Práctico nro 1 SO

1. ¿Qué es GCC?

GCC (GNU Compiler Collection) es un conjunto de compiladores de código abierto desarrollado por GNU. Soporta múltiples lenguajes como C, C++, Fortran, Ada, Go y D. Es multiplataforma, altamente optimizable y compatible con estándares modernos. Se usa principalmente en entornos Unix-like y es esencial en desarrollo de software. Ejemplo básico:

```
gcc -o programa programa.c
```

Compila `programa.c` y genera el ejecutable `programa`.

2. ¿Qué es make y para que se usa?

`make` es una herramienta de automatización que simplifica la compilación de proyectos de software. Lee un archivo `Makefile` que define reglas con objetivos, dependencias y comandos. Automatiza la compilación, recompilando solo lo necesario cuando hay cambios, lo que ahorra tiempo en proyectos grandes. Ejemplo básico de `Makefile`:

```
programa: main.c funciones.c
    gcc -o programa main.c funciones.c
```

Uso: Ejecutar `make` en la terminal compila el proyecto según las reglas del `Makefile`. Es esencial para gestionar proyectos complejos.

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 preinstaladas en la VM):

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

No, si tiro make y luego make lo que ocurre es que make detecta que no cambió nada y evita recompilar.

b. 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, se volvieron a compilar porque usa la última fecha de modificación para saber si algo cambió o no.

c. ¿Por qué “run” es un target “phony”?

“Phony” se usa para aquellos que no generen archivos, por ejemplo run, que ejecuta el programa.

d. 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?

En el ejemplo, la regla para `dlinkedlist.o` no define comandos explícitos para generarlo, pero el programa se compila correctamente porque `make` utiliza **reglas implícitas**. Estas reglas predefinidas asumen que existe un archivo `dlinkedlist.c` (con el mismo nombre base) y aplican automáticamente el comando:

```
gcc -c dlinkedlist.c -o dlinkedlist.o
```

Además, la dependencia de `dlinkedlist.h` asegura que, si el archivo de cabecera cambia, `make` recompilará `dlinkedlist.o` para mantener la consistencia. Esto permite compilar correctamente sin necesidad de especificar comandos explícitos en la regla.

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

El kernel Linux es un **kernel monolítico modular**. Esto significa que todas las funcionalidades del sistema operativo se ejecutan en el espacio del kernel, pero permite cargar módulos dinámicamente para extender sus capacidades.

Características principales:

1. **Monolítico**: Eficiente, con comunicación rápida entre componentes.
2. **Modular**: Permite cargar/descargar módulos (ej: controladores) en tiempo de ejecución.

3. **Portable**: Escrito en C, con abstracción de hardware, soporta múltiples arquitecturas (x86, ARM, etc.).
4. **Capas**: User Space, System Call Interface, Kernel Space (gestión de memoria, procesos, dispositivos) y Hardware.

Pros:

- **Eficiencia**: Comunicación rápida al ser monolítico.
- **Flexibilidad**: Módulos permiten adaptarse sin reiniciar.
- **Portabilidad**: Soporta múltiples plataformas.

Contras:

- **Complejidad**: Al ser monolítico, es grande y complejo.
- **Inestabilidad potencial**: Un error en un módulo puede afectar todo el sistema.
- **Menos modular que un microkernel**: Menos aislamiento entre componentes.

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

El versionado de los kernels Linux sigue un esquema de tres números (desde 2005): X.Y.Z, donde:

1. X: Número de versión principal (cambia con grandes hitos o cambios significativos).
2. Y: Número de versión secundaria.
    - Par: Versión estable (ej: 6.2, 6.4).
    - Impar: Versión de desarrollo (ya no se usa, era común antes de 2005).
3. Z: Número de revisión (parches y correcciones de errores).

Ejemplo:

- 6.4.10: Versión 6, estable (4), con 10 parches aplicados.

Características adicionales:

- LTS (Long-Term Support): Versiones con soporte extendido para uso en producción.
- Frecuencia: Nuevas versiones estables cada 2-3 meses.

