Cambridge Small (2025), , 1–13

CAMBRIDGE
UNIVERSITY PRESS

ARTICULO CIENTIFICO

Procesos en Linux

Merino Vidal Mateo Alejandro

Universidad Mayor de San Simón
Cochabamba, Bolivia
202301308@est.umss.edu

Abstract

El presente informe tiene como objetivo ejecutar dos códigos en lenguaje C, relacionados con la creación e identificación de procesos. Para ello, se utilizó una distribución de Linux denominada "adios". Sin embargo, dado que el sistema operativo principal es Windows, fue necesario recurrir a una máquina virtual, específicamente VirtualBox, que permitió ejecutar dicha distribución dentro de un entorno virtualizado.

Keywords: proceso, jerarquía, sistema operativo

1. Introducción

Cada acción iniciada, desde abrir el shell de explorer hasta ejecutar un simple comando conlleva diversos procesos. Estos procesos están basados en una jerarquía, similar al de un árbol genealógico, por lo que es necesario emplear términos como padres e hijos para observar el nivel y relación de cada uno.

Cada proceso cuenta con su propia información, incluyendo su identificador de proceso (PID) y el identificador de su proceso padre (PPID). Un proceso padre es aquel que crea uno o varios procesos hijos a través de una llamada al sistema.

Los procesos hijos heredan casi todo del padre como: código, variables, archivos abiertos, etc. Cada uno cuenta con su propio identificador único de proceso, distinto al del padre.
Además, tienen la capacidad de ejecutarse en paralelo con el padre o esperar instrucciones, dependiendo de la situación.

Es posible realizar esa creación de procesos mediante la llamada al sistema fork().

Fork: Es una llamada al sistema en Unix/Linux, que permite a un proceso crear una copia de sí mismo, generando un nuevo proceso hijo. Este proceso hijo hereda la mayoría de las características del proceso padre, pero tendrá un identificador de proceso único. Tanto el proceso padre como el hijo continuarán ejecutándose desde el mismo punto después del fork().

Esta llamada al sistema también posee un valor de retorno, el cual brinda información importante, es decir, si el valor de retorno es:

- Numero Positivo: Estamos en el proceso padre y devuelve el PID del proceso hijo creado.
- Cero: Estamos en el proceso hijo.
- Menos Uno: Hubo un error y no se creo el proceso hijo.

2. Desarrollo

2.1 Procedimiento

Para el presente trabajo, se tuvieron que seguir una serie de instrucciones, las cuales permitieron utilizar la distribucion "adios" de linux y ejecutar los códigos necesarios.

Entre esas instrucciones están:

1. Copiar y pegar los dos códigos correspondientes de la practica de TSO a archivos de texto (txt); y almacenarlos en un USB.

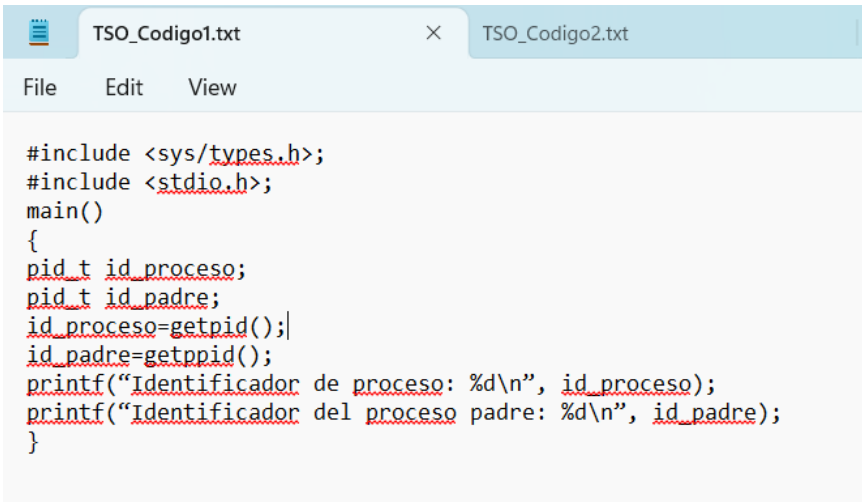


Figure 1. Codigo1

Se observa en la fig.1 una implementación en C, la cual permite la obtención del identificador del proceso mediante la llamada al sistema (system call).
Se obtiene como resultado el identificador del proceso padre e hijo.

