

PRÁCTICA 4

Programación de procesos en C para linux

| Estudiante | Escuela | Asignatura |
|---|--|---------------------|
| Rodrigo Infanzón Acosta rinfanzona@ulasalle.edu.pe | Carrera Profesional de Ingeniería de Software | Sistemas Operativos |

| Informe | Tema | Duración |
|---------|---|----------|
| 05 | Programación de procesos en C para linux | 04 horas |

| Semestre académico | Fecha de inicio | Fecha de entrega |
|--------------------|-----------------|------------------|
| 2024 - B | 04/10/24 | 11/10/24 |

Índice

| | |
|--|----------|
| 1. Objetivos | 2 |
| 2. Recursos y herramientas utilizados | 2 |
| 3. URL de repositorio Github | 2 |
| 4. Marco teórico | 3 |
| 5. El árbol de procesos | 3 |
| 5.1. El proceso INIT | 3 |
| 5.2. PS y PSTREE | 3 |
| 5.2.1. PS | 3 |
| 5.2.2. PSTREE | 4 |
| 6. Identificadores PID, PPID, UID, EUID, GID y EGID | 5 |
| 6.1. Process ID (PID) | 5 |
| 6.2. Parent Process ID (PPID) | 5 |
| 6.3. UID y EUID | 5 |
| 7. Creación y terminación de procesos | 5 |
| 7.1. Creación de un nuevo proceso | 5 |
| 7.1.1. La llamada al sistema - fork() | 5 |
| 8. Finalización de procesos | 6 |
| 8.1. exit | 6 |
| 8.2. Wait y waitpid | 6 |
| 9. Llamadas exec | 6 |

| | |
|--|-----------|
| 10. Actividades propuestas | 7 |
| 10.1. Actividad 1 | 7 |
| 10.2. Actividad 2 | 8 |
| 11. Ejercicios propuestos | 11 |
| 11.1. Ejercicio 1 | 11 |
| 11.2. Ejercicio 2 | 14 |
| 11.3. Ejercicio 3 | 16 |
| 11.4. Ejercicio 4 | 19 |
| 11.5. Ejercicio 5 | 21 |
| 12. Cuestionario | 24 |
| 12.1. ¿Cuál es la principal característica de crear un proceso utilizando la función FORK? . . | 24 |
| 12.2. ¿Cuántos procesos FORK se pueden crear de forma secuencial? ¿Existe algún límite establecido por el sistema operativo? | 24 |
| 12.3. ¿Qué trabajo realiza la función EXECLP? Explique utilizando un ejemplo de utilización del comando. | 25 |
| 12.4. Implementación | 26 |

1. Objetivos

- El estudiante comprenderá y analizará el funcionamiento interno de los sistemas operativos desde la utilización de comandos hasta la programación de procesos.
- El estudiante deberá de probar, analizar y explicar el comportamiento de los procesos planteados. Además de interpretar la funcionalidad que conlleva la programación de procesos a nivel del sistema operativo.

2. Recursos y herramientas utilizados

- Sistema operativo utilizado: Windows 10 pro 22H2 de 64 bits SO. 19045.4170.
- Hardware: Ryzen 7 5700X 4.0 GHz, RAM 32 GB DDR4 3200 MHz, RTX 4060 Asus Dual.
- Virtual Box 7.0.20-163906-Win
- Git 2.44.0.
- Sistema invitado utilizado: Fedora Linux 40 Cinnamon Spin
- Conocimientos básicos en Git.
- Conocimientos básicos en sistemas operativos.
- Conocimientos básicos en Linux.
- C y C++ (gcc y g++).

3. URL de repositorio Github

URL para la práctica 4 en el repositorio GitHub:

- <https://github.com/RodrigoStranger/sistemas-operativos-24b/tree/main/Practica%204>

4. Marco teórico

Un proceso es una instancia en ejecución de un programa. Si está ejecutando dos terminales actualmente, entonces está ejecutando dos procesos de un mismo programa (el shell).