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

Un usuario puede recompilar el kernel en GNU/Linux por los siguientes motivos:

1. Optimización: Ajustar el kernel para mejorar el rendimiento en hardware específico.
2. Soporte de hardware: Habilitar controladores o funcionalidades no incluidas en el kernel predeterminado.

3. Reducción de tamaño: Eliminar módulos innecesarios para sistemas con recursos limitados.
4. Parches personalizados: Aplicar modificaciones o parches no oficiales para necesidades específicas.
5. Actualización manual: Usar una versión más reciente del kernel que no está disponible en los repositorios.
6. Investigación y aprendizaje: Entender el funcionamiento interno del kernel o realizar experimentos.
7. Seguridad: Habilitar o deshabilitar características de seguridad según requerimientos.

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.

Opciones para configurar la compilación del kernel Linux:

1. make config:
    - Interfaz: Línea de comandos, pregunta cada opción.
    - Ventaja: Simple, sin dependencias adicionales.
    - Desventaja: Tedioso y lento.
2. make menuconfig:
    - Interfaz: Menús en modo texto (usa ncurses).
    - Ventaja: Más eficiente que make config.
    - Desventaja: Requiere familiaridad con las opciones.
3. make xconfig:
    - Interfaz: Gráfica (basada en Qt).
    - Ventaja: Intuitiva y visual.
    - Desventaja: Necesita entorno gráfico y Qt.
4. make gconfig:
    - Interfaz: Gráfica (basada en GTK).
    - Ventaja: Similar a xconfig, pero con GTK.
    - Desventaja: Necesita entorno gráfico y GTK.
5. make oldconfig:
    - Interfaz: Actualiza una configuración existente.
    - Ventaja: Ideal para actualizar kernels.
    - Desventaja: No es útil para configuraciones desde cero.
6. make defconfig:
    - Interfaz: Configuración predeterminada para la arquitectura.
    - Ventaja: Rápido y funcional.

    - Desventaja: No personalizado.
7. make localmodconfig:
    - Interfaz: Configura basada en módulos cargados.
    - Ventaja: Reduce el tamaño del kernel.
    - Desventaja: Puede omitir módulos necesarios.

Resumen:

- Texto: menuconfig (eficiente) o config (simple pero lento).
- Gráfico: xconfig (Qt) o gconfig (GTK).
- Automático: oldconfig (actualizaciones), defconfig (rápido), localmodconfig (mínimo).

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

a. make menuconfig

Tarea: Abre una interfaz de menús en modo texto para configurar las opciones del kernel. Genera o actualiza el archivo .config.

b. make clean

Tarea: Limpia los archivos generados durante la compilación (objetos, binarios, etc.), dejando solo el código fuente y el archivo .config.

c. make (investigue la funcionalidad del parámetro -j)

Tarea: Compila el kernel. El parámetro -j especifica el número de trabajos paralelos (ej: -j4 usa 4 núcleos de CPU), acelerando la compilación.

d. make modules (utilizado en antiguos kernels, actualmente no es necesario)

Tarea: Compilaba solo los módulos del kernel. Actualmente, make compila todo (kernel y módulos) en un solo paso.

e. make modules_install

Tarea: Instala los módulos compilados en el directorio /lib/modules/<versión-del-kernel>, listos para ser usados.

f. make install

Tarea: Instala el kernel compilado, copiando la imagen del kernel (vmlinuz) y el archivo de configuración a /boot, y actualiza el gestor de arranque (GRUB, LILO, etc.).

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?

- Ubicación inicial: /arch/<arquitectura>/boot/.
- Reubicación: /boot/vmlinuz-<versión-del-kernel>.
- Comando automático: make install (copia la imagen y configura el gestor de arranque).

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

- initramfs: Sistema de archivos temporal en RAM para preparar el arranque.
- Funcionalidad: Carga módulos, monta el sistema raíz y ejecuta scripts de inicialización.
- No necesario: Si el kernel tiene todos los controladores integrados y el sistema raíz es simple.

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

Si no se configura es imposible para que el gestor de arranque reconozca la imagen y la cargue, para esto se debe configurar, por ejemplo con update-grub.

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

Un módulo es un código cargable dinamicamente que extiende funcionalidades del kernel.

Comandos:

- Ver módulos cargados: lsmod.
- Cargar módulo: insmod o modprobe.
- Descargar módulo: rmmod o modprobe -r.
- Información del módulo: modinfo.
- Generar dependencias: depmod.

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?

- Parche: Archivo con cambios para el código fuente del kernel.
- Razones: Corrección de errores, nuevas funcionalidades, optimizaciones y compatibilidad.
- Comando: patch -p1 < archivo_parche.patch.

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

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

Un procesador ARM big.LITTLE combina núcleos de alto rendimiento ("big") y núcleos de bajo consumo ("LITTLE") en un solo chip.

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

- Con Energy-aware Scheduling (EAS) habilitado, el scheduler asigna un proceso que se despierta al núcleo más eficiente en términos de energía, considerando:
    - La carga de trabajo del proceso.
    - El estado actual de los núcleos (activos, inactivos, frecuencia, etc.).
    - El consumo energético estimado.

c. ¿A qué tipo de dispositivos opinás que beneficia mas esta caracteristica?

Dispositivos móviles: Smartphones y tablets, donde la eficiencia energética es crítica para prolongar la duración de la batería.
Dispositivos embebidos: Sistemas IoT y wearables, que requieren bajo consumo y rendimiento equilibrado.

Servidores y edge computing: En entornos con restricciones de energía o que priorizan la eficiencia.

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

a. ¿Cuál es su propósito?

memfd_secret() permite crear regiones de memoria que son inaccesibles incluso para el kernel. Estas regiones están diseñadas para almacenar datos sensibles (como claves criptográficas) de manera segura, evitando que otros procesos o el propio kernel puedan leerlos.

b. ¿Para qué puede ser utilizada?

Protección de datos sensibles: Ideal para aplicaciones que manejan información crítica, como:

- Claves criptográficas.
- Contraseñas.
- Datos personales o financieros.

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

No, el kernel no puede acceder al contenido de las regiones de memoria creadas con memfd_secret(). Estas regiones están diseñadas para ser completamente privadas, incluso para el kernel, lo que las hace extremadamente seguras contra accesos no autorizados.

Compilación del kernel

1. Descarga, los archivos ya vinieron con la máquina virtual.

2. Preparación del código fuente:

a. Posicionarse en el directorio donde está el código fuente y descomprimirlo:

```
$ cd $HOME/kernel/
$ tar xvf /usr/src/linux-6.13.tar.xz
```

b. Emparchar el código para actualizarlo a la versión 6.8 usando la herramienta patch:

```
$ cd $HOME/kernel/linux-6.13
$ xzcat ../patch-6.13.7.xz | patch -p1
```

3. Pre-configuración del kernel:

a. 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
```

b. 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
```

c. A fin de construir un kernel a medida para la máquina virtual usaremos a 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
```

- Deshabilitamos la firma criptográfica del kernel: Cryptographic API → Certificates for signature checking → Additional X.509 certificates for signature checking.
- Habilitamos soporte para el sistema de archivos btrfs. Fyle Systems —> Btrfs filesystem support.
- Soporte para dispositivos de Lookback: Device Drivers → Block Devices —> Loopback device support

5. Luego de configurar nuestro kernel, realizaremos la compilación del mismo y sus módulos.

a. Para realizar la compilación deberemos ejecutar:

```
$ make -jX
```

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
```

Para esto tuve que pasarme a root porque no tenía permisos.
No lo menciona, pero debe ejecutar update-grub2 para que grub se actualize.

7. Como último paso, a través del comando reboot, reiniciaremos nuestro equipo y probaremos el nuevo kernel recientemente compilado.

a. 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.

b. 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.

c. Para verificar qué kernel se está ejecutando en este momento puede usar el comando:

```
$ uname -r
```

Prueba del kernel

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/
```

En mi caso lo hice como root entonces use home/kernelbtrfs.image para la ruta de la imagen

4. Diríjase a /mnt/btrfs y verifique el contenido del archivo README.md.