TSO_Codigo1.txt TSO_Codigo2.txt

File Edit View

```
#include <sys/types.h>;
#include <stdio.h>;
main()
{
pid_t pid;
pid=fork();
switch(pid){
case -1: /* error en el fork() */
perror("fork");
break;
case 0: /* proceso hijo */

printf("Proceso %d; padre = %d \n", getpid(), getppid() );
break;
default: /* proceso padre */
printf("Proceso %d; padre = %d \n", getpid(), getppid() );
}
}
```

Figure 2. Codigo2

Se observa en la fig.2 una implementación en C, la cual permite la creación de un proceso hijo a través del system call "fork()", el cual sera una copia del proceso en el que estamos actualmente, que seria el padre.

Se obtiene como resultado:

-El PID del proceso padre y el PID de su padre (el shell).
-El PID del proceso hijo y el PID del proceso padre (que es el proceso original antes de ejecutar fork()).

2. Iniciar la herramienta "Virtual Box", la cual permitirá ejecutar la distribucion "adios" de linux en una maquina virtual.

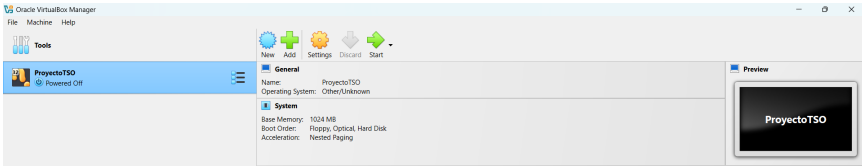


Figure 3. Virtual Box

3. Ejecutar el emulador con la distribucion de linux, conocida como "adios".

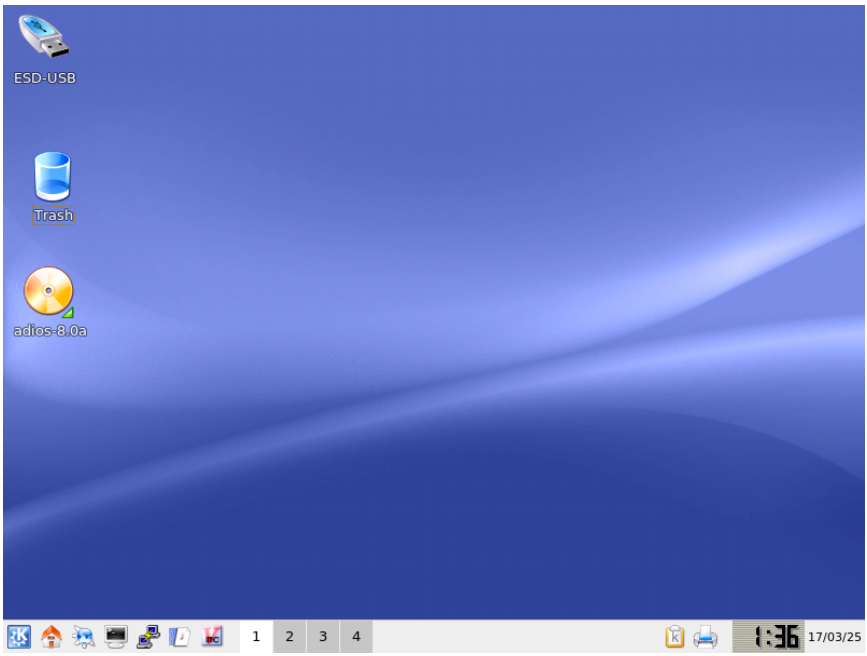


Figure 4. Emulador

Se observa en la fig.4 el entorno de trabajo proporcionado por la maquina virtual, así como un acceso directo al directorio ESD-USB, que confirma el reconocimiento del USB y la presencia de los archivos TXT con los códigos a ejecutar.