5. El árbol de procesos

- Su PID: Process ID o identificador de procesos.
- El número de su proceso padre PPID, Parent Process ID o identificador del proceso padre.

En la parte superior de esta jerarquía se encuentra el proceso INIT, usualmente posee: $PID = 1$

5.1. El proceso INIT

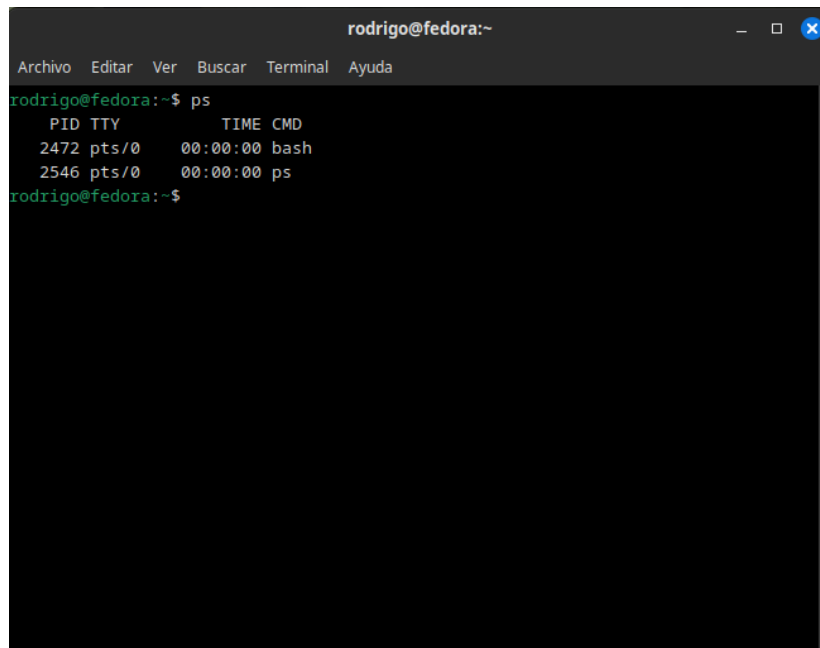
Al momento de iniciar el sistema operativo, se desencadena una secuencia de ejecuciones de procesos. En primer lugar, se carga el núcleo de Linux (Kernel). Luego, dependiendo de los sistemas puede existir un proceso con $PID = 0$, denominado planificador, o swapper. Seguido, se lee un fichero llamado inittab donde se relacionan una serie de procesos que deben arrancarse para permanecer activos (daemons). Se asegura que en caso de error estos puedan volverse a iniciar.

5.2. PS y PSTREE

Estos dos comandos muestran una lista de los procesos en ejecución del sistema:

5.2.1. PS

Encargado de mostrar solo los procesos iniciados por su usuario actual:



```
rodrigo@fedora:~  
Archivo Editar Ver Buscar Terminal Ayuda  
rodrigo@fedora:~$ ps  
  PID TTY          TIME CMD  
 2472 pts/0    00:00:00 bash  
 2546 pts/0    00:00:00 ps  
rodrigo@fedora:~$
```

```
rodrigo@fedora:~$ ps -e -o pid,ppid,command
PID    PPID  COMMAND
1       0    /usr/lib/systemd/systemd --switched-root --system --deserialize=44 rhgb
2       0    [kthreadd]
3       2    [pool_workqueue_release]
4       2    [kworker/R-rcu_gp]
5       2    [kworker/R-sync_wq]
6       2    [kworker/R-slub_flushwq]
7       2    [kworker/R-netns]
8       2    [kworker/0:0-cgroup_destroy]
10      2    [kworker/0:0H-events_highpri]
13      2    [kworker/R-mm_percpu_wq]
14      2    [rcu_tasks_kthread]
15      2    [rcu_tasks_rude_kthread]
16      2    [rcu_tasks_trace_kthread]
17      2    [ksoftirqd/0]
18      2    [rcu_preempt]
19      2    [rcu_exp_par_gp_kthread_worker/0]
20      2    [rcu_exp_gp_kthread_worker]
21      2    [migration/0]
22      2    [idle_inject/0]
23      2    [cpuhp/0]
24      2    [cpuhp/1]
25      2    [idle_inject/1]
26      2    [migration/1]
27      2    [ksoftirqd/1]
29      2    [kworker/1:0H-events_highpri]
30      2    [cpuhp/2]
31      2    [idle_inject/2]
```

