Práctica 1 - Kernel Linux

A - Introducción

El propósito de esta primera sección de la práctica es introducir los conceptos preliminares que necesitará el alumno, para desarrollar la actividad práctica de la sección B de esta guía de estudio.

1. ¿Qué es GCC?

GCC (GNU Compiler Collection) es un compilador desarrollado por el proyecto GNU que soporta múltiples lenguajes de programación, incluyendo C, C++, Objective-C, Fortran, Ada, y Go, entre otros. Es uno de los compiladores más utilizados en sistemas operativos tipo Unix, como Linux, y es una pieza fundamental en muchas distribuciones.

Ejemplo básico para compilar un archivo en C: `gcc -o mi_programa programa.c`

- `gcc` es el compilador.
- `-o mi_programa` especifica el nombre del archivo ejecutable generado.
- `programa.c` es el archivo fuente en C.

Algunas opciones útiles de GCC:

- `-Wall`: Muestra todos los mensajes de advertencia posibles.
- `-g`: Genera información de depuración.
- `-O2`, `-O3`: Habilita diferentes niveles de optimización.
- `-c`: Compila sin enlazar, generando un archivo objeto (.o).
- `-l`: Especifica bibliotecas a enlazar.

2. ¿Qué es make y para que se usa?

`make` es una herramienta del proyecto GNU, específicamente GNU Make. Su función principal es generar ejecutables u otros archivos a partir de archivos fuente. Para hacer esto, lee las instrucciones de un archivo llamado *Makefile* (que distingue entre mayúsculas y minúsculas).

El Makefile contiene reglas que definen cómo generar un archivo específico, llamado "target" (objetivo). Un target puede depender de otros targets, que se conocen como prerequisitos. Cada regla en el Makefile especifica los comandos necesarios (la "receta") para crear el target a partir de sus prerequisitos.

`make` se utiliza para:

- Organizar el proceso de compilación de software, como el kernel de Linux, de manera que se pueda ejecutar con unos pocos comandos.
- Automatizar la generación de ejecutables a partir de código fuente en lenguajes como C, Assembler y Rust. En el caso de C, por ejemplo, make puede automatizar los pasos de compilación con *gcc* para generar archivos objeto y luego enlazarlos para crear el ejecutable.
- Realizar otras tareas como generar archivos PDF a partir de documentos LaTeX o Markdown, o empaquetar proyectos de Python en un solo archivo.
- Optimizar el proceso de construcción, ya que si los archivos fuente no han cambiado desde la última ejecución, make no vuelve a realizar las mismas acciones. Esto ahorra tiempo en compilaciones posteriores.
- Definir targets que no generan archivos, como targets para mostrar ayuda (help) o para limpiar archivos generados (clean), los cuales se suelen marcar como "phony".

En el contexto de la compilación del kernel de Linux, el comando `$ make` busca el archivo Makefile en el directorio del código fuente, interpreta sus directivas y comienza el proceso de compilación del kernel. También se utiliza `$ make modules` para compilar los módulos del kernel y `$ sudo make install` para instalar el kernel.

3. La carpeta /home/so/practica1/ejemplos/01-make de la VM contiene ejemplos de uso de make. Analice los ejemplos, en cada caso ejecute `make` y luego `make run` (es opcional ejecutar el ejemplo 4, el mismo requiere otras dependencias que no vienen pre instaladas en la VM):

Ejemplos de Ejecución:

```
root@so:/home/so/practica1/ejemplos/01-make/01-helloworld# make
gcc -Wall --std=c99 -o helloworld helloworld.c
root@so:/home/so/practica1/ejemplos/01-make/01-helloworld# make run
./helloworld
Hello world!
```

Ejecución de 01-helloworld

```
root@so:/home/so/practica1/ejemplos/01-make/02-multiplefiles# make
cc -Wall --std=c11   -c -o dlinkedlist.o dlinkedlist.c
gcc -Wall --std=c11 dlinkedlist.o revert.c -o revert
root@so:/home/so/practica1/ejemplos/01-make/02-multiplefiles# make run
./revert
Value 9
Value 8
Value 7
Value 6
Value 5
Value 4
Value 3
Value 2
Value 1
Value 0
```

Ejecución de 02-multiplefiles

- Vuelva a ejecutar el comando `make`. ¿Se volvieron a compilar los programas? ¿Por qué?

El comportamiento de `make` se basa en la **fecha de modificación de los archivos**. Cuando ejecutás el primer `make`, el compilador genera los archivos objeto (`.o`) y el ejecutable solo si:

1. El archivo fuente (`.c`, `.cpp`, etc.) **es más reciente** que el archivo objeto.
2. El archivo objeto **es más reciente** que el ejecutable.

Una vez que los archivos se compilan correctamente, la fecha de modificación del ejecutable se actualiza. Por lo tanto, si no cambiaste ningún archivo fuente después de la última compilación, `make` detecta que **el ejecutable ya está actualizado** y no vuelve a compilar.

```
root@so:/home/so/practica1/ejemplos/01-make/01-helloworld# make
make: No se hace nada para 'all'.
```

Re-ejecución de comando `make`

- Cambie la fecha de modificación de un archivo con el comando `touch` o editando el archivo y ejecute `make`. ¿Se volvieron a compilar los programas? ¿Por qué?

