



# SEGURIDAD, PROCESOS Y ZÓCALOS

## AA1. GESTIÓN DE PROCESOS

Boj Cobos, Daniel  
DAM  
04/2023

<b>INSTALACIÓN DEL ENTORNO DE DESARROLLO EN UNA VM DEBIAN</b>	<b>3</b>
<b>Instalación de Debian como Máquina Virtual</b>	<b>3</b>
<b>Configuración de la carpeta compartida entre el host y la VM</b>	<b>4</b>
<b>Instalación personalizada con configuración y personalización del Entorno de Desarrollo</b>	<b>4</b>
Instalación mínima de Debian	5
Instalación de los paquetes de software	6
Instalación de Qtile	6
Instalación de software adicional mediante Snap	7
Configuración del script xinit	7
Inicio del display server	7
<b>Opcional: Cambiar la versión de Python 3</b>	<b>8</b>
<b>Opcional: Modo transparente</b>	<b>8</b>
<b>Instalación de GuestAdditions</b>	<b>9</b>
<b>MONITOREO DE PROCESOS MEDIANTE EL COMANDO PSTREE</b>	<b>9</b>
<b>Ejercicio: Ejecutar el comando pstree para mostrar el árbol de procesos en ejecución marcando los procesos actuales junto con su PID.</b>	<b>11</b>
<b>LISTADO DE PROCESO CON EL COMANDO PS</b>	<b>11</b>
<b>Ejercicio: Comprobar usando el comando ps con las opciones -la que los procesos son padre e hijo.</b>	<b>12</b>
<b>Ejercicio: Inicio, seguimiento y terminación de un proceso.</b>	<b>13</b>
<b>RECOMENDACIÓN ADICIONAL</b>	<b>13</b>
<b>ANÁLISIS DE PROGRAMA PYTHON</b>	<b>14</b>
<b>Funcionalidad del programa</b>	<b>14</b>
<b>Aspectos clave</b>	<b>14</b>
os.pipe	14
os.fork	15
Bloque condicional	15
exit(código)	15
<b>Análisis pormenorizado del Código</b>	<b>15</b>
Ejecución del código con las nuevas implementaciones	18

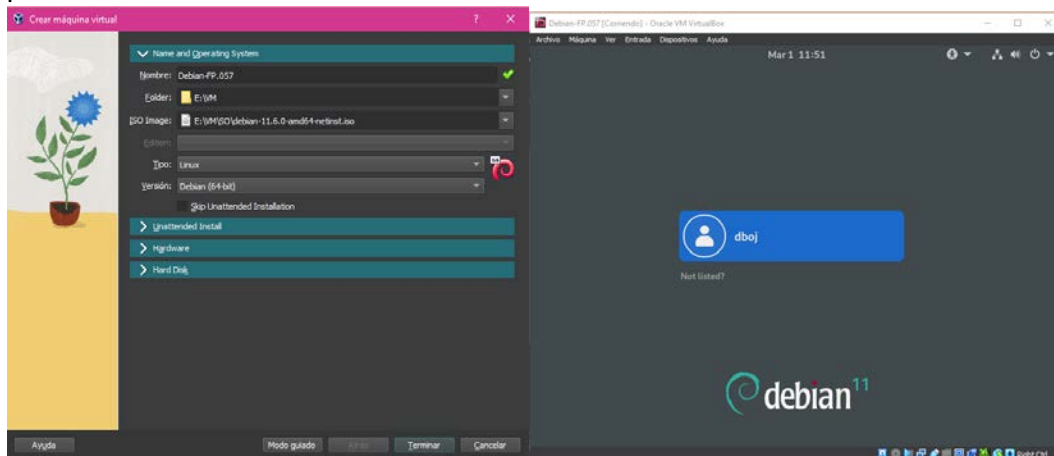
<b>ANEXO 1. INTRODUCCIÓN A LAS LIBRERÍAS</b>	<b>18</b>
<b>MÓDULO OS</b>	<b>18</b>
Importación	18
Métodos más comunes para el manejo de directorios	18
Acceso a variables de entorno	19
Acceso a información de sobre los procesos del sistema	19
Nombres de rutas y archivos	20
Acceso y manejo de archivos	20
Pipes	20
<b>MÓDULO SYS</b>	<b>21</b>
Importación	21
Funcionalidades principales	21
<b>RECURSOS</b>	<b>21</b>

# AA1. GESTIÓN DE PROCESOS

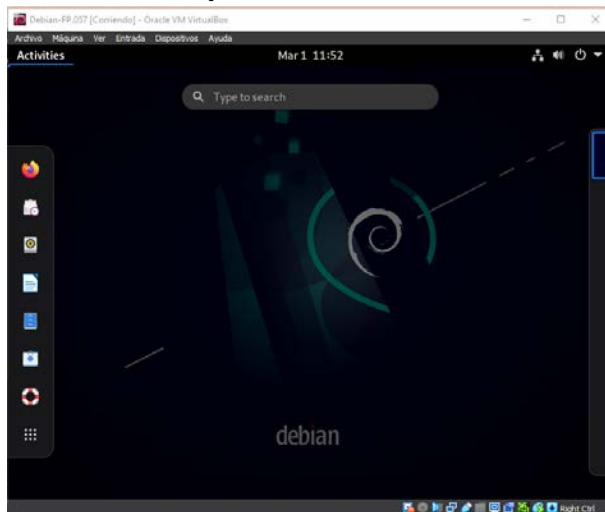
## Instalación del entorno de Desarrollo en una VM Debian

### Instalación de Debian 11 como Máquina Virtual

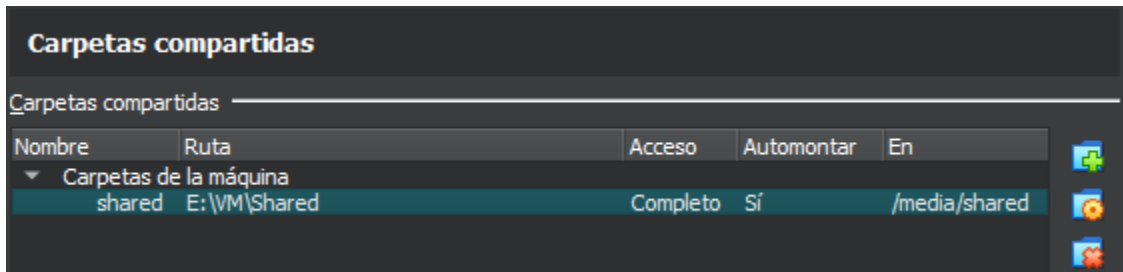
Para una instalación básica por defecto de Debian, la versión 7 de VirtualBox incluye una opción de instalación desatendida que nos permitiría dejar preparada la VM sin mayor intervención por nuestra parte que la configuración de la máquina previa a la instalación. Podemos usarlo siempre y cuando no necesitemos una configuración especial de las particiones.



### VM de Debian por defecto en funcionamiento



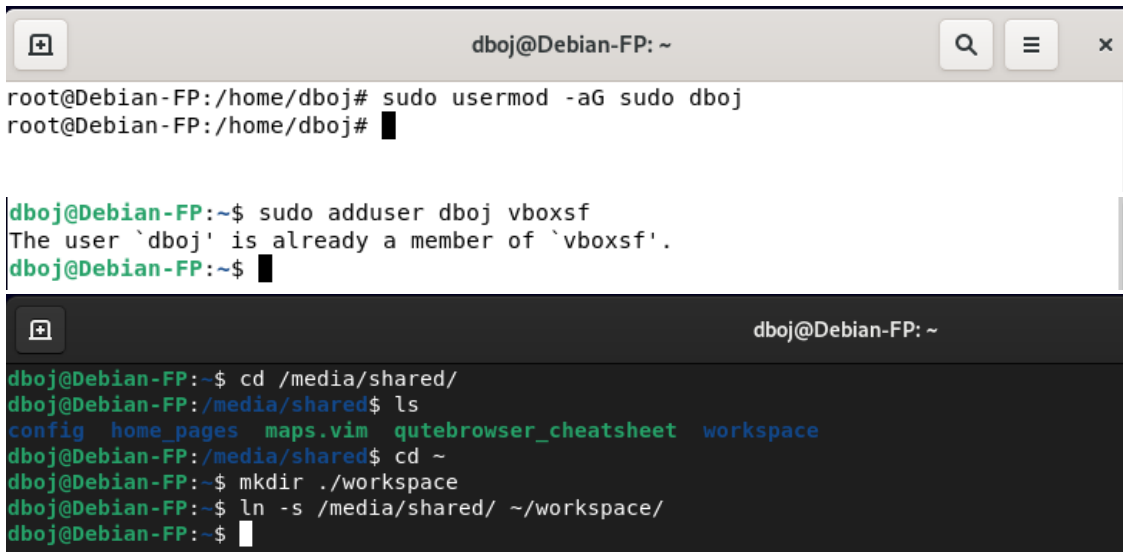
## Configuración de la carpeta compartida entre el host y la VM



**Carpetas compartidas**

Carpetas compartidas

Nombre	Ruta	Acceso	Automontar	En
▼ Carpetas de la máquina				
shared	E:\VM\Shared	Completo	Sí	/media/shared

```

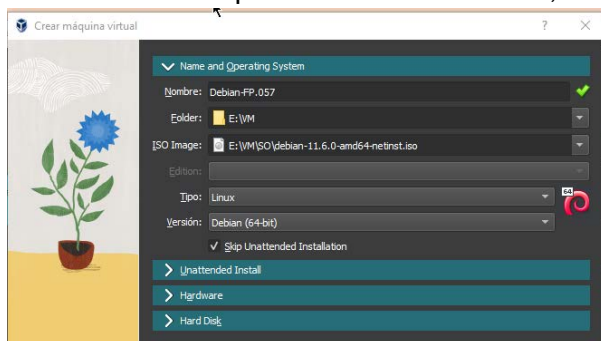
root@Debian-FP:/home/dboj# sudo usermod -aG sudo dboj
root@Debian-FP:/home/dboj# 

dboj@Debian-FP:~$ sudo adduser dboj vboxsf
The user `dboj' is already a member of `vboxsf'.
dboj@Debian-FP:~$ 

dboj@Debian-FP:~$ cd /media/shared/
dboj@Debian-FP:/media/shared$ ls
config  home_pages  maps.vim  qutebrowser_cheatsheet  workspace
dboj@Debian-FP:/media/shared$ cd ~
dboj@Debian-FP:~$ mkdir ./workspace
dboj@Debian-FP:~$ ln -s /media/shared/ ~/workspace/
dboj@Debian-FP:~$
  
```

## Instalación personalizada con configuración y personalización del Entorno de Desarrollo

En este caso, debemos seleccionar la opción de *Omitir Instalación desatendida* al crear nuestra VM. Aunque bastan 2GB de RAM, he escogido 8, 20GB de disco y dos núcleos.



Aunque la opción anterior sea perfectamente válida, un usuario con ganas de aprender sobre Linux preferirá realizar una instalación a medida y aprovechar la absoluta libertad que ofrecen estos sistemas Unix. Hay que recordar que Linux nació con la intención de ser una opción Open Source totalmente configurable, adaptable y personalizable y que es por ello que un usuario avanzado podrá aprovechar al máximo este sistema.

En mí caso, quiero partir de una instalación mínima, únicamente el kernel del SO, y a partir de ahí preparar un entorno basado en el Window Manager tipo tile Qtile en lugar de usar un Escritorio. Con este tipo de instalación, partiremos desde un SO con únicamente acceso a terminal y, desde esta, configuraremos e instalaremos los paquetes iniciales para tener nuestro SO totalmente a medida.

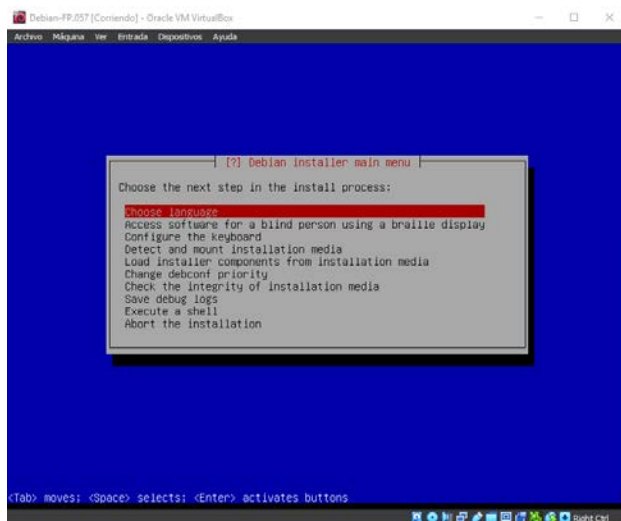
Una vez finalizada la instalación de los paquetes básicos y el software necesario para ejecutar el WM, instalaremos Qtile y accederemos al sistema mediante el display server, usaremos Xorg, una vez iniciado el WM, acabaremos de personalizar nuestro entorno instalando y configurando el resto de software.

Hemos de tener en cuenta que, en este tipo de instalaciones, hemos de ir añadiendo paquetes al momento de irlos necesitando, por ejemplo, para esta actividad, he tenido que instalar *psmisc* manualmente para poder usar *pstree*.

## Instalación mínima de Debian

Comenzaremos por, una vez configurada nuestra VM, cargar la iso de Debian y realizar escoger la opción de *instalación avanzada* para dejar únicamente instalado el *kernel* del SO. De este modo, podremos personalizar completamente a nuestro gusto nuestro sistema.

### GUI de instalación avanzada

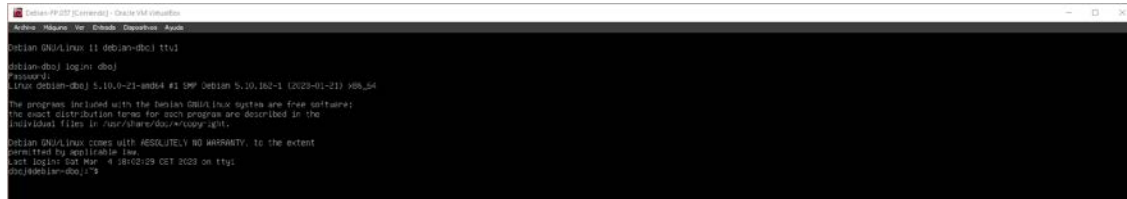


Durante el proceso de instalación, escogeremos dividir nuestro disco en particiones independiente para el boot, root y home y podemos crear una partición *swap*. Además, el mismo instalador nos solicitará si queremos crear una clave de *root* y si queremos crear un usuario, opción que escogeremos para generar nuestro usuario ya que en Linux es buena práctica tener siempre separados el usuario *root* de cualquier otro tipo de usuario, aunque sean administradores de alto nivel.

Por otro lado, cuando nos pregunte que servicios queremos instalar, deseleccionaremos todas las opciones ya que no queremos instalar ningún gestor de escritorio.

Una vez finalizada la instalación, al reiniciar el equipo, veremos cómo se ejecuta el sistema en modo terminal y este nos pide identificarnos. Para este tipo de entorno, he decidido no instalar ningún tipo de gestor de logeo y realizar únicamente el logeo por terminal.

## Instalación mínima con acceso a terminal



## Instalación de los paquetes de software

Basaré mi Entorno de Desarrollo en un gestor de ventanas de tipo tile, *Qtile*; además, usaré *Kitty* como terminal, ya que es muy rápida y personalizable, y, aunque instale también VSCode como solicita la actividad, usaré *NeoVim* como editor de texto principal.

Además, y para poder instalar y ejecutar el entorno, es necesario instalar una serie de paquetes de software, controladores y librerías. Entre ellos, gestores de descargas, compresores, *Python3* y algunas de sus librerías (*Qtile* está desarrollado en *Python3*), algunos controladores gráficos y de audio, un gestor de buffers y programas para realizar la personalización del entorno. Entre el software a instalar, es muy importante instalar un display server para poder ejecutar nuestro entorno gráfico, en este caso se usa *Xorg*, es muy importante instalar también *xinit* para poder generar el script de entrada al display server. Además, instalamos algunas librerías de cabeceras de Linux imprescindibles o muy útiles.

## La lista de paquetes instalados es la siguiente:

“neofetch git wget slapd curl make zip unzip neovim nano kitty compton libxft-dev libxcb1-dev python3 python3-pip python3-gi python3-gi-cairo python-dbus-dev libpangocairo-1.0-0 alsa-utils pavucontrol pcmanfm qutebrowser suckless-tools tmux nitrogen zsh xorg xserver-xorg locate gcc xinit dkms build-essential linux-headers-generic linux-headers-amd64 nodejs npm”

Para instalarlos usamos un comando **\$ sudo apt install <paquete> -yy**.

Por último, podemos instalar algunas fuentes especialmente diseñadas para usar en el terminal:

**\$ sudo apt install fonts-font-awesome fonts-jetbrains-mono**

## Instalación de Qtile

A continuación, usamos el gestor de paquetes de Python, *pip3*, para instalar *Qtile* y las librerías de Python necesarias para ejecutarlo. Para ello, ejecutamos el comando:

## \$ Pip3 install pynvim xcffib cairocffi qtile

Nótese que, se ha aprovechado para instalar también la librería *pynvim* que actualiza *NeoVim* a Python3.

## Instalación de software adicional mediante Snap

Para no tener que instalar *keyrings* y nuevos repositorios en APT, podemos instalar VSCode directamente desde el gestor *Snap* que, en previsión, instalamos en la primera carga de software.

Ejecutamos:

## \$ sudo snap install code classic

Una vez instalado todo el software, realizamos un reboot del sistema para poder ejecutar Qtile.

## Configuración del script xinit

Antes de entrar a nuestro entorno gráfico, realizaremos la configuración del archive script que sirve para que *Xorg* inicie el WM indicándole que ejecute *Qtile*.

1. En el directorio raíz de nuestro home editamos el archivo oculto *.xinitrc*:  
\$ nvim ~/.xinitrc
2. Añadimos la siguiente línea de Código:  
nitrogen --restore & compton & qtile start

Nitrogen → Forma parte del software de personalización, permite gestionar fondos de pantalla en Qtile. Usamos la opción *--restore* para que recupere el fondo que hayamos configurado/. Compton → Es un *compositor* para servidores X de Linux. Permite que apliquemos efectos a nuestras ventanas como la transparencia.

Qtile start → Es el comando para iniciar Qtile.

## Inicio del display server

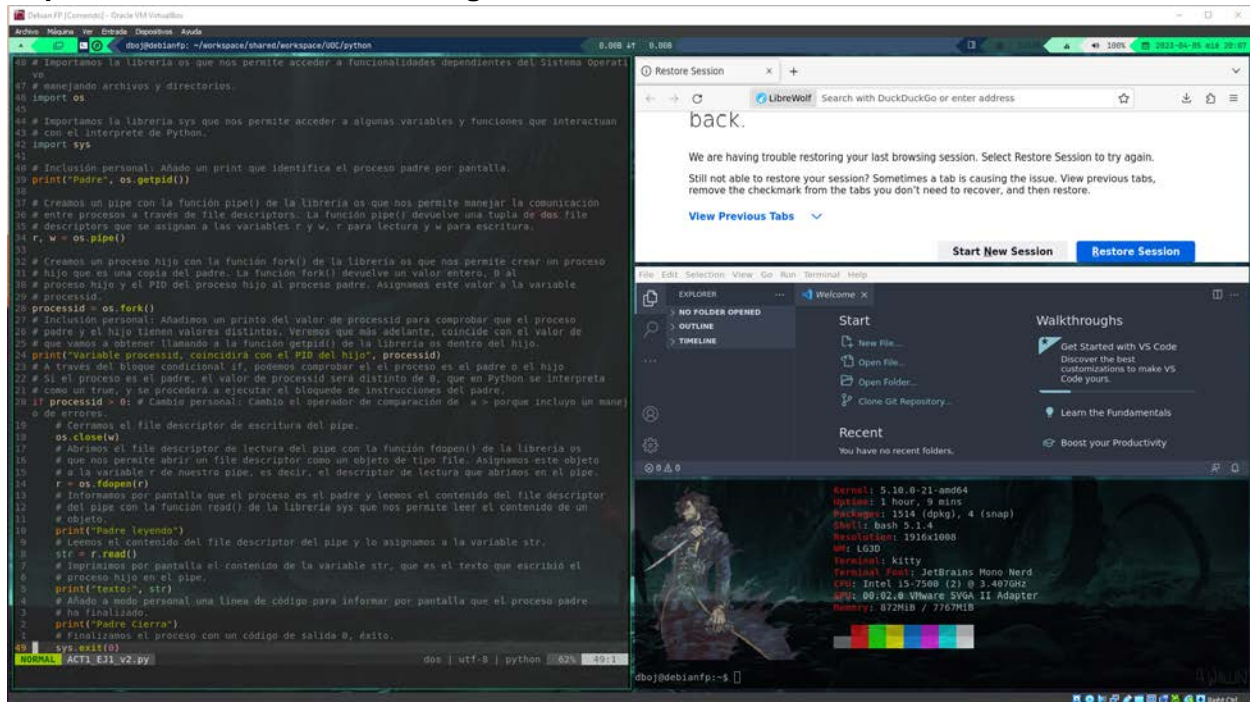
Una vez terminada la configuración inicial, entramos al WM ejecutando:

## \$ startx

A continuación, podremos realizar las distintas configuraciones del software instalado para personalizar nuestro entorno y para prepararlo para el Desarrollo. Entre todas, la más importante es la de *NeoVim* ya que convierte al editor en un auténtico IDE. Tengo las distintas configuraciones disponibles en mi GitHub personal, en el repositorio ***personal-dot-configs***.

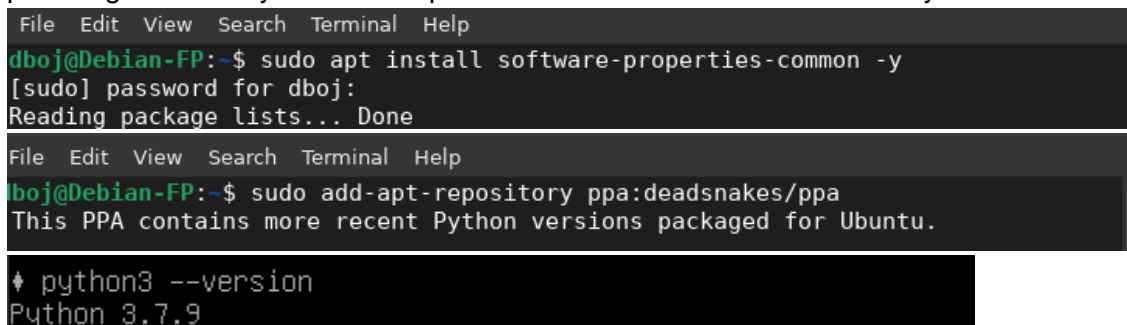


## Aspecto de Debian una vez configurado el entorno



## Opcional: Cambiar la versión de Python 3

Para instalar la versión 3.7 de Python hay que habilitar la instalación de nuevos repositorios para el gestor APT y añadir el repositorio con todas las versiones de Python.

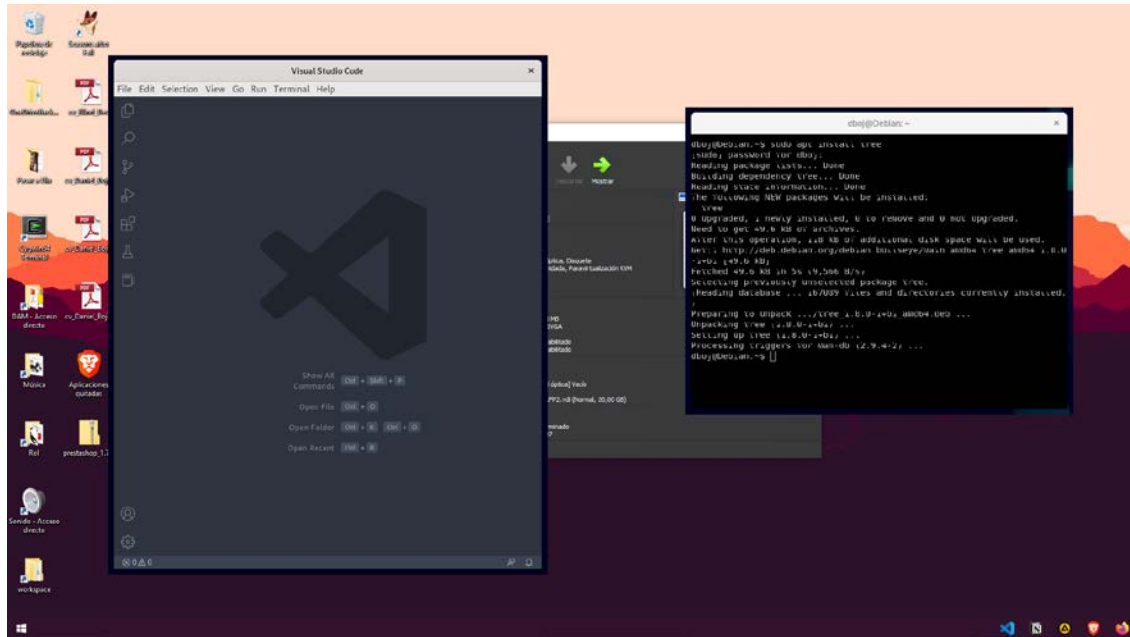


## Opcional: Modo transparente

Para poder ejecutar el modo transparente debemos haber instalado Debian 11 con un Desktop Manager en lugar de un Window Manager. Para demostrar cómo funciona, realicé una instalación estándar de Debian 11 a parte.

Para ejecutar el **modo fluido** es necesario configurar la máquina con Aceleración de Gráficos para poder usar GLX e iniciar Debian con un escritorio. Podemos entrar y salir del modo fluido con CTRL-L.

Vemos como en este modo se superponen las ventanas de nuestra VM en ejecución al escritorio de Windows del Host.



## Isntalación de GuestAdditions

1. Cargar la ISO en VBOX
2. Navegar a /media/cdrom
3. Instalar dependencias: `$ sudo apt-get install -y dkms build-essential linux-headers-generic linux-headers-amd64`
4. Cambiar a superusuario
5. Ejecutar → `$ bash ./VBoxLinuxAdditions.run`
6. Reiniciar el sistema.

## Monitoreo de procesos mediante el comando pstree

Antes de entrar en el análisis del comando, hay que indicar que en Linux, PID es el identificador único de un proceso en ejecución y PPID indica el ID del proceso padre, o sea, el proceso que lo ha iniciado.

El comando *ps*tree muestra todos los procesos en ejecución mediante una formato jerárquico en forma de árbol. Siendo siempre el proceso *init* el proceso raíz de la jerarquía o, si lo especificamos mediante las opciones del comando, el proceso indicado en la ejecución de *ps*tree. Para este ultimo escenario, debemos indicar el PID del proceso. Además, también podemos indicar un nombre de usuario para capturar solo los procesos iniciados por este.

**Sintaxis:**

**\$ pstree [OPCIONES] [PID] [USUARIO]**

```
systemd--ModemManager--2*[{ModemManager}]
--VBoxDRMClient--3*[{VBoxDRMClient}]
--VBoxService--8*[{VBoxService}]
--avahi-daemon--avahi-daemon
--cron
--dbus-daemon
--dhclient--3*[{dhclient}]
--kitty--bash--pstree
--9*[{kitty}]
--login--bash--startx--xinit--Xorg--9*[{Xorg}]
--sh--compton
--qtile--2*[{qtile}]
--polkitd--2*[{polkitd}]
--qutebrowser--QtWebEngineProc
--QtWebEngineProc--QtWebEngineProc--QtWebEngineProc--8*[{QtWebEngineProc}]
--QtWebEngineProc--4*[{QtWebEngineProc}]
--26*[{qutebrowser}]
--rsyslogd--3*[{rsyslogd}]
--rtkit-daemon--2*[{rtkit-daemon}]
--snapd--11*[{snapd}]
--systemd--(sd-pam)
--dbus-daemon
--pulseaudio--2*[{pulseaudio}]
--systemd-journal
--systemd-logind
--systemd-timesyn--{systemd-timesyn}
--systemd-udev
--udisksd--4*[{udisksd}]
--wpa_supplicant
dboj@debianfp:~$
```

Si queremos que aparezcan los PID de los procesos, debemos incluir la opción -p:

```
--login(566)--bash(772)--startx(913)--xinit(935)--Xorg(936)--{Xorg}(937)
--{Xorg}(938)
--{Xorg}(939)
--{Xorg}(940)
--{Xorg}(941)
--{Xorg}(942)
--{Xorg}(943)
--{Xorg}(944)
--{Xorg}(949)
--sh(950)--compton(952)
--qtile(953)--{qtile}(992)
--{qtile}(1009)
```

Y, como comentábamos, añadiendo el nombre de un usuario filtramos únicamente los procesos pertenecientes a este:

```
dboj@debianfp:~$ pstree -p
systemd(1)---ModemManager(588)---{ModemManager}(607)
--{ModemManager}(609)
--VBoxDRMClient(681)---{VBoxDRMClient}(697)
--{VBoxDRMClient}(698)
--{VBoxDRMClient}(699)
```

### Opciones comunes

A continuación se especifican algunas de las opciones más comunes en la ejecución del comando *pstree*.

- -p → Muestra los PID de los procesos.
- -c → Deshabilita la fusion de ramas idénticas.
- -t → Muestra los nombres completos de los hilos.
- -s PID → Muestra los procesos principales del proceso que pasamos como parámetro.
- -n → Ordena la lista de proceso por PID.
- -g → Muestra el PGID.
- -a → Muestra los argumentos CLI para ver cómo se inició la ejecución del proceso.
- -h → Resalta el proceso actual y todos sus ancestros.

- -H PID → Resalta el proceso que indicamos por parámetro.

## Ejercicio: Ejecutar el comando `pstree` para mostrar el árbol de procesos en ejecución marcando los procesos actuales junto con su PID.

Para realizar esta consulta usaremos el comando `pstree` con las opciones `-p` y `-h`, podremos, si quisiéramos perfilar aún más, añadir el nombre de usuario para filtrar únicamente los procesos activos que pertenecen al usuario.

### \$ `pstree -hp`

Añado capturas de pantalla de los elementos que quiero remarcar porque el árbol entero es muy grande. Veremos cómo, junto al nombre del proceso y entre paréntesis, se muestra el PID del proceso.

1. Init como proceso raíz en activo

```
dboj@debianfp:~$ pstree -hp
systemd(1)─ModemManager(588)─{ModemManager}(607)
                        └─{ModemManager}(609)
```

2. Terminal kitty como proceso en active

```
─kitty(18352)─bash(18366)─pstree(20427)
                        │
                        ├─{kitty}(18357)
                        ├─{kitty}(18358)
                        ├─{kitty}(18359)
                        └─{kitty}(18360)
```

3. Observamos que, en las capturas de pantalla, los procesos no activos aparecen sin resaltar, en un tono gris suave.
4. Mediante el operador `>>` escribimos la salida del comando en un archivo de texto que se mostrará en el Apéndice A.

```
dboj@debianfp:~$ pstree -hp >> ~/workspace/shared/pstree.txt
dboj@debianfp:~$
```

## Listado de proceso con el comando `ps`

El comando `ps` nos permite obtener información de los procesos que se están ejecutando en Linux. Este comando nos arroja una lista con todos los procesos en ejecución en el sistema. Aunque si lo ejecutamos sin opciones, obtendremos una salida que nos servirá de poco, ya que únicamente nos devolverá como procesos en ejecución el del `nash`, la terminal desde donde lo lanzamos, y del propio proceso `ps`:

```
dboj@debianfp:~$ ps
  PID TTY          TIME CMD
 18366 pts/0    00:00:00 bash
 24302 pts/0    00:00:00 ps
```

Para poder obtener información más útil, debemos añadir opciones en la ejecución del comando:

- -e y -A → Muestra todos los procesos en el sistema
- f → Muestra la jerarquía de procesos.
- -a → Muestra todos los procesos excepto los no asociados con una terminal.
- a → Muestra todos los procesos de todos los usuarios excepto los no asociados con una terminal.
- r → Solo muestra procesos en ejecución.
- x → Muestra todos los procesos, incluidos los que no controla tty.
- -p → Selecciona el PID o el PPID.
- -f → Muestra los datos en formato completo.
- -l → Muestra los datos en formato largo.

## Ejercicio: Comprobar usando el comando *ps* con las opciones -la que los procesos son padre e hijo.

Para ello, tendremos que lanzar el comando y comprobar los PID y PPID que hemos obtenido mediante el comando *ps*. En cuanto a las opciones, -l devuelve los datos con un formato largo, lo que aumenta enormemente la legibilidad del retoro y -a sirve para iterar sobre todos los procesos en ejecución del sistema que estén asociados con una terminal, si quisiéramos ver el conjunto total de procesos, deberíamos usar -e o -A.

Para filtrar la lista y realizar una identificación más rápida de los procesos, lanzaremos el comando con la opción -p PID1 PID2...PIDn para filtrar por las PID o PPID que nos interesan y f para ver más cómodamente las jerarquías de procesos.

```
dboj@debianfp:~$ ps f -lp 18352 18366
 F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
 0 R  1000    18352         1   6  80   0 - 252785 -      tty1        2:46 kitty
 0 S  1000    18366    18352   0  80   0 - 2214 -      pts/0        0:00 \_ /bin/bash
```

Podemos observar la relación jerárquica de 2 formas distinta.

1. Gracias a la opción f se muestra visualmente en la columna CMD mediante la inclusión del carácter \ que indica que el proceso es hijo del anterior.
2. Mediante la relación entre el PID de kitty y el PPID de /bin/bash.

Si lanzamos el comando *\$ps -la* sin filtrar por PID, deberemos navegar por la lista para ir comprobando las jerarquías. Para facilitar el proceso, lanzamos el comando con la opción f y con el pipe | less. De este modo, encontramos las entradas dónde podemos deducir del mismo modo que en ejemplo anterior, la relación entre procesos padre y procesos hijo:

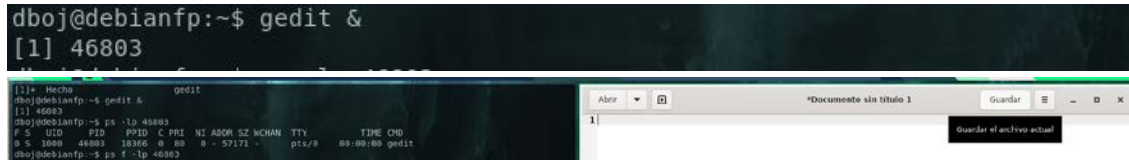
```
0 S  1000    18352         1   6  80   0 - 252785 -      tty1        2:58 kitty
0 S  1000    18366    18352   0  80   0 - 2214 -      pts/0        0:00 \_ /bin/bash
0 R  1000    41017    18366   0  80   0 - 2457 -      pts/0        0:00 \ ps f -lae
```



## Ejercicio: Inicio, seguimiento y terminación de un proceso.

1. En nuestro terminal, ejecutamos `$ gedit &`. Este comando nos abrirá un editor de texto, iniciando su ejecución desde terminal y, además, mediante `&`, nos mostrará el PID del proceso iniciado. Además, al ejecutar el comando, se nos abrirá la ventana con el editor de texto.

```
dboj@debianfp:~$ gedit &
[1] 46803
```



2. Ejecutamos el comando `$ ps -lp 46803` para comprobar cuál es el proceso padre de *gedit* y podemos ver que se trata del terminal desde donde lo ejecutamos, en este caso, *kitty*.

```
dboj@debianfp:~$ ps -lp 46803
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    46803    18366  0  80   0 - 57171 - pts/0      00:00:00 gedit
```

3. Comprobamos de nuevo añadiendo el PPID al comando y la opción `f`.

```
dboj@debianfp:~$ ps f -lp 18366 46803
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    18366    18352  0  80   0 - 2383 - pts/0      0:00 /bin/bash
0 S  1000    46803    18366  0  80   0 - 120620 - pts/0      0:00 \_ gedit
dboj@debianfp:~$
```

4. Finalmente, podemos terminar el proceso usando el comando `$ kill -9 PID`. Este comando, mediante la adición del *signal* 9, terminará el proceso indicado mediante el PID de forma abrupta y producirá con ello un *error fatal*. así, se trata de un comando que siempre terminara un proceso sea cuál sea su estado, pero que puede tener consecuencias inesperadas sobre nuestro sistema. Al ejecutarlo, simplemente veremos cómo se Cierra nuestra ventana del editor. Además, si intentamos monitorizar el PID del proceso, ya no aparecerá como active, sino que se nos informará de que ha sido terminado.

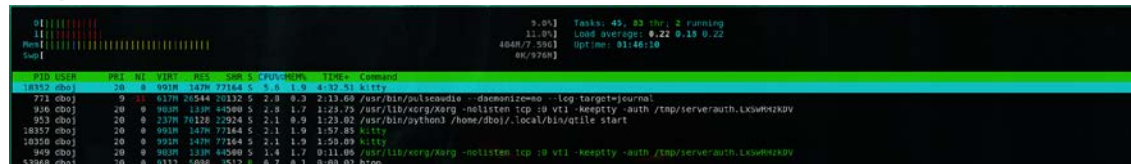
```
[1]+ Hecho gedit
dboj@debianfp:~$ gedit &
[1] 46803
dboj@debianfp:~$ ps -lp 46803
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    46803    18366  0  80   0 - 57171 - pts/0      00:00:00 gedit
dboj@debianfp:~$ ps f -lp 46803
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    18366    18352  0  80   0 - 2383 - pts/0      0:00 /bin/bash
0 S  1000    46803    18366  0  80   0 - 120620 - pts/0      0:00 \_ gedit
dboj@debianfp:~$ ps f -lp 18366 46803
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    18366    18352  0  80   0 - 2383 - pts/0      0:00 /bin/bash
0 S  1000    46803    18366  0  80   0 - 120620 - pts/0      0:00 \_ gedit
dboj@debianfp:~$ kill -9 46803
dboj@debianfp:~$
dboj@debianfp:~$ ps -lp 46803
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
[1]+ Terminado (killed) gedit
```

## Recomendación adicional

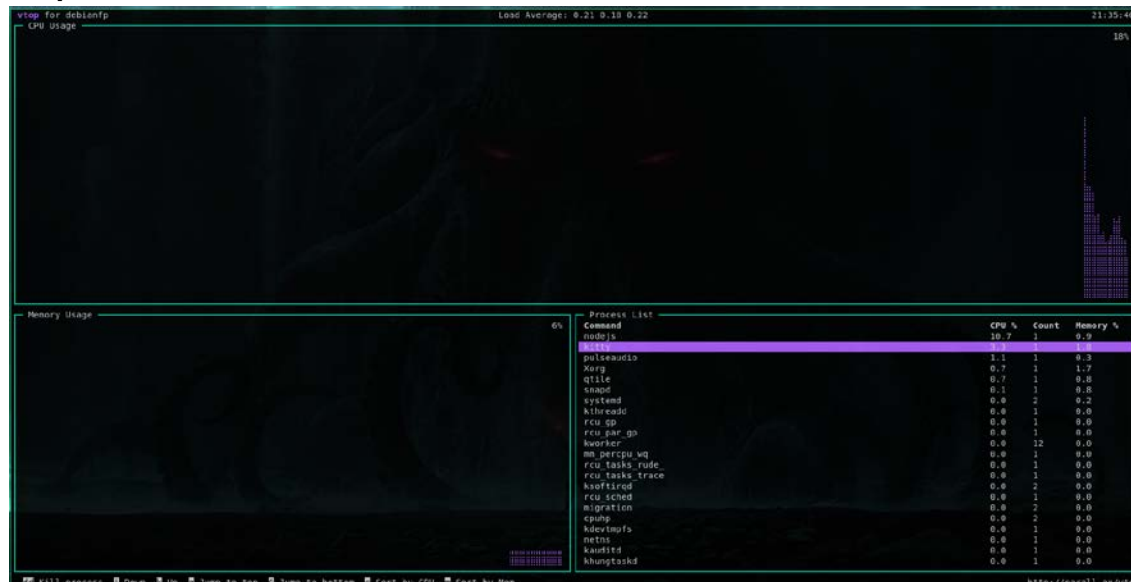
Creo que, dado que estamos analizando herramientas de monitorización de procesos y del sistema, es importante señalar el comando *top*. Este comando nos muestra en la pantalla del terminal toda la información de los procesos en ejecución, se actualiza en tiempo real hasta que detenemos el proceso y nos monitoriza el consume de recursos del sistema.

Personalmente, prefiero usar un fork que tenga mayor legibilidad, recomiendo en este caso *htop* o *vtop*.

## Htop



## Vtop



# Análisis de programa Python

## Funcionalidad del programa

El objetivo del programa a analizar es crear un proceso hijo que, en el momento de la creación, es una copia exacta del padre, que en este caso es nuestro programa, pero a partir de ese momento, sus estados son independientes. Para comprobarlo, conectamos ambos procesos mediante un pipe de lectura y escritura en el que el proceso padre leerá los datos que escribamos en el proceso hijo. Para ello, usamos descriptores de archivo para acceder a los procesos en el modo que deseemos.

## Aspectos clave

### os.pipe

**Función:** Genera un pipe entre dos procesos y los enlaza entre sí, accediendo a uno en modo lectura y al otro en modo escritura.

Para entenderlo mejor, hemos de comprender que los procesos corren en espacios de memoria separados y lo que haga uno, no influirá al otro, por eso necesitamos crear un *pipe*

para mantener los dos procesos conectados y poder *leer* en un proceso los que *escribimos* en el otro, entendiendo que todo ello se está generando de una forma relativamente simultánea.

## os.fork

**Función:** La función sirve para crear un proceso hijo y no recibe argumentos de entrada. Realiza una llamada al sistema para crear un proceso *hijo* que es una copia del *padre*, el proceso que llama a la función. Ambos tienen PID diferentes y corren en espacios de memoria separados. En el momento de crearse tiene las mismas variables y con los mismos valores asignados que el padre, pero a partir de ese instante, los dos van a modificar las variables de forma independiente, es decir, sus estados quedan separados.

**Retorno:** Si se ha ejecutado la función con éxito, el retorno al padre es el PID del proceso hijo y el retorno al hijo es 0, de este modo, cada proceso es capaz de identificarse a sí mismo. En cambio, si sucede algún error, el proceso padre retorna -1 y el proceso hijo no es creado.

## Bloque condicional

El bloque condicional sirve para que cada proceso sea capaz de identificarse a sí mismo y ejecutar la lógica apropiada según hayamos decidido, ya que los dos procesos van a ejecutarse paralelos hasta que los detengamos. Esto se debe a que, como hemos comentado, el valor de retorno de *fork* permite identificar si se trata del proceso es el hijo o el padre y, también, a que el proceso hijo se inicia como una copia del padre, así que en su estado inicial incluirá el bloque condicional que hemos diseñado.

## exit(código)

Es importante que incluyamos el comando para indicar al proceso que termine de ejecutarse. Se considera una buena práctica, aunque no siempre sea imprescindible, por ello debe usarse siempre.

## Análisis pormenorizado del Código

A continuación, incluyo el código del programa comentado función por función, para analizar en profundidad su funcionamiento. Además, he incluido algunos cambios en el código original para ver mejor cómo se realiza la ejecución.

```
# Importamos la librería os que nos permite acceder a funcionalidades
dependientes del Sistema Operativo
# manejando archivos y directorios.
import os

# Importamos la librería sys que nos permite acceder a algunas variables y
funciones que interactúan
# con el interprete de Python.
import sys
```



```
# Inclusión personal: Añado un print que identifica el proceso padre por
pantalla.
print("Padre", os.getpid())

# Creamos un pipe con la función pipe() de la librería os que nos permite manejar
la comunicación
# entre procesos a través de file descriptors. La función pipe() devuelve una
tupla de dos file
# descriptors que se asignan a las variables r y w, r para lectura y w para
escritura.
r, w = os.pipe()

# Creamos un proceso hijo con la función fork() de la librería os que nos permite
crear un proceso
# hijo que es una copia del padre. La función fork() devuelve un valor entero, 0
al
# proceso hijo y el PID del proceso hijo al proceso padre. Asignamos este valor a
la variable
# processid.
processid = os.fork()
# Inclusión personal: Añadimos un printo del valor de processid para comprobar
que el proceso
# padre y el hijo tienen valores distintos. Veremos que más adelante, coincide
con el valor de
# que vamos a obtener llamando a la función getpid() de la librería os dentro del
hijo.
print("Variable processid, coincidirá con el PID del hijo", processid)
# A través del bloque condicional if, podemos comprobar el el proceso es el padre
o el hijo
# Si el proceso es el padre, el valor de processid será distinto de 0, que en
Python se interpreta
# como un true, y se procederá a ejecutar el bloque de instrucciones del padre.
if processid > 0: # Cambio personal: Cambio el operador de comparación de a >
porque incluyo un manejo de errores.
    # Cerramos el file descriptor de escritura del pipe.
    os.close(w)
    # Abrimos el file descriptor de lectura del pipe con la función fdopen() de
la librería os
    # que nos permite abrir un file descriptor como un objeto de tipo file.
Asignamos este objeto
    # a la variable r de nuestro pipe, es decir, el descriptor de lectura que
abrimos en el pipe.
    r = os.fdopen(r)
    # Informamos por pantalla que el proceso es el padre y leemos el contenido
del file descriptor
```

```
# del pipe con la función read() de la librería sys que nos permite leer el
contenido de un
# objeto.
print("Padre leyendo")
# Leemos el contenido del file descriptor del pipe y lo asignamos a la
variable str.
str = r.read()
# Imprimimos por pantalla el contenido de la variable str, que es el texto
que escribió el
# proceso hijo en el pipe.
print("texto:", str)
# Añado a modo personal una línea de código para informar por pantalla que el
proceso padre
# ha finalizado.
print("Padre Cierra")
# Finalizamos el proceso con un código de salida 0, éxito.
sys.exit(0)

# Si el proceso es el hijo, processid devuelve 0, que en Python se interpreta como
un false,
# y se procederá a ejecutar el bloque de instrucciones del hijo.
elif not processid: # Cambio personal: Cambio el operador de comparación a not
porque incluyo un manejo de errores.
    # Inclusión personal: Añado un print que identifica el proceso hijo por
pantalla.
    print("Hijo", os.getpid())
    # Cerramos el file descriptor de lectura del pipe.
    os.close(r)
    # Abrimos el descriptor de escritura del pipe con la función fdopen() de la
librería os
    # que permite que abramos un descriptor como un objeto de tipo file y
asignamos el objeto
    # a la variable w de nuestro pipe, es decir, el descriptor de escritura que
abrimos en el pipe.
    # Además, indicamos que el acceso al archivo se hará en modo escritura, 'w',
usando la
    # la función write().
    w = os.fdopen(w, 'w')
    # Informamos por pantalla que el proceso es el hijo y que estamos escribiendo
en el pipe hijo.
    print("Hijo escribiendo")
    # Usamos la función de sys write() para escribir en el file descriptor w del
pipe.
    # Escribimos el texto "Hello World", el cual será leído por el proceso padre.
    w.write("Hello World")
```

```
# Cerramos el file descriptor de escritura del pipe.
w.close()
# Informamos por pantalla que el proceso hijo ha finalizado.
print("Hijo Cierra")
# Finalizamos el proceso con un código de salida 0, éxito.
sys.exit(0)
else: # Cambio personal: Incluyo un manejo de errores.
# Informamos por pantalla que el proceso no es ni el padre ni el hijo.
print("Error en el proceso")
# Finalizamos el proceso con un código de salida 1, error.
sys.exit(1)
```

## Ejecución del código con las nuevas implementaciones

```
ACT1_EJ1_v2.py
dboj@debianfp:~/workspace/shared/workspace/UOC/python$ python3 ACT1_EJ1_v2.py
Padre 2125
Variable processid, coincidirá con el PID del hijo 2126
Padre leyendo
Variable processid, coincidirá con el PID del hijo 0
Hijo 2126
Hijo escribiendo
Hijo Cierra
texto: Hello World
Padre Cierra
dboj@debianfp:~/workspace/shared/workspace/UOC/python$
```

# Anexo 1. Introducción a las librerías

## Módulo os

El módulo Os de Python ofrece una serie de herramientas para trabajar sobre nuestro sistema. En resumen, nos ofrece un marco para interactuar con nuestro sistema.

### Importación

```
$ import os
```

### Métodos más comunes para el manejo de directorios

Os permite navegar por el árbol de directorios, crear y eliminar directorios o acceder a su contenido.

En el manejo de directorios podemos usar `.` y `..` y `//`. Las referencias de los directorios se pasan como strings.

`os.getcwd` Retorna el directorio actual  
`os.mkdir` Crea un nuevo directorio, `makedirs` crea directorios de forma recursiva.  
`os.chdir` Permite cambiar de directorio.  
`os.rmdir` Elimina un directorio, `rmdirs` elimina directorios de forma recursiva.  
`os.listdir` Lista todos los archivos y subdirectorios del directorio especificado.  
`os.uname` Devuelve el nombre del sistema operativo e información sobre este.  
`os.chroot` Cambia al directorio raíz.  
`os.chmod` Cambia los permisos de un archivo o de un directorio.  
`os.chown` Cambia el propietario de un archivo o de un directorio.  
`os.rename` Renombra un archivo o directorio.  
`os.symlink` Crea un enlace simbólico.

## Acceso a variables de entorno

El módulo `Os` también nos permite acceder a las variables de entorno del sistema y tiene un submódulo que implementa un diccionario con métodos:

- `os.getenv(key, default=None)` → Devuelve el valor de la variable de entorno.
- `os.environ` → Diccionario con las variables de entorno del sistema.

## Acceso a información de sobre los procesos del sistema

De igual modo, tenemos herramientas para manejar la información de los UID de nuestros procesos.

Mediante la librería `Os` podemos conseguir los ID o cambiarlos. Algunas funciones comunes:

`os.getlogin` Devuelve el nombre del usuario logeado en la terminal.  
`os.getpid` Devuelve el ID del proceso actual.  
`os.getppid` Devuelve el ppid del proceso actual  
`os.getuid` Devuelve el ID de usuario del proceso.  
`os.fork` Crea un nuevo proceso hijo heredado de un proceso padre. Devuelve 0 en el proceso hijo y el PID del hijo en el proceso padre. Solo puede usarse en plataformas UNIX.

## Nombres de rutas y archivos

Os implementa el submódulo `os.path` que ofrece acceso a métodos sobre las rutas de archivos y directorios.

<code>os.path.abspath</code>	Ruta absoluta
<code>os.path.basename</code>	Directorio base
<code>os.path.exists</code>	Saber si un directorio existe
<code>os.path.getatime</code>	Conocer último acceso a un directorio
<code>os.path.getsize</code>	Conocer tamaño del directorio
<code>os.path.isabs</code>	Saber si una ruta es absoluta
<code>os.path.isfile</code>	Saber si una ruta es un archivo
<code>os.path.isdir</code>	Saber si una ruta es un directorio
<code>os.path.islink</code>	Saber si una ruta es un enlace simbólico
<code>os.path.ismount</code>	Saber si una ruta es un punto de montaje

## Acceso y manejo de archivos

La librería `Os` nos permite acceder, leer, crear y manejar archivos.

<code>os.open</code>	Abre un archivo y setea las flags según el modo de acceso a este. Nos devuelve el descriptor del archivo.
<code>os.close</code>	Cierra un descriptor de archivo.
<code>os.dup</code>	Devuelve un duplicado del descriptor.
<code>os.fdopen</code>	Devuelve un objeto con el archivo abierto relacionado con un descriptor.
<code>os.fstat</code>	Informa del estado de un descriptor.
<code>os.fsync</code>	Fuerza la escritura en disco del archivo relacionado al descriptor.
<code>os.open</code>	Abre un archivo.

## Pipes

En relación al manejo de descriptores, podemos trabajar mediante un *pipe* que nos permite a la vez el acceso en modo lectura a un descriptor y en modo escritura a otro. Es decir, relacionamos ambos descriptores, leyendo en uno y escribiendo en otro.

<code>os.pipe</code>	Genera el pipe.
<code>os.popen</code>	Genera un pipe desde o hasta la línea de comandos.

## Módulo sys

El módulo Sys permite interactuar con el intérprete de Python en tiempo de ejecución, pudiéndolo configurar durante la ejecución del programa, y, a la vez, una serie de herramientas para interactuar con el entorno fuera del programa.

### Importación

```
$ import sys
```

### Funcionalidades principales

- `sys.exit` Permite finalizar la ejecución del programa y devolver un valor que indica un código para una finalización esperada, 0, o inesperada, otro valor.
- `sys.argv` Retorna la lista de argumentos con la que se inicia el programa. Con la que se realiza la llamada al intérprete.
- `sys.path` Lista de directorios donde Python busca los módulos cuando se realiza un import en el programa.
- `sys.version` Devuelve una cadena que muestra la versión del intérprete de Python.

## Recursos

3.11.2 Documentation. (s. f.). <https://docs.python.org/3/>

GeeksforGeeks. (2021, 11 octubre). *Python os.fork method*.

<https://www.geeksforgeeks.org/python-os-fork-method/>

*Python OS File/Directory Methods*. (s. f.).

[https://www.tutorialspoint.com/python/os\\_file\\_methods.htm?key=python+os](https://www.tutorialspoint.com/python/os_file_methods.htm?key=python+os)

*Python OS File/Directory Methods*. (s. f.).

[https://www.tutorialspoint.com/python/os\\_file\\_methods.htm?key=python+os](https://www.tutorialspoint.com/python/os_file_methods.htm?key=python+os)

*What is Python's Sys Module*. (s. f.). <https://www.tutorialspoint.com/what-is-python-s-sys-module>