5.2.2. PSTREE

Este comando muestra los procesos en forma de un árbol, permitiendo identificar los procesos padre e hijos, o procesos dependientes.

```
rodrigo@fedora:~$ pstree
systemd--ModemManager--3*[{ModemManager}]
--NetworkManager--3*[{NetworkManager}]
--2*[{VBoxClient}--VBoxClient--3*[{VBoxClient}]]
--VBoxClient--VBoxClient--4*[{VBoxClient}]
--VBoxDRMClient--4*[{VBoxDRMClient}]
--VBoxService--8*[{VBoxService}]
--abrt-dbus--3*[{abrt-dbus}]
--3*[{abrt-dump-journ}]
--abrt-d--3*[{abrt-d}]
--accounts-daemon--3*[{accounts-daemon}]
--alsactl
--atd
--auditd--sedispatch
--2*[{auditd}]
--avahi-daemon--avahi-daemon
--chronyd
--colord--3*[{colord}]
--crrond
--csd-printer--3*[{csd-printer}]
--cupsd
--dbus-broker-lau--dbus-broker
--firewalld--{firewalld}
--gnome-keyring-d--4*[{gnome-keyring-d}]
--gssproxy--5*[{gssproxy}]
--httpd--httpd
--2*[{httpd}--68*[{httpd}]]
--2*[{httpd}--52*[{httpd}]]
--ixbalanco--{ixbalanco}
```

6. Identificadores PID, PPID, UID, EUID, GID y EGID

6.1. Process ID (PID)

Al iniciar un nuevo proceso se le asigna un identificador de proceso único (PID). Este identificador debe utilizarse por el administrador para referirse a un proceso dado al ejecutar un comando. Los PID son asignados por el sistema en orden creciente comenzando desde cero. Si antes de un reinicio del sistema se llega al número máximo, se vuelve a comenzar desde cero, saltando los identificadores de los procesos que siguen activos.

6.2. Parent Process ID (PPID)

La creación de nuevos procesos en Linux se realiza mediante la duplicación de un proceso existente invocando el comando `fork()`. Al proceso original se le llama padre y al nuevo proceso hijo. El PPID de un proceso es el PID de su proceso padre.

6.3. UID y EUID

Los procesos tienen un EUID (Effective User ID), y un UID cuando ambos coinciden. El UID es el identificador de usuario real que coincide con el identificador del usuario que invocó el proceso.

El EUID es el identificador de usuario efectivo y se llama así porque es el identificador que se tiene en cuenta a la hora de considerar los permisos.

El UID ayuda a identificar quien es el propietario actual de ese proceso. El UID también es un atributo presente en otros elementos del sistema. Por ejemplo, los ficheros y directorios del sistema tienen este atributo. De esta forma cuando un proceso intenta efectuar una operación sobre un fichero. El kernel comprobará si el EUID del proceso coincide con el UID del fichero.

Por ejemplo, si se establece que determinado fichero solo puede ser leído por su propietario, el kernel denegará todo intento de lectura a un proceso que no tenga un EUID igual al UID del fichero salvo que se trate del usuario Kernel o Root.

7. Creación y terminación de procesos

7.1. Creación de un nuevo proceso

El núcleo del sistema operativo (kernel) es el encargado de realizar la mayoría de las funciones básicas como los procesos. La gestión de los procesos se realiza mediante las llamadas al sistema, que se encuentran implementadas en el lenguaje C.