Si, si se cambia la fecha de modificación de un archivo fuente usando el comando `touch` o editando el archivo, `make` **recompilará los programas afectados**, `make` verifica las **marcas de tiempo (timestamps)** de los archivos para determinar si es necesario recompilar. Si la fecha de modificación de un archivo fuente (`.c`) es más reciente que el archivo objeto (`.o`) correspondiente, `make` asume que el código fuente ha cambiado y vuelve a compilar ese archivo en particular.

```
root@so:/home/so/practica1/ejemplos/01-make/01-helloworld# touch helloworld.c
root@so:/home/so/practica1/ejemplos/01-make/01-helloworld# make
gcc -Wall --std=c99 -o helloworld helloworld.c
```

Ejecución post comando `touch`

- ¿Por qué “run” es un target “phony”?

El target `run` es un target "phony" porque no corresponde a un archivo real en el sistema de archivos, sino que es una convención para ejecutar acciones específicas. Un target *phony* en un Makefile es un objetivo que no genera un archivo de salida. En lugar de producir un archivo, su función es ejecutar comandos. Por lo tanto, no importa si existe un archivo llamado `run` en el directorio, `make` siempre ejecutará los comandos asociados al target.

Razones por las que se declara como phony:

- Evitar conflictos con archivos reales: Si en el mismo directorio existiera un archivo llamado `run`, `make` lo consideraría como un archivo actualizado y no ejecutaría los comandos del target. Al declarar el target como phony, garantizamos que siempre se ejecute.
- Claridad y convenciones: Es común utilizar targets como `run`, `clean`, `install`, etc., que representan acciones y no productos finales. Declararlos como phony deja claro que no generan archivos.
- Optimización: `make` omite verificaciones innecesarias de timestamps para estos targets, lo que hace el proceso un poco más rápido.
- En el ejemplo 2 la regla para el target `dlinkedlist.o` no define cómo generar el target, sin embargo el programa se compila correctamente. ¿Por qué es esto?

El programa se compila correctamente porque `make` utiliza una regla implícita o automática para generar el archivo objeto desde el código fuente. Esto hace que, aunque no haya una regla explícita para `dlinkedlist.o` en el *Makefile*, el proceso de compilación funcione igual.

Las reglas implícitas o automáticas son una serie de reglas que GNU Make utiliza cuando no se encuentran reglas explícitas en el Makefile. Una de las reglas más comunes y útiles es la que maneja la compilación de archivos objeto (`.o`) a partir de archivos fuente (`.c`).

Lo que sucede es: `gcc -c dlinkedlist.c -o dlinkedlist.o`

> **Nota** >
>
> Si no usás la VM podés descargar los ejemplos desde: https://gitlab.com/unlp-so/codigo-para-practicas

4. ¿Qué es el kernel de GNU/Linux? ¿Cuáles son sus funciones principales dentro del Sistema Operativo?

El kernel de GNU/Linux es el programa fundamental que ejecuta otros programas y gestiona los dispositivos de hardware. Se encarga de asegurar que el software y el hardware puedan trabajar en conjunto. En un sentido estricto, es el Sistema Operativo mismo.

Sus funciones principales dentro del Sistema Operativo son:

- Administración de la memoria principal.
- Administración del uso de la CPU.
- Gestión de los recursos de hardware.
- Control de la ejecución de los procesos.
- Actúa como interfaz entre las aplicaciones y el hardware.
- Implementa servicios esenciales como:
  - Manejo de memoria.
  - Manejo de la CPU.
  - Administración de procesos.
  - Comunicación y Concurrencia.
  - Gestión de la E/S (Entrada/Salida).
- Es responsable de facilitar a los procesos un acceso seguro al hardware, utilizando para ello una interfaz conocida como “llamadas al sistema” (system calls).
- Se ejecuta en modo supervisor o privilegiado, lo que le permite acceder al conjunto completo de instrucciones, incluyendo el acceso al hardware, la capacidad de direccionar la memoria y programar la CPU.

5. Explique brevemente la arquitectura del kernel Linux teniendo en cuenta: tipo de kernel, módulos, portabilidad, etc.

La arquitectura del kernel de Linux se caracteriza por ser principalmente un **núcleo monolítico híbrido.**

Tipo de Kernel:

- Si bien su estructura base es la de un kernel monolítico, donde la mayoría de los servicios del sistema operativo (gestión de procesos, memoria, archivos, controladores, etc.) se ejecutan en modo supervisor, la capacidad de cargar y descargar módulos dinámicamente le otorga la característica de híbrido.

Módulos:

- Los módulos del kernel son fragmentos de código que pueden cargarse y descargarse en el mapa de memoria del sistema operativo (kernel) bajo demanda. Estos módulos permiten extender la funcionalidad del kernel "en caliente", sin necesidad de reiniciar el sistema. Pueden proporcionar soporte para nuevo hardware, como controladores de dispositivos, o agregar nuevas funcionalidades, como soporte para diferentes sistemas de archivos. Todo el código de los módulos se ejecuta en modo kernel (privilegiado).

Portabilidad:

- Una característica destacada del kernel de Linux es su alta portabilidad. Desde la misma estructura de código fuente, se da soporte a todas las arquitecturas. Esto significa que el kernel puede ejecutarse en una amplia variedad de hardware, desde sistemas embebidos hasta servidores y supercomputadoras.