4. Se inicializa el CMD o la interfaz de comandos del sistema.

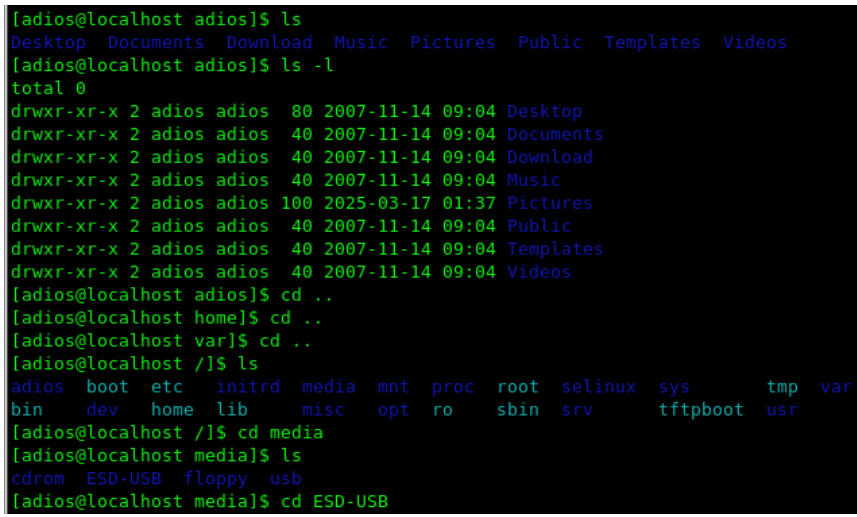


Figure 5. Ingreso al directorio del USB

Se observa en la fig.5 que mediante los comandos "ls" y "ls -l", se obtiene un listado de los directorios y archivos contenidos en el directorio actual. Posteriormente, se retrocede varios directorios mediante el comando "cd ..", hasta llegar al directorio raíz que contiene el directorio "media". Se accede al directorio "media" que contiene el directorio "ESD-USB" correspondiente al USB y se accede a ese directorio.

5. Una vez ubicados en el directorio ESD-USB, se pide un listado de los archivos, para confirmar la existencia de cada código con su correspondiente archivo TXT.

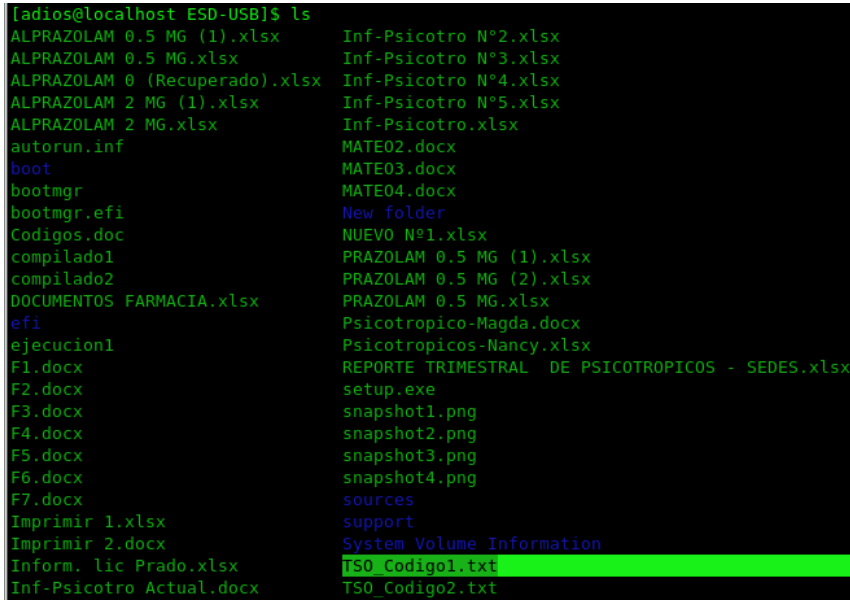


Figure 6. Lista de archivos en el directorio ESD-USB

Se observa en la fig.6 la existencia de los dos archivos TXT, que están con el nombre de "TSO_Codigo1.txt" y "TSO_Codigo2.txt".

6. Se utiliza el comando "mv" para cambiar el nombre de los archivos y sus extensiones de txt a c. Este cambio se realiza para ambos archivos txt con la finalidad de poder compilarlos y ejecutarlos después.

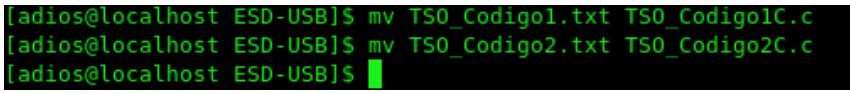


Figure 7. Renombrado de los archivos TXT

Se observa en la fig.7 que el archivo "TSO_Codigo1.txt" cambio a "TSO_Codigo1C.c" y el archivo "TSO_Codigo2.txt" cambio a "TSO_Codigo2C.c".