7.1.1. La llamada al sistema - `fork()`

La forma de iniciar un proceso se realiza mediante otro proceso que realiza una llamada al sistema con la función `fork()`.

`fork()` duplica un proceso, generando dos procesos casi idénticos. La única diferencia se encuentra en los valores PID y PPID.

Un proceso puede pasar al proceso hijo una serie de variables, pero un proceso hijo no puede pasar ningún valor en una variable a su padre. Además, `fork()` retorna un valor numérico que será 1 en caso de fallo, pero si tiene éxito se habrá producido la duplicación de procesos y se retornará un valor distinto para el proceso hijo.

Al proceso hijo se le retornará el valor 0 y al proceso padre se le retornará el PID del proceso hijo.

`int PID = fork()`

- `PID = 0` en el hijo,
- `PID > 0` en el padre (y contiene el PID del hijo)
- `PID < 0` si error

8. Finalización de procesos

8.1. `exit`

`void exit (int status)`

`exit` finaliza al proceso que lo invocó, con un código de estado igual al byte inferior del parámetro `status`.

Todos los descriptores de archivo abiertos son cerrados y sus buffers sincronizados. Si hay procesos hijos cuando el padre ejecuta un `exit ()`, el PPID de los hijos se cambia a 1 (proceso `init`). Es la única llamada al sistema que nunca retorna.

El valor del parámetro `status` se utiliza para comunicar al proceso padre la forma en que el proceso hijo termina. Por convenio, este valor suele ser 0 si el proceso termina correctamente y cualquier otro valor en caso de terminación anormal.

El proceso padre puede obtener este valor a través de la llamada al sistema `wait`

8.2. `Wait` y `waitpid`

Espera a que se finalice los procesos hijo, permitiendo obtener sus estados de salida. Una señal de no bloqueo o de no ignorar, puede reactivar el proceso padre.

`pid_t wait (int *statusp)`

Si hay varios procesos hijos, `wait` espera hasta que uno de ellos termine. No es posible especificar la espera un determinado hijo. `wait` retorna el PID del hijo que termina (o 1) si no se crearon hijos, o si ya no hay más hijos por los que se debe esperar) y almacena el código del estado de finalización del proceso hijo (parámetro `status` en su llamada al sistema `exit`) en la dirección del parámetro `statusp`.

9. Llamadas `exec`

Las llamadas `exec` reemplazan al programa de un proceso con otro. Por este motivo, `exec` no retorna ningún valor, a menos que haya ocurrido un error. Esta llamada se puede invocara partir de una familia de funciones:

- **`execvp` y `execlp`**: aceptan el nombre de un programa, el cual debe encontrarse en la ruta actual de ejecución; las funciones que no tengan `p` en el nombre deben recibir la ruta completa del programa.
- **`execv`, `execvp` y `execve`**: aceptan la lista de argumentos para el nuevo programa en la forma de un arreglo de punteros a cadenas terminado en `NULL`.
- **`execl`, `execlp` y `execle`**: acepta la lista de argumentos usando el mecanismo `varargs` de C.
- **`execve` y `execle`**: aceptan un argumento adicional, un arreglo de variables de entorno. Este también debería ser un arreglo de punteros a cadenas terminado en `NULL`. Cada cadena debe ser de la forma `VARIABLE=valor`.

La lista de argumentos que recibe el programa es como los argumentos que especificamos cuando ejecutamos un comando en el shell. Esta lista está disponible a través de los parámetros `argc` y `argv` de la función `main`.

Nota: cuando ejecutamos el programa desde el shell, el nombre del programa es el elemento `argv[0]`, el primer argumento es `argv[1]` y así sucesivamente. Cuando usemos `exec` también debemos pasar el nombre del programa como primer argumento.