Capas o Componentes Principales:

- *Interfaz de Llamadas al Sistema (System Call Interface):* Esta capa proporciona el mecanismo para que los procesos de usuario soliciten servicios al kernel, como acceso al hardware o gestión de procesos.
- *Sistema de Archivos Virtual (Virtual File System - VFS):* Esta capa proporciona una interfaz unificada para los diferentes sistemas de archivos, permitiendo que las aplicaciones accedan a ellos de manera transparente.
- *Hardware:* Esta es la capa física que el kernel gestiona directamente.
- *Espacio del Usuario (User Space):* Contiene las aplicaciones y servicios del sistema.
- *Espacio del Núcleo/Kernel (Kernel Space):* Aquí residen las partes críticas del kernel, entre ellas encontramos:
  - *Gestión de Procesos:* Este componente se encarga de la creación, planificación y finalización de los procesos, así como de la gestión de la concurrencia.
  - *Gestión de Memoria:* Responsable de la administración de la memoria principal, incluyendo la asignación y liberación de memoria, la memoria virtual y la protección de la memoria.
  - *Control de Dispositivos (Device Control):* Esta parte del kernel gestiona la interacción con el hardware del sistema a través de los controladores de dispositivos (device drivers). Los drivers se ejecutan en modo privilegiado.

6. ¿Cómo se define el versionado de los kernels Linux en la actualidad?

En la actualidad (a partir de la versión 3.0 del kernel Linux), el esquema de versionado se define de la siguiente manera: **A.B.C[-rcX]**

Donde:

- A denota la revisión mayor. Este número cambia con menor frecuencia, generalmente cada varios años. El cambio de la versión 2.6 a la 3.0 y luego a la 4.0, 5.0 y 6.0 son ejemplos de cambios en esta revisión mayor. Estos cambios a menudo marcan la introducción de funcionalidades significativas o un hito en la historia del kernel, como los 20 años del SO en el caso del paso a la versión 3.0 o la decisión de los usuarios en el caso de la versión 4.0.
- B denota la revisión menor. Este número cambia con más frecuencia que la revisión mayor y generalmente indica la adición de nuevos drivers o características al kernel.

- C es el número de revisión. Este número se incrementa con cada nueva versión estable que incluye correcciones de errores y pequeñas mejoras.
- -rcX indica una versión de prueba ("release candidate"). La "rc" seguida de un número (X) señala una versión candidata a ser la próxima versión estable. Estas versiones se publican durante la ventana de corrección de bugs después de la ventana de merge donde se incorporan nuevas características.

Este esquema de tres números (A.B.C) se adoptó nuevamente con el lanzamiento de la versión 3.0 en 2011, después de que la versión 2.6 llegara a su fin.

Anteriormente, entre la versión 2.6 y la 3.0, se utilizaba un esquema A.B.C.[D] donde D se usaba para correcciones graves sin agregar nueva funcionalidad.

Antes de la versión 2.6, el segundo número (Y en X.Y.Z) indicaba si era una versión de producción (número par) o de desarrollo (número impar).

7. ¿Cuáles son los motivos por los que un usuario/a GNU/Linux puede querer re-compilar el kernel?

Un usuario/a de GNU/Linux puede querer re-compilar el kernel por varios motivos importantes:

- Soportar nuevos dispositivos: Esto incluye la necesidad de añadir soporte para hardware que no está soportado por el kernel precompilado, como por ejemplo, una placa de video específica.
- Agregar mayor funcionalidad: Recompilar el kernel permite incorporar nuevas funcionalidades que no están habilitadas por defecto, como el soporte para nuevos sistemas de archivos.
- Optimizar el funcionamiento según el sistema: La recompilación permite ajustar el kernel específicamente para el hardware en el que se va a ejecutar, lo que puede resultar en un mejor rendimiento.
- Adaptarlo al sistema (quitar soporte de hardware no utilizado): Un usuario puede querer reducir el tamaño del kernel y potencialmente mejorar la eficiencia al eliminar el soporte para hardware que no está presente en su sistema. Al personalizar la configuración del kernel antes de la compilación, se pueden deshabilitar opciones innecesarias.
- Corrección de bugs (problemas de seguridad o errores de programación): Si se descubre un error de seguridad o un fallo en el kernel, un usuario podría recompilar el kernel después de aplicar un parche que solucione el problema.

8. ¿Cuáles son las distintas opciones y menús para realizar la configuración de opciones de compilación de un kernel? Cite diferencias, necesidades (paquetes adicionales de software que se pueden requerir), pro y contras de cada una de ellas.

Existen tres interfaces principales que permiten generar y configurar el archivo `.config` del kernel de Linux, el cual contiene las directivas de qué debe compilar el kernel. Estas opciones son:

`make config`:

- Descripción: Esta opción presenta una interfaz de texto secuencial. Las opciones de configuración se presentan una por una en la terminal, y el usuario debe responder a cada pregunta (generalmente con "y" para sí, "n" para no, o "m" para compilar como módulo).
- Diferencias: Su principal diferencia radica en su naturaleza lineal y basada en texto, sin navegación o búsqueda directa de opciones.
- Necesidades: No requiere paquetes de software adicionales más allá de las herramientas de desarrollo básicas necesarias para la compilación del kernel (como GCC, Make, etc.).
- Pros:
  - Es la opción más básica y siempre disponible en cualquier entorno de terminal.
  - No depende de entornos gráficos o librerías adicionales.
- Contras:
  - Se considera tediosa.
  - Es propensa a errores debido a la gran cantidad de opciones y la falta de una visión general de la configuración.
  - La navegación y la búsqueda de opciones específicas son difíciles.

`make xconfig`:

- Descripción: Esta opción proporciona una interfaz gráfica para la configuración del kernel, utilizando un sistema de ventanas.
- Diferencias: Se diferencia de `make config` y `make menuconfig` por su interfaz gráfica, que permite una navegación más visual y el uso del ratón para seleccionar opciones.
- Necesidades: Requiere un *sistema de ventanas* (como X Window System) instalado y configurado en el sistema. Si X no está instalado, esta opción no estará disponible.
- Pros:
  - Ofrece una interfaz más intuitiva y fácil de usar para aquellos familiarizados con entornos gráficos.
  - Puede facilitar la exploración y búsqueda de diferentes opciones de configuración.
- Contras:
  - No está disponible en todos los sistemas, especialmente en aquellos que no tienen un entorno gráfico instalado.
  - Depende de la correcta funcionalidad del sistema de ventanas.

`make menuconfig`:

- **Descripción:** Esta opción utiliza *ncurses*, una librería que permite generar una interfaz con paneles directamente en la terminal. Presenta un menú jerárquico donde las opciones se organizan por categorías, y se puede navegar utilizando el teclado.
- **Diferencias:** Se diferencia de `make config` por su interfaz basada en menús y navegación, y de `make xconfig` por ser una aplicación de terminal en lugar de una GUI.
- **Necesidades:** Requiere la librería *ncurses* y sus archivos de desarrollo instalados en el sistema4 . En sistemas Debian y derivados, el paquete *libncurses-dev* proporciona estos archivos.
- **Pros:**
  - Generalmente se considera la opción más utilizada y recomendada.
  - Ofrece una interfaz organizada y navegable directamente en la terminal, sin necesidad de un entorno gráfico.
  - Permite una búsqueda más fácil de las opciones a través de las categorías de menú.
- **Contras:**
  - Aunque es más amigable que `make config`, la navegación se realiza principalmente con el teclado, lo que podría ser menos intuitivo para algunos usuarios acostumbrados a interfaces gráficas.
  - Depende de la disponibilidad de la librería *ncurses*.

En resumen, la elección de la interfaz de configuración del kernel dependerá de las preferencias del usuario, el entorno en el que esté trabajando (si tiene o no un sistema de ventanas), y las herramientas que tenga instaladas. `make menuconfig` se destaca como la opción más comúnmente recomendada por su equilibrio entre usabilidad y disponibilidad en la terminal, siempre que la librería *ncurses* esté presente.

9. Indique qué tarea realiza cada uno de los siguientes comandos durante la tarea de configuración/compilación del kernel:

- `make menuconfig`

Este comando invoca una interfaz de configuración basada en texto utilizando la librería *ncurses*. Permite al usuario seleccionar las opciones que se incluirán en la compilación del kernel, como soporte para diferentes dispositivos, sistemas de archivos y funcionalidades. La configuración realizada a través de esta interfaz se guarda en el archivo `.config` en el directorio del código fuente del kernel.

- `make clean`

Este comando elimina los archivos generados en una compilación anterior (como los archivos `.o` y otros archivos temporales de compilación) del directorio del código fuente del kernel. Tiene como **propósito** asegurarse de que la compilación actual comience desde un estado

limpio, sin residuos de compilaciones previas, lo que evita errores derivados de archivos obsoletos.

- `make (investigue la funcionalidad del parámetro -j)`

El comando `make` por sí mismo se encarga de la compilación del kernel de acuerdo con las reglas definidas en el archivo `Makefile`. Al ejecutarlo, `make` construye todos los objetos necesarios (como los archivos `.o` y los binarios) para crear el kernel final.

`make -jN`: El parámetro `-j` permite la compilación en paralelo, donde `N` es el número de procesos (hilos) a ejecutar simultáneamente. Tiene como propósito mejorar la velocidad de compilación al usar varios núcleos de la CPU para realizar compilaciones en paralelo.

- `make modules (utilizado en antiguos kernels, actualmente no es necesario)`

Este comando compila únicamente los módulos del kernel (drivers y otros componentes que pueden ser cargados y descargados dinámicamente en el sistema).

Antes era común usar este comando cuando querías compilar solo los módulos sin recompilar todo el kernel. En las versiones más recientes del kernel, no es necesario, ya que la compilación de módulos se maneja automáticamente cuando se compila el kernel con `make`.

Actualmente con las versiones más nuevas del kernel, simplemente ejecutar `make` compila tanto el kernel como los módulos sin necesidad de ejecutar `make modules` por separado.

- `make modules_install`

Este comando instala los módulos del kernel que se han compilado, copiándolos a las ubicaciones apropiadas dentro del sistema de archivos (generalmente en `/lib/modules/`). Tiene como propósito asegurar ,luego de que se haya compilado el kernel y sus módulos, que los módulos estén disponibles para ser cargados por el sistema. Esto es necesario para que los controladores y otros módulos del kernel funcionen correctamente.

- `make install`

Este comando instala el kernel compilado y sus archivos asociados en el sistema. Esto incluye copiar el archivo de imagen del kernel (`vmlinuz`), los módulos del kernel y otros archivos relacionados (como el archivo de configuración del kernel) en los directorios adecuados. Tiene como propósito instalar el kernel compilado para que el sistema lo pueda usar como el kernel de arranque.

Tareas que realiza:

- Instalar el kernel en el directorio `/boot/`.

- Instalar el archivo de configuración del kernel (`.config`).
- Instalar los módulos del kernel (utilizando `make modules_install`).
- Actualizar el gestor de arranque (como GRUB) para que reconozca el nuevo kernel.

10. Una vez que el kernel fue compilado, ¿dónde queda ubicada su imagen? ¿dónde debería ser reubicada? ¿Existe algún comando que realice esta copia en forma automática?

Una vez que el kernel ha sido compilado, la imagen del kernel se encuentra ubicada en el directorio `directorio-del-código/arch/arquitectura/boot/`. La imagen del kernel debería ser reubicada al directorio `/boot/` para que el gestor de arranque (como GRUB) pueda encontrarla e iniciar el sistema con el nuevo kernel. El comando que realiza esta copia de forma automática es `$ sudo make install` ya que hay una regla en el archivo `Makefile` que se encarga de instalar el kernel y otros archivos necesarios en el directorio `/boot/` al ejecutar este comando.

11. ¿A qué hace referencia el archivo `initramfs` ? ¿Cuál es su funcionalidad? ¿Bajo qué condiciones puede no ser necesario?

Un `initramfs` es un sistema de archivos temporal que se monta durante el arranque del sistema.

Su funcionalidad principal es contener ejecutables, drivers y módulos que son necesarios para lograr iniciar el sistema. Esto es crucial porque el kernel recién compilado podría no tener incorporado todo el soporte de hardware necesario para acceder al disco raíz donde reside el sistema operativo completo. El `initramfs` proporciona un entorno mínimo con las herramientas esenciales para montar el sistema de archivos raíz. Una vez que el sistema arranca por completo, este disco temporal se desmonta.

Basándonos en su funcionalidad, podríamos inferir que si el kernel estuviera compilado con soporte built-in (no como módulos) para todo el hardware esencial necesario para acceder al disco raíz (controlador de disco, sistema de archivos, etc.), entonces teóricamente un `initramfs` podría no ser estrictamente requerido. En este escenario, el kernel tendría todo lo necesario incorporado para montar el sistema de archivos raíz directamente al arrancar.

12. ¿Cuál es la razón por la que una vez compilado el nuevo kernel, es necesario reconfigurar el gestor de arranque que tengamos instalado?

Una vez que se compila e instala un nuevo kernel, es necesario reconfigurar el gestor de arranque (como GRUB) para que pueda reconocer y ofrecer la opción de arrancar con este nuevo kernel.

- El gestor de arranque necesita conocer la ubicación del nuevo kernel.

- El gestor de arranque necesita información adicional sobre el nuevo kernel como por ejemplo sobre el `initramfs`.
- El proceso de instalación del kernel no configura automáticamente el gestor de arranque.

Luego de instalar el kernel, para que el gestor de arranque lo reconozca simplemente deberemos ejecutar, como usuario privilegiado, el siguiente comando: `# update-grub2`. Este comando es el encargado de escanear los kernels instalados en el sistema, generar la configuración necesaria y actualizar el archivo de configuración del gestor de arranque (normalmente `/boot/grub/grub.cfg` para GRUB 2) para incluir el nuevo kernel como una opción de arranque.

13. ¿Qué es un módulo del kernel? ¿Cuáles son los comandos principales para el manejo de módulos del kernel?

Un módulo del Kernel es un fragmento de código que puede cargarse o descargarse en el mapa de memoria del Sistema Operativo (Kernel) bajo demanda. Su funcionalidad principal radica en que permiten extender la funcionalidad del Kernel en "caliente", es decir, sin necesidad de reiniciar el sistema.

Comandos principales

- `lsmod`: Muestra una lista de todos los módulos que están actualmente cargados en el kernel.
- `modinfo`: Proporciona información detallada acerca de un módulo, como la descripción, autor, dependencias y más.
- `insmod`: Carga un módulo en el kernel. Es necesario proporcionar la ruta completa del archivo `.ko` (archivo de módulo). `sudo insmod /path/to/module.ko`.
- `modprobe`: Es la herramienta más utilizada para cargar o descargar módulos de manera inteligente, ya que se encarga de gestionar las dependencias entre módulos.
  - *Cargar* `sudo modprobe nombre_del_modulo`.
  - *Eliminar* `sudo modprobe -r nombre_del_modulo`.
  - *Listar módulos que se pueden cargar en el sistema* `modprobe -l`
- `rmod`: Elimina un módulo cargado en el kernel. A veces, se requiere que el módulo no esté en uso (por ejemplo, no debe estar en uso por ningún proceso o dispositivo).
- `depmod`: Genera o actualiza el archivo de dependencias de los módulos del kernel. Esto es necesario cuando se compilan nuevos módulos o se actualiza el kernel.
- `dmesg`: Muestra los mensajes del buffer del kernel, donde se registran las actividades de carga y descarga de módulos. Es útil para ver si un módulo se cargó correctamente o si hubo algún error.

14. ¿Qué es un parche del kernel? ¿Cuáles son las razones principales por las cuáles se deberían aplicar parches en el kernel? ¿A través de qué comando se realiza la aplicación de parches en el kernel?

Un **parche del kernel** es un mecanismo que permite aplicar actualizaciones sobre una versión base. Se basa en *archivos diff* (archivos de diferencia), que indican qué agregar y qué quitar.

Las **razones principales** por las cuales se deberían aplicar parches en el kernel son:

- **Agregar funcionalidad**, como nuevos drivers o correcciones menores.
- A veces puede resultar más sencillo descargar y aplicar el archivo de diferencia en vez de descargar todo el código de la nueva versión.

La **aplicación de parches** en el kernel se realiza a través del comando `patch`. Un ejemplo de esto sería `$ cd linux; xzcat ../patch-6.13.7.xz | patch -p1`.

- En ese ejemplo se utiliza `xzcat` para descomprimir un archivo de parche llamado `patch-6.13.7.xz` y la salida se dirige al comando `patch` con la opción `-p1`. La opción `-p1` le indica a `patch` que ignore el primer componente de la ruta en los nombres de archivo dentro del *archivo diff*.
- Un parámetro útil para el comando patch es `--dry-run`, el cual permite simular la aplicación del parche sin realizar cambios reales.

15. Investigue la característica Energy-aware Scheduling incorporada en el kernel 5.0 y explique brevemente con sus palabras:

- ¿Qué característica principal tiene un procesador ARM big.LITTLE?

La característica principal de un procesador **ARM big.LITTLE** es su **arquitectura de núcleos de alto rendimiento y núcleos de bajo consumo energético**, diseñados para trabajar juntos de manera eficiente. En este diseño, el **big** representa los núcleos de alto rendimiento (más potentes pero también más demandantes en términos de energía), mientras que el **LITTLE** hace referencia a los núcleos de bajo consumo energético (menos potentes pero mucho más eficientes energéticamente).

La idea es que el sistema puede utilizar los núcleos **big** cuando se requieren altos niveles de rendimiento, como en tareas intensivas (juegos, edición de video), y los núcleos **LITTLE** para tareas ligeras que no demandan mucha potencia de procesamiento, como navegación web o tareas de fondo. Esto ayuda a optimizar el consumo de energía y mejorar la eficiencia general del dispositivo.

- En un procesador ARM big.LITTLE y con esta característica habilitada. Cuando se despierta un proceso ¿a qué procesador lo asigna el scheduler?

Con la característica Energy-aware Scheduling (EAS) habilitada en el kernel 5.0, el scheduler tiene en cuenta no solo la carga del procesador, sino también el consumo de energía y la eficiencia de los núcleos. Cuando un proceso se despierta, el scheduler de Linux en sistemas ARM big.LITTLE decide asignar el proceso al núcleo de bajo consumo energético (LITTLE) si el proceso no requiere un rendimiento alto. Esto ayuda a ahorrar energía al evitar el uso innecesario de los núcleos de alto rendimiento (big).

Sin embargo, si el proceso requiere un mayor rendimiento, el scheduler puede optar por asignarlo a un núcleo de alto rendimiento (big). Esta decisión depende de la carga de trabajo y de las políticas de gestión de energía que estén configuradas en el sistema.

- ¿A qué tipo de dispositivos opinás que beneficia más esta característica?

La característica de Energy-aware Scheduling (EAS) beneficia especialmente a dispositivos móviles como smartphones, tabletas, wearables y dispositivos IoT que utilizan procesadores ARM big.LITTLE. Estos dispositivos tienen una limitación de energía (debido a la batería) y se benefician enormemente de poder alternar entre núcleos de bajo consumo y de alto rendimiento según las necesidades del sistema.

> **Información** >
>
> Ver https://docs.kernel.org/scheduler/sched-energy.html

16. Investigue la system call `memfd_secret()` incorporada en el kernel 5.14 y explique brevemente con sus palabras

- ¿Cuál es su propósito?

La system call `memfd_secret()`, introducida en el kernel 5.14, tiene como propósito proporcionar una forma de crear regiones de memoria anónima (sin respaldo en archivos en disco) que están marcadas como secretas. Esta memoria es no intercambiable (non-swappable), lo que significa que no puede ser escrita en el espacio de intercambio del sistema (swap) ni ser volcada en el disco. Además, la memoria creada por esta system call está diseñada para garantizar privacidad y seguridad, protegiendo el contenido de la memoria de ser visible para otros procesos, incluso aquellos con privilegios elevados (por ejemplo, root).

- ¿Para qué puede ser utilizada?

La `memfd_secret()` puede ser utilizada principalmente para casos en los que se necesite manejar datos sensibles en la memoria, tales como claves criptográficas, datos de autenticación o cualquier otro tipo de información confidencial que no debe ser persistente en disco ni expuesta a otros procesos.

Al ser una región de memoria que no puede ser paginada a disco, es ideal para aplicaciones que requieren mantener información secreta y asegurar que no quede expuesta, ni siquiera en volúmenes de intercambio o archivos temporales.

- ¿El kernel puede acceder al contenido de regiones de memoria creadas con esta system call?

No, el kernel no puede acceder al contenido de la memoria creada mediante `memfd_secret()`. Esto es parte de las garantías de seguridad de la system call: la memoria marcada como "secreta" está protegida para que ni el kernel ni otros procesos puedan acceder a su contenido. Esta memoria se marca con las protecciones adecuadas para evitar que se voltee a disco (swap) o se acceda de otras maneras, garantizando la privacidad y confidencialidad de la información que contiene.

> **Información >**
>
> El siguiente artículo contiene bastante información al respecto:
> https://lwn.net/Articles/865256/

B - Ejercicio taller: Compilación del kernel Linux

El propósito de este ejercicio es que las y los estudiantes comprendan los pasos básicos del proceso de compilación del kernel de GNU/Linux.

Si bien esta práctica es guiada es aconsejable que las y los alumnas/os investiguen las distintas opciones y comandos utilizados.

Para la realización de este taller compilaremos la versión 6.13.7 del kernel Linux. Pero en lugar de descargar la versión deseada descargaremos la 6.7 y la actualizaremos a 6.8 mediante la aplicación un parche (patch) a modo de práctica.

Compilaremos un kernel Linux con las siguientes funcionalidades:

- Soporte para sistemas de archivos `BTRFS`.
- Soporte para la utilización de dtar ispositivos de bloques loopback.

> **Nota - Máquina virtual >**
>
> La cátedra provee una máquina virtual que ya cuenta con el software requerido para la compilación así cómo el código fuente del kernel, patch y el archivo btrfs.image.xz necesario para esta práctica. Estos archivos se encuentran en el directorio `/home/so/kernel`.

Se sugiere utilizar la máquina virtual para esta práctica a fin de evitar problemas de booteo si se configura mal el bootloader.

En caso de no utilizar la máquina virtual provista, se deberá realizar la instalación del software requerido para la instalación (librerías, compiladores, etc.)

Credenciales de acceso de la máquina virtual:

Tipo	Usuario	Contraseña
Normal	so	so
Administrador	root	toor

El usuario `so` no cuenta con privilegios para ejecutar `sudo` por lo cuál para ejecutar comandos que requieran estos privilegios será necesario ejecutar `su -` o `su -c comando arg ...`.

Nota - Ejemplos de la shell

En los comandos de ejemplo de esta práctica se verá que algunos comandos empiezan con `$` y otros con `#`. Estos símbolos representan el *prompt* del usuario y no deben escribirse cuando se copie el comando.

El símbolo `$` significa que el comando debe ejecutarse con un usuario sin privilegios. En el caso de la máquina virtual, es el usuario `so`. El símbolo `#` significa que el comando debe ejecutarse con privilegios usando el usuario `root`.

1. Descargue los siguientes archivos en un sistema GNU/Linux moderno, sugerimos descargarlo en el directorio `$HOME/kernel/` (donde $HOME es el directorio del usuario no privilegiado que uses):
    1. El archivo *btrfs.image.xz* publicado en la página web de la cátedra.
    2. El código fuente del kernel 6.13 (https://mirrors.edge.kernel.org/pub/linux/kernel/v6.x/linux-6.13.tar.xz).
    3. El parche para actualizar ese código fuente a la versión 6.13.7 (https://cdn.kernel.org/pub/linux/kernel/v6.x/patch-6.13.7.xz).
2. Preparación del código fuente:
    1. Posicionarse en el directorio donde está el código fuente y descomprimirlo:
    ```
    $ cd $HOME/kernel/
    $ tar xvf /usr/src/linux-6.13.tar.xz
    ```

Desde la VM yo tuve que tirar `$ tar xvf ./linux-6.13.tar.xz`

2. Emparchar el código para actualizarlo a la versión 6.8 usando la herramienta *patch*:
   `$ cd $HOME/kernel/linux-6.13`
   `$ xzcat /usr/src/patch-6.13.7.xz | patch -p1`

   Desde la VM yo tuve que tirar `$ xzcat ../patch-6.13.7 | patch -p1`

3. Pre-configuración del kernel:
   1. Usaremos como base la configuración del kernel actual, esta configuración por convención se encuentra en el directorio `/boot`. Copiaremos y renombraremos la configuración actual al directorio del código fuente con el comando:
      `$ cp /boot/config-$(uname -r) $HOME/kernel/linux-6.13/.config`
   2. Generaremos una configuración adecuada para esta versión del kernel con olddefconfig. olddefconfig toma la configuración antigua que acabamos de copiar y la actualiza con valores por defecto para las opciones de configuración nuevas.
      `$ cd $HOME/kernel/linux-6.13`
      `$ make olddefconfig`
   3. A fin de construir un kernel a medida para la máquina virtual usaremos a continuación localmodconfig que configura como módulos los módulos del kernel que se encuentran cargados en este momentos deshabilitando los módulos no utilizados. Es probable que make pregunte por determinadas opciones de configuración, si eso sucede presionaremos la tecla *Enter* en cada opción para que quede el valor por defecto hasta que make finalize.
      `$ make localmodconfig`
4. Configuración personalizada del kernel. Utilizaremos la herramienta menuconfig para configurar otras opciones. Para ello ejecutaremos:
   `$ make menuconfig`
   1. Habilitar las siguientes opciones para poder acceder a *btrfs.tar.xz*:
      1. File Systems → Btrfs filesystem support.
      2. Device Drivers → Block Devices → Loopback device support.
   2. Deshabilitar las siguientes opciones para reducir el tamaño del kernel y los recursos necesarios para compilarlo:
      1. General setup → Configure standard kernel features (expert users).
      2. Kernel hacking → Kernel debugging.

Tip - Uso de menuconfig >

La forma de movernos a través de este menú es utilizando las flechas del cursor, la tecla Enter y la barra espaciadora. La barra espaciadora permite decidir si la opción seleccionada será incluida en nuestro kernel, si será soportada a través de módulos, o bien si no se dará soporte a la funcionalidad (<*>, <M>, <> respectivamente).

Una vez seleccionadas las opciones necesarias, saldremos de este menú de configuración a través de la opción *Exit*, guardando los cambios.

5. Luego de configurar nuestro kernel, realizaremos la compilación del mismo y sus módulos.
    1. Para realizar la compilación deberemos ejecutar:
    `$ make -jX`

Tip - Sobre la compilación >

X deberá reemplazarse por la cantidad de procesadores con los que cuente su máquina. En máquinas con más de un procesador o núcleo, la utilización de este parámetro puede acelerar mucho el proceso de compilación, ya que ejecuta X jobs o procesos para la tarea de compilación en forma simultánea.

El comando *lscpu* permite ver la cantidad de CPUs y/o cores disponibles. **Recordar que la cantidad de CPUs de las máquinas virtuales es configurable.**

La ejecución de este último **puede durar varios minutos, o incluso horas** , dependiendo del tipo de hardware que tengamos en nuestra PC y las opciones que hayamos seleccionado al momento de la configuración.

Una vez finalizado este proceso, debemos verificar que no haya arrojado errores. En caso de que esto ocurra debemos verificar de qué tipo de error se trata y volver a la configuración de nuestro kernel para corregir los problemas.

Una vez cambiada la configuración tendremos que volver a compilar nuestro kernel. Previo a esta nueva compilación debemos correr el comando *make clean* para eliminar los archivos generados con la configuración vieja.

Tip - Configuración de la VM >

Configuren la VM para darle la mayor cantidad de Cores que puedan para que la compilación no tarde 5 siglos. En mi caso yo puse 4 cores y tardó 53 minutos.

6. Finalizado este proceso, debemos reubicar las nuevas imágenes en los directorios correspondientes, instalar los módulos, crear una imagen `initramfs` y reconfigurar nuestro gestor de arranque. En general todo esto se puede hacer de forma automatizada con los siguientes comandos.

```
$ make modules_install
$ make install
```

Tip - Instalación en otras distribuciones

En algunas distribuciones GNU/Linux el comando *make install* sólo instala la imagen del kernel pero no genera la imagen *initramfs* ni configura el gestor de arranque. En esos casos será necesario hacer esas tareas de forma manual.

Los comandos a utilizar varían de acuerdo a la distribución usada.

Tip - Instalación manual en distribuciones basadas en Debian

En caso de querer entender mejor el proceso de instalación en lugar de ejecutar *make install* es posible instalar la imagen del kernel manualmente de la siguiente forma:

```
# cp $HOME/kernel/linux-6.13/arch/x86_64/boot/bzImage\ /boot/vmlinuz-6.13.7
# cp $HOME/kernel/linux-6.13/System.map /boot/System.map-6.13.7
# cp $HOME/kernel/linux-6.13/.config /boot/config-6.13.7
# mkinitramfs -o /boot/initrd.img-6.13.7 6.13.7 # update-grub2
```

Ejecución

Yo tuve que tirar esos comandos con la sesión del root, para eso podemos cambiarnos a la sesión usando `su root`

7. Como último paso, a través del comando `reboot`, reiniciaremos nuestro equipo y probaremos el nuevo *kernel* recientemente compilado.
    1. En el gestor de arranque veremos una nueva entrada que hace referencia al nuevo *kernel*. Para *bootear*, seleccionamos esta entrada y verificamos que el sistema funcione correctamente.
    2. En caso de que el sistema no arranque con el nuevo *kernel*, podemos reiniciar el equipo y *bootear* con nuestro *kernel* anterior para corregir los errores y realizar una nueva compilación.
    3. Para verificar qué kernel se está ejecutando en este momento puede usar el comando: `$ uname -r`

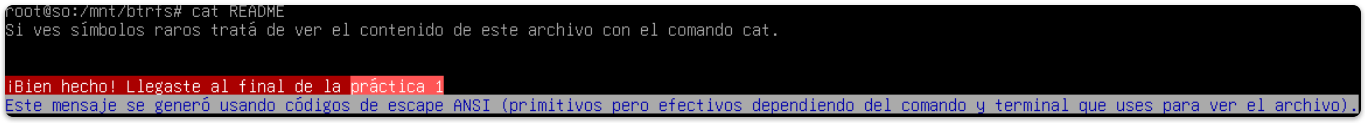
C - Poner a prueba el kernel compilado

btrfs.image.xz es un archivo de 110MiB formateado con el filesystem `BTRFS` y luego comprimido con la herramienta `xz`. Dentro contiene un script que deberás ejecutar en una máquina con acceso a Internet (puede ser la máquina virtual provista por la cátedra) para realizar la entrega obligatoria de esta práctica.

Para acceder al script deberás descomprimir este archivo y montarlo como si fuera un disco usando el driver "Loopback device" que habilitamos durante la compilación del kernel.

Usando el kernel 6.13.7 compilado en esta práctica:

1. Descomprimir el filesystem con:
   `$ unxz btrfs.image.xz`
2. Verificaremos que dentro del directorio */mnt* exista al menos un directorio donde podamos montar nuestro pseudo dispositivo. Si no existe el directorio, crearlo. Por ejemplo podemos crear el directorio */mnt/btrfs/*.
3. A continuación montaremos nuestro dispositivo utilizando los siguientes comandos:
   `$ su -`
   `# mount -t btrfs -o loop $HOME/btrfs.image /mnt/btrfs/`
4. Diríjase a */mnt/btrfs* y verifique el contenido del archivo README.md.



Nota para el uso

Yo tuve que crear la carpeta `/mnt/btrfs/` con el root porque la carpeta `/mnt/` estaba vacía en mi caso.
Para hacer el mount desde la sesión del *root* yo tuve que hacer `mount -t btrfs -o loop /home/so/kernel/btrfs.image /mnt/btrfs/`