7. Se utiliza el comando "cat" para observar el contenido de ambos archivos, en los cuales se puede detectar errores de sintaxis como el uso de ';' en los #include de ambos archivos y la diferencia de comillas dobles a las aceptadas.

```
[adios@localhost ESD-USB]$ cat TSO_Codigo1C.c
#include <sys/types.h>;
#include <stdio.h>;
main()
{
pid_t id_proceso;
pid_t id_padre;
id_proceso=getpid();
id_padre=getppid();
printf("Identificador de proceso: %d\n", id_proceso);
printf("Identificador del proceso padre: %d\n", id_padre);
}[adios@localhost ESD-USB]$
```

Figure 8. Contenido del archivo TSO_Codigo1C.c

```
}[adios@localhost ESD-USB]$ cat TSO_Codigo2C.c
#include <sys/types.h>;
#include <stdio.h>;
main()
{
pid_t pid;
pid=fork();
switch(pid){
case -1: /* error en el fork() */
perror("fork");
break;
case 0: /* proceso hijo */

printf("Proceso %d; padre = %d \n", getpid(), getppid() );
break;
default: /* proceso padre */
printf("Proceso %d; padre = %d \n", getpid(), getppid() );
}
}[adios@localhost ESD-USB]$
```

Figure 9. Contenido del archivo TSO_Codigo2C.c

8. Se utiliza el comando "nano" para editar el código en ambos archivos y eliminar la presencia de ';' en cada #include, como también reemplazar las comillas dobles no aceptadas por las correctas.

```
[adios@localhost ESD-USB]$ nano TSO_Codigo1C.c
[adios@localhost ESD-USB]$ nano TSO_Codigo2C.c
```

Figure 10. Uso del comando "nano"

```
 GNU nano 2.0.6                File: TSO_Codigo1C.c                      Modified

#include <sys/types.h>
#include <stdio.h>
main()
{
pid_t id_proceso;
pid_t id_padre;
id_proceso=getpid();
id_padre=getppid();
printf("Identificador de proceso: %d\n", id_proceso);
printf("Identificador del proceso padre: %d\n", id_padre);
}
```

Figure 11. Corrección del archivo TSO_Codigo1C.c

```
 GNU nano 2.0.6                File: TSO_Codigo2C.c                      Modified

#include <sys/types.h>
#include <stdio.h>
main()
{
pid_t pid;
pid=fork();
switch(pid){
case -1: /* error en el fork() */
perror("fork");
break;
case 0: /* proceso hijo */

printf("Proceso %d; padre = %d \n", getpid(), getppid() );
break;
default: /* proceso padre */
printf("Proceso %d; padre = %d \n", getpid(), getppid() );
}
}
```

Figure 12. Corrección del archivo TSO_Codigo2C.c

9. Se utiliza el comando "gcc –o" para compilar ambos archivos con la extensión c. Al hacer esto, se crea un nuevo archivo que es un ejecutable, por cada uno. Del archivo "TSO_Codigo1C.c" se crea el ejecutable "compilado1" y del archivo "TSO_Codigo2C.c" se crea el ejecutable "compilado2".

```
[adios@localhost ESD-USB]$ gcc -o compilado1 TSO_Codigo1C.c
[adios@localhost ESD-USB]$ gcc -o compilado2 TSO_Codigo2C.c
```

Figure 13. Uso del comando "gcc -o"