10. Actividades propuestas

10.1. Actividad 1

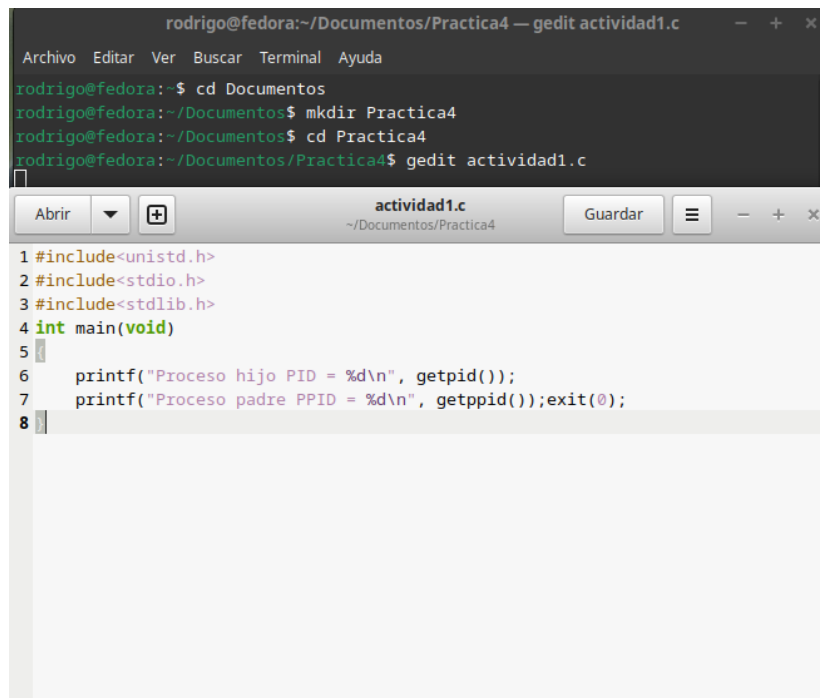
Analice el siguiente código y de una interpretación del resultado obtenido de acuerdo al marco teórico:

```
1 #include<unistd.h>
2 #include<stdio.h>
3 #include<stdlib.h>
4 int main(void)
5 {
6     printf("Proceso hijo PID = %d\n", getpid());
7     printf("Proceso padre PPID = %d\n", getppid());
8     exit(0);
9 }
```

Resolución:

Podemos escribir y ejecutar el código para ver el proceso de salida y analizar.

Escribimos el código:



The screenshot shows a terminal window and a code editor. The terminal window, titled "rodrigo@fedora:~/Documentos/Practica4 — gedit actividad1.c", displays the following commands and output:

```
rodrigo@fedora:~$ cd Documentos
rodrigo@fedora:~/Documentos$ mkdir Practica4
rodrigo@fedora:~/Documentos$ cd Practica4
rodrigo@fedora:~/Documentos/Practica4$ gedit actividad1.c
```

The code editor, titled "actividad1.c", shows the following code:

```
1 #include<unistd.h>
2 #include<stdio.h>
3 #include<stdlib.h>
4 int main(void)
5 {
6     printf("Proceso hijo PID = %d\n", getpid());
7     printf("Proceso padre PPID = %d\n", getppid());
8     exit(0);
9 }
```

Creamos una carpeta llamada `Practica4` dentro del directorio `Documentos`, es ahí donde crearemos nuestros códigos `.c`.

Compilamos y ejecutamos el código:

```
rodrigo@fedora:~/Documentos/Practica4
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~$ cd Documentos
rodrigo@fedora:~/Documentos$ mkdir Practica4
rodrigo@fedora:~/Documentos$ cd Practica4
rodrigo@fedora:~/Documentos/Practica4$ gedit actividad1.c
rodrigo@fedora:~/Documentos/Practica4$ gcc --version
gcc (GCC) 14.2.1 20240912 (Red Hat 14.2.1-3)
Copyright (C) 2024 Free Software Foundation, Inc.
Esto es software libre; vea el código para las condiciones de copia