"
   echo "24. Ejercicio 24"
   echo "25. Ejercicio 25"
   echo "26. Ejercicio 26"
   echo "27. Ejercicio 27"
   echo "28. Ejercicio 28"
   echo "29. Ejercicio 29"
   echo "30. Ejercicio 30"
   echo "31. Ejercicio 31"
```

```
    echo "99. Salir"
    echo "------------------------------"
    echo
    echo -n "Introducir número de script a ejecutar: "
}

# Algoritmo
while true; do
    mostrar_menu
    read opcion
    echo
    case "$opcion" in
        3)
            echo "Ejecutando Ejercicio 3..."
            echo
            bash "$SCRIPTS_DIR/ejercicio3.sh"
            ;;
        12a)
            echo "Ejecutando Ejercicio 12a..."
            echo
            bash "$SCRIPTS_DIR/ejercicio12a.sh"
            ;;
        12b)
            echo "Ejecutando Ejercicio 12b..."
            echo
            bash "$SCRIPTS_DIR/ejercicio12b.sh"
            ;;
        12c)
            echo "Ejecutando Ejercicio 12c..."
            echo
            bash "$SCRIPTS_DIR/ejercicio12c.sh"
            ;;
        13a)
            echo "Ejecutando Ejercicio 13a..."
            echo
            bash "$SCRIPTS_DIR/ejercicio13a.sh"
            ;;
        13b)
            echo "Ejecutando Ejercicio 13b..."
            echo
            bash "$SCRIPTS_DIR/ejercicio13b.sh"
            ;;
        13c)
            echo "Ejecutando Ejercicio 13c..."
            echo
            bash "$SCRIPTS_DIR/ejercicio13c.sh"
            ;;
        17)
            echo "Ejecutando Ejercicio 17..."
            echo
```

```
    bash "$SCRIPTS_DIR/ejercicio17.sh"
    ;;
19)
    echo "Ejecutando Ejercicio 19..."
    echo
    bash "$SCRIPTS_DIR/ejercicio19.sh"
    ;;
20)
    echo "Ejecutando Ejercicio 20..."
    echo
    bash "$SCRIPTS_DIR/ejercicio20.sh"
    ;;
21)
    echo "Ejecutando Ejercicio 21..."
    echo
    bash "$SCRIPTS_DIR/ejercicio21.sh"
    ;;
22)
    echo "Ejecutando Ejercicio 22..."
    echo
    bash "$SCRIPTS_DIR/ejercicio22.sh"
    ;;
23)
    echo "Ejecutando Ejercicio 23..."
    echo
    bash "$SCRIPTS_DIR/ejercicio23.sh"
    ;;
24)
    echo "Ejecutando Ejercicio 24..."
    echo
    bash "$SCRIPTS_DIR/ejercicio24.sh"
    ;;
25)
    echo "Ejecutando Ejercicio 25..."
    echo
    bash "$SCRIPTS_DIR/ejercicio25.sh"
    ;;
26)
    echo "Ejecutando Ejercicio 26..."
    echo
    bash "$SCRIPTS_DIR/ejercicio26.sh"
    ;;
27)
    echo "Ejecutando Ejercicio 27..."
    echo
    bash "$SCRIPTS_DIR/ejercicio27.sh"
    ;;
28)
    echo "Ejecutando Ejercicio 28..."
    echo
```

```
        bash "$SCRIPTS_DIR/ejercicio28.sh"
        ;;
    29)
        echo "Ejecutando Ejercicio 29..."
        echo
        bash "$SCRIPTS_DIR/ejercicio29.sh"
        ;;
    30)
        echo "Ejecutando Ejercicio 30..."
        echo
        bash "$SCRIPTS_DIR/ejercicio30.sh"
        ;;
    31)
        echo "Ejecutando Ejercicio 31..."
        echo
        bash "$SCRIPTS_DIR/ejercicio31.sh"
        ;;
    99)
        echo "Saliendo del menú. ¡Hasta luego!"
        echo
        break
        ;;
    *)
        echo "Opción inválida. Intentar nuevamente"
        ;;
  esac
  echo
  echo -n "Presionar ENTER para continuar..."
  read
done
```

Ejercicio 20.

Realizar un script que simule el comportamiento de una estructura de PILA e implementar las siguientes funciones aplicables sobre una estructura global definida en el script:

- *push: Recibe un parámetro y lo agrega en la pila.*
- *pop: Saca un elemento de la pila.*
- *length: Devuelve la longitud de la pila.*
- *print: Imprime todos elementos de la pila.*

Dentro del mismo script y, utilizando las funciones implementadas:

- *Agregar 10 elementos a la pila*
- *Sacar 3 de ellos.*
- *Imprimir la longitud de la pila.*
- *Luego, imprimir la totalidad de los elementos que en ella se encuentran.*

```
#!/bin/bash
# Script: Ejercicio20.sh
# Uso: ./Ejercicio20.sh

# ESTRUCTURA GLOBAL
pila=()

# FUNCIONES

# Push: Recibe un parámetro y lo agrega a la pila
push() {
   pila+=("$1")
}

# Pop: Saca un elemento de la pila
pop() {
   if [ ${#pila[@]} -eq 0 ]; then
      echo "La pila está vacía. No se puede hacer pop"
   else
      unset 'pila[-1]'
   fi
}

# Length: Devuelve la longitud de la pila
length() {
   echo "${#pila[@]}"
}

# Print: Imprime todos los elementos de la pila
print() {
   for elem in "${pila[@]}"; do
```

```
        echo " - $elem"
    done
}

# PRUEBA

echo "Agregando 10 elementos a la pila..."
for i in {1..10}; do
    push "Elemento_$i"
done

echo "Sacando 3 elementos de la pila..."
for i in {1..3}; do
    pop
done

echo "Longitud de la pila: $(length)"

echo "Impresión de la totalidad de los elementos en la pila:"
print
```

Ejercicio 21.

Dada la siguiente declaración al comienzo de un script:

num=(10 3 5 7 9 3 5 4). (la cantidad de elementos del arreglo puede variar).

Implementar la función productoria dentro de este script, cuya tarea sea multiplicar todos los números que el arreglo contiene.

```
#!/bin/bash
# Script: Ejercicio21.sh
# Uso: ./Ejercicio21.sh

# Vector
num=(10 3 5 7 9 3 5 4)

# Algoritmo
productoria() {
   local prod=1
   for n in "${num[@]}"; do
      prod=$((prod * n))
   done
   echo "$prod"
}

# Resultado
echo "La productoria del vector (${num[@]}) es: $(productoria)"
```

Ejercicio 22.

Implementar un script que recorra un arreglo compuesto por números e imprima en pantalla sólo los números pares y que cuente sólo los números impares y los informe en pantalla al finalizar el recorrido.

```
#!/bin/bash
# Script: Ejercicio22.sh
# Uso: ./Ejercicio22.sh

# Vector
num=(10 3 5 7 9 3 5 4)

# Algoritmo
impares=0
for n in "${num[@]}"; do
   if (( n % 2 == 0 )); then
      echo "Número par: $n"
   else
      ((impares++))
   fi
done

# Resultado
echo "Cantidad de números impares: $impares"
```

Ejercicio 23.

Dada la definición de 2 vectores del mismo tamaño y cuyas longitudes no se conocen.

vector1=(1 .. N).
vector2=(1.. N).

Por ejemplo:

vector1=(1 80 65 35 2) y vector2=(5 98 3 41 8).

Completar este script de manera tal de implementar la suma elemento a elemento entre ambos vectores y que la misma sea impresa en pantalla de la siguiente manera:

- *La suma de los elementos de la posición 0 de los vectores es 6.*
- *La suma de los elementos de la posición 1 de los vectores es 178.*
- *La suma de los elementos de la posición 4 de los vectores es 10.*

```
#!/bin/bash
# Script: Ejercicio23.sh
# Uso: ./Ejercicio23.sh

# Vectores
vector1=(1 80 65 35 2)
vector2=(5 98 3 41 8)

# Algoritmo
for ((i=0; i<${#vector1[@]}; i++)); do
   suma=$(( ${vector1[i]} + ${vector2[i]} ))
   echo "La suma de los elementos de la posición $i de los vectores es $suma"
done
```

Ejercicio 24.

Realizar un script que agregue, en un arreglo, todos los nombres de los usuarios del sistema pertenecientes al grupo "users". Adicionalmente, el script puede recibir como parámetro:

- *"-b n": Retorna el elemento de la posición n del arreglo si el mismo existe. Caso contrario, un mensaje de error.*
- *"-l": Devuelve la longitud del arreglo.*
- *"-i": Imprime todos los elementos del arreglo en pantalla.*

```
#!/bin/bash
# Script: Ejercicio24.sh
# Uso: ./Ejercicio24.sh

# Grupo
group="users"

# Obtención de todos los nombres de los usuarios del sistema pertenecientes al grupo "$group"
usuarios=($(getent group "$group" | cut -d: -f4 | tr ',' ' '))

# Verificación de usuarios encontrados
if [ ${#usuarios[@]} -eq 0 ]; then
   echo "No se encontraron usuarios del sistema pertenecientes al grupo '$group'"
   exit 1
fi

# Algoritmo
case "$1" in
   -b)
      n=$2
      if [ -z "$n" ]; then
         echo "Error: Se debe especificar un número después de '-b'"
         exit 1
      elif [ "$n" -ge 0 ] && [ "$n" -lt "${#usuarios[@]}" ]; then
         echo "Elemento en la posición $n: ${usuarios[$n]}"
      else
         echo "Error: No existe la posición $n en el arreglo"
      fi
      ;;
   -l)
      echo "La longitud del arreglo es: ${#usuarios[@]}"
      ;;
   -i)
      echo "Usuarios en el grupo '$group':"
      for u in "${usuarios[@]}"; do
         echo " - $u"
      done
```

```
        ;;
    *)
        echo "Uso: $0 [-b n | -l | -i]"
        echo " -b n : Retorna el elemento de la posición n del arreglo"
        echo " -l   : Retorna la longitud del arreglo"
        echo " -i   : Imprime todos los elementos del arreglo"
        ;;
esac
```

Ejercicio 25.

Escribir un script que reciba una cantidad desconocida de parámetros al momento de su invocación (debe validar que, al menos, se reciba uno). Cada parámetro representa la ruta absoluta de un archivo o directorio en el sistema. El script deberá iterar por todos los parámetros recibidos y, sólo para aquellos parámetros que se encuentren en posiciones impares (el primero, el tercero, etc.), verificar si el archivo o directorio existen en el sistema, imprimiendo en pantalla qué tipo de objeto es (archivo o directorio). Además, se deberá informar la cantidad de archivos o directorios inexistentes en el sistema.

```
#!/bin/bash
# Script: Ejercicio25.sh
# Uso: ./Ejercicio25.sh

# Control de cantidad de parámetros
if [ $# -lt 1 ]; then
   echo "Error: Se debe ingresar, al menos, 1 parámetro"
   echo "Uso: $0 <ruta1> <ruta2> ..."
   exit 1
fi

# Algoritmo
inexistentes=0
posicion=1
for ruta in "$@"; do
   if (( posicion % 2 != 0 )); then
      if [ -e "$ruta" ]; then
         if [ -f "$ruta" ]; then
            echo "Posición $posicion: '$ruta' existe y es un archivo"
         elif [ -d "$ruta" ]; then
            echo "Posición $posicion: '$ruta' existe y es un directorio"
         else
            echo "Posición $posicion: '$ruta' existe, pero no es ni archivo ni directorio"
         fi
      else
         echo "Posición $posicion: '$ruta' no existe"
         ((inexistentes++))
      fi
   fi
   ((posicion++))
done

# Resultado
echo "Cantidad de archivos o directorios inexistentes: $inexistentes"
```

Ejercicio 26.

Realizar un script que implemente, a través de la utilización de funciones, las operaciones básicas sobre arreglos:

- *inicializar: Crea un arreglo llamado array vacío.*
- *agregar_elem <parametro1>: Agrega al final del arreglo el parámetro recibido.*
- *eliminar_elem <parametro1>: Elimina del arreglo el elemento que se encuentra en la posición recibida como parámetro. Debe validar que se reciba una posición válida.*
- *longitud: Imprime la longitud del arreglo en pantalla.*
- *imprimir: Imprime todos los elementos del arreglo en pantalla.*
- *inicializar_con_valores <parametro1><parametro2>: Crea un arreglo con longitud <parametro1> y, en todas las posiciones, asigna el valor <parametro2>.*

```
#!/bin/bash
# Script: Ejercicio26.sh
# Uso: ./Ejercicio26.sh

# FUNCIONES

# Inicializar: Crea un arreglo llamado array vacío
inicializar() {
   array=()
   echo "Arreglo llamado 'array' creado vacío"
}

# Agregar_elem: Agrega al final del arreglo el parámetro recibido
agregar_elem() {
   if [ $# -ne 1 ]; then
      echo "Error: Se debe indicar el valor para agregar"
      return 1
   fi
   array+=("$1")
   echo "Elemento '$1' agregado al final del arreglo"
}

# Eliminar_elem: Elimina del arreglo el elemento que se encuentra en la posición recibida como parámetro
eliminar_elem() {
   if [ $# -ne 1 ]; then
      echo "Error: Se debe indicar la posición a eliminar"
      return 1
   fi
   local pos=$1
   if [ "$pos" -lt 0 ] || [ "$pos" -ge "${#array[@]}" ]; then
      echo "Error: Posición fuera de rango (0 a $((${#array[@]} - 1)))"
      return 1
```

```
    fi
    unset 'array[pos]'
    array=("${array[@]}")
    echo "Elemento en posición $pos eliminado"
}

# Longitud: Imprime la longitud del arreglo
longitud() {
    echo "Longitud del arreglo: ${#array[@]}"
}

# Imprimir: Imprime todos los elementos del arreglo
imprimir() {
    if [ ${#array[@]} -eq 0 ]; then
        echo "El arreglo está vacío"
    else
        echo "Elementos del arreglo: ${array[@]}"
    fi
}

# Inicializar_con_valores: Crea un arreglo con longitud <parametro1> y, en todas las
posiciones, asigna el valor <parametro2>
inicializar_con_valores() {
    if [ $# -ne 2 ]; then
        echo "Error: Se debe indicar longitud y valor"
        return 1
    fi
    local longitud=$1
    local valor=$2
    array=()
    for ((i = 0; i < longitud; i++)); do
        array+=("$valor")
    done
    echo "Arreglo llamado 'array' creado con longitud $longitud y valor '$valor' en todas
las posiciones"
}

# PRUEBA

inicializar
agregar_elem "A"
agregar_elem "B"
agregar_elem "C"
imprimir
eliminar_elem 1
imprimir
longitud
inicializar_con_valores 3 X
imprimir
```

Ejercicio 27.

Realizar un script que reciba como parámetro el nombre de un directorio. Se deberá validar que el mismo exista y, de no existir, causar la terminación del script con código de error 4. Si el directorio existe, deberá contar por separado la cantidad de archivos que en él se encuentran para los cuales el usuario que ejecuta el script tiene permiso de lectura y escritura, e informar dichos valores en pantalla. En caso de encontrar subdirectorios, no deberán procesarse y tampoco deberán ser tenidos en cuenta para la suma a informar.

```
#!/bin/bash
# Script: Ejercicio27.sh
# Uso: ./Ejercicio27.sh

# Control de cantidad de parámetros
if [ $# -ne 1 ]; then
   echo "Error: Se debe ingresar, exactamente, 1 parámetro"
   echo "Uso: $0 <directorio>"
   exit 1
fi

# Asignación de parámetros
directorio="$1"

# Verificación de que el directorio exista
if [ ! -d "$directorio" ]; then
   echo "Error: '$directorio' no es un directorio válido"
   exit 4
fi

# Algoritmo
rw_count=0
r_count=0
w_count=0
for archivo in "$directorio"/*; do
   if [ -f "$archivo" ]; then
      if [ -r "$archivo" ] && [ -w "$archivo" ]; then
         ((rw_count++))
      fi
      if [ -r "$archivo" ]; then
         ((r_count++))
      fi
      if [ -w "$archivo" ]; then
         ((w_count++))
      fi
   fi
done

# Resultados
```

```
echo "Archivos con permisos de lectura y escritura: $rw_count"
echo "Archivos con permiso de lectura: $r_count"
echo "Archivos con permiso de escritura: $w_count"
```

Ejercicio 28.

Implementar un script que agregue a un arreglo todos los archivos del directorio /home cuya terminación sea .doc. Adicionalmente, implementar las siguientes funciones que permitan acceder a la estructura creada:

- *verArchivo <nombre_de_archivo>: Imprime el archivo en pantalla si el mismo se encuentra en el arreglo. Caso contrario, imprime el mensaje de error "Archivo no encontrado" y devuelve, como valor de retorno, 5.*
- *cantidadArchivos: Imprime la cantidad de archivos del /home con terminación .doc.*
- *borrarArchivo <nombre_de_archivo>: Consulta al usuario si quiere eliminar el archivo lógicamente. Si el usuario responde "Sí", elimina el elemento sólo del arreglo. Si el usuario responde "No", elimina el archivo del arreglo y también del FileSystem. Debe validar que el archivo exista en el arreglo. En caso de no existir, imprime el mensaje de error "Archivo no encontrado" y devuelve, como valor de retorno, 10.*

Ejercicio 29.

Realizar un script que mueva todos los programas del directorio actual (archivos ejecutables) hacia el subdirectorio "bin" del directorio HOME del usuario actualmente logueado. El script debe imprimir en pantalla los nombres de los que mueve e indicar cuántos ha movido o que no ha movido ninguno. Si el directorio "bin" no existe, deberá ser creado.

Ejercicio 30.

Implementar la estructura de datos Set (Conjunto de valores) en Bash. Un conjunto se define como una colección de valores únicos, es decir, que sólo almacena una vez cada valor, aun cuando se intente agregar el mismo valor más de una vez. La implementación debe soportar las siguientes operaciones mediante funciones:

- *initialize: Inicializa el set vacío.*
- *initialize_with: Inicializa el set con un conjunto de valores que recibe como argumento (debe validar que se reciba, al menos, uno).*
- *add: Agrega un valor al conjunto, el cual recibe como argumento. No debe agregar elementos repetidos. El resultado de la operación será un éxito sólo si el valor puede ser agregado al conjunto.*
- *remove: Elimina uno o más valores del conjunto, los cuales recibe como argumentos. Si la operación elimina, al menos, un valor, se considera un éxito.*
- *contains: Chequea si el conjunto contiene un valor recibido como argumento. El resultado será éxito si el valor está en el conjunto.*
- *print: Imprime los elementos del conjunto, de a uno por línea.*
- *print_sorted: Imprime los elementos del conjunto, de a uno por línea y ordenados alfabéticamente. Tip: Investigar cómo combinar el comando sort con la función print.*

En un script separado, incorporar y utilizar las funciones implementadas para desarrollar un juego de bingo. El bingo deberá generar números aleatorios dentro de un rango entre 0 y un valor máximo que puede especificarse mediante un argumento del script, de manera opcional. El valor máximo no puede ser 0 ni superior a 32.767, y en caso de no especificarse, se tomará como valor por defecto 99. En cada ronda, se generará un nuevo número que ya no haya sido utilizado y se lo cantará, imprimiendo en la salida estándar. Luego de esto, se esperará entrada del usuario para saber si se debe cantar "BINGO" para finalizar la partida o se debe cantar un nuevo número. Al finalizar, el script deberá imprimir, en orden, los números que se cantaron hasta que se produjo el bingo.

Tip: Investigar la variable de entorno $RANDOM de Bash para obtener valores aleatorios.

Ejercicio 31.

Realizar un script que reciba como argumento una lista de posibles nombres de usuarios del sistema y, para cada uno de los que, efectivamente, existan en el sistema y posean un directorio personal configurado que sea válido, realice las modificaciones necesarias en su directorio personal para que tenga un subdirectorio llamado "directorio_iso" con la siguiente estructura:

Para resolver la creación de los directorios y archivos, utilizar la funcionalidad "Brace Expansion" brindada por bash: https://www.gnu.org/software/bash/manual/html_node/Brace-Expansion.html.

Juan Menduiña

Trabajo Práctico N° 4:

.

Ejercicio 1.