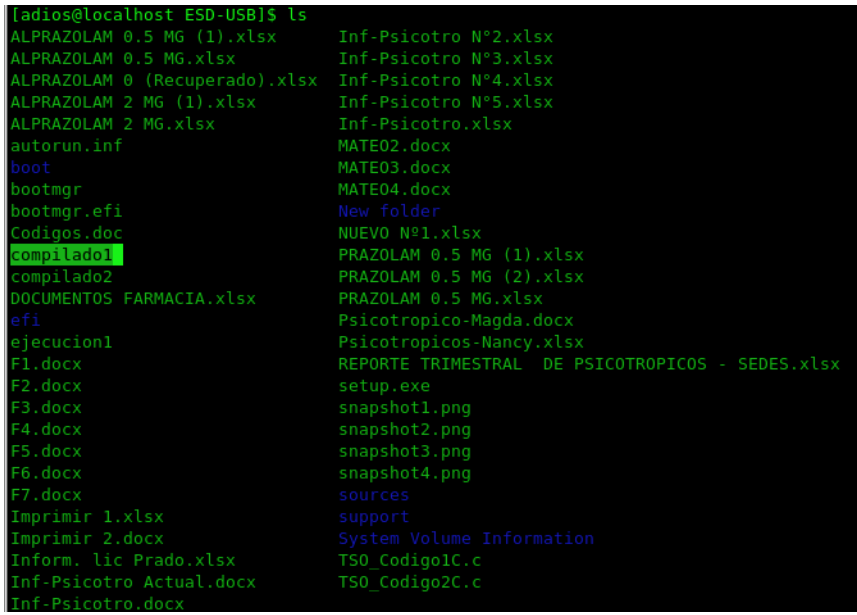


Figure 14. Verificación de archivos compilados

Se observa en la fig.14 , que mediante el comando "ls" es posibles confirmar la existencia de los nuevos archivos compilados, nombrados como "compilado1" y "compilado2".

10. Se utiliza el comando "./" para ejecutar los archivos compilados "compilado1" y "compilado2".

    En el caso del archivo "compilado1", este nos da como resultado:

```
[adios@localhost ESD-USB] $ ./compilado1
Identificador de proceso:  2487
Identificador del proceso padre:  2376
```

    En el caso del archivo "compilado2", este nos da como resultado:

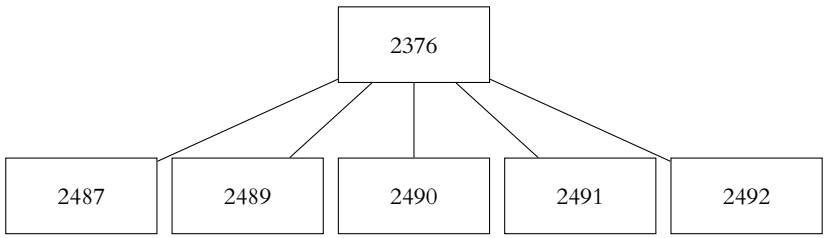
```
[adios@localhost ESD-USB] $ ./compilado2
Proceso 2502; padre = 2501
Proceso 2501; padre = 2376
```

    Posteriormente, al ejecutar los archivos varias veces, nos permite tener mayor información sobre los identificadores de los procesos.

```
[adios@localhost ESD-USB]$ ./compilado1
Identificador de proceso: 2487
Identificador del proceso padre: 2376
[adios@localhost ESD-USB]$ ./compilado1
Identificador de proceso: 2489
Identificador del proceso padre: 2376
[adios@localhost ESD-USB]$ ./compilado1
Identificador de proceso: 2490
Identificador del proceso padre: 2376
[adios@localhost ESD-USB]$ ./compilado1
Identificador de proceso: 2491
Identificador del proceso padre: 2376
[adios@localhost ESD-USB]$ ./compilado1
Identificador de proceso: 2492
```

Figure 15. Ejecución del archivo "compilado1"

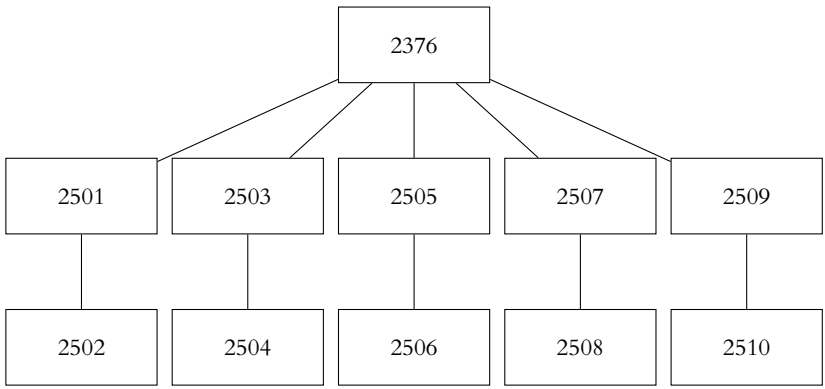
Se observa en la fig.15 que el proceso padre, que es el shell en el que estamos tiene como identificador 2376 y posee diversos hijos con distintos identificadores cada uno. Esto da como resultado una estructura similar a:



```
[adios@localhost ESD-USB]$ ./compilado2
Proceso 2502; padre = 2501
Proceso 2501; padre = 2376
[adios@localhost ESD-USB]$ ./compilado2
Proceso 2504; padre = 2503
Proceso 2503; padre = 2376
[adios@localhost ESD-USB]$ ./compilado2
Proceso 2506; padre = 2505
Proceso 2505; padre = 2376
[adios@localhost ESD-USB]$ ./compilado2
Proceso 2508; padre = 2507
Proceso 2507; padre = 2376
[adios@localhost ESD-USB]$ ./compilado2
Proceso 2510; padre = 2509
Proceso 2509; padre = 2376
```

Figure 16. Ejecución del archivo "compilado2"

Se observa en la fig.16 que el proceso padre, que es el shell en el que estamos, tiene como identificador 2376 y posee diversos hijos con distintos identificadores. Como consecuencia de ejecutar el programa múltiples veces, se muestra que el proceso 2376 es el padre de los procesos 2501, 2503, 2505, 2507, y 2509. Estos, a su vez, crean sus propios hijos, como los procesos 2502, 2504, 2506, 2508, y 2510. Esto da como resultado una estructura similar a:



2.2 Cuestionario

1) ¿El proceso hijo empieza la ejecución del código en su punto de inicio, o en la sentencia que está después del fork()?

R.-Empieza en la sentencia después del fork(), ya que el proceso hijo se crea en el momento que se llama a fork(). Es a partir de ese punto que el proceso hijo adquiere un identificador único y las mismas características que el proceso padre. Ambos procesos, continúan ejecutando el código a partir de la sentencia que sigue al fork(), pero de forma independiente.

2) Observar, que el hijo no es totalmente idéntico al padre, algunos de los valores del BCP han de ser distintos. Responder: ¿cuáles deberían ser las diferencias más importantes?

Afirmación	Verdadero (V) / Falso (F)
El proceso hijo tiene su propio identificador de proceso, distinto al del padre.	V
El proceso hijo tiene una nueva descripción de la memoria. Aunque el hijo tenga los mismos segmentos con el mismo contenido, no tienen por qué estar en la misma zona de memoria.	V
El tiempo de ejecución del proceso hijo se iguala a cero.	V
Todas las alarmas pendientes se desactivan en el proceso hijo.	V
El conjunto de señales pendientes se pone a vacío.	V
El valor que retorna el sistema operativo como resultado del fork es distinto en el hijo y el padre.	V

3) Observar que las modificaciones que realice el proceso padre sobre sus registros e imagen de memoria después del fork() no afectan al hijo y, viceversa, las del hijo no afectan al padre. Sin embargo, el proceso hijo tiene su propia copia de los descriptores del proceso padre. Este hace que el hijo tenga acceso a los archivos abiertos por el proceso padre. El padre y el hijo comparten el puntero de posición de los archivos abiertos en el padre. Responder si esto podría afectar y de que manera a lo que acaba de observar.

R.-Se observó durante la ejecución del código, que el proceso hijo en realidad tiene una copia independiente de los registros e información del proceso padre. Sin embargo, este también posee los descriptores o referencias de archivos a los que apunta el proceso padre, por lo que, si el padre realiza una operación de lectura o escritura sobre un archivo, este llega a afectar a la posición del puntero de archivo del hijo, ya que ambos lo comparten.

3. Conclusión

- El uso de la llamada al sistema fork() permite crear procesos hijos que heredan características del proceso padre. Sin embargo, cada hijo cuenta con su propio identificador, distinto al del padre.

- La jerarquía de procesos, con un proceso padre que genera procesos hijos, es clave para comprender cómo se organiza y gestiona el flujo de ejecución en un sistema. Esta organización de los procesos es similar al de un árbol genealógico, por lo que es posible representarla gráficamente para una mejor comprensión.

- La ejecución de programas en la distribución Linux "adios", mediante la aplicación "VirtualBox" que se uso para la visualización, resalta la importancia de entender los comandos básicos del sistema operativo y la gestión de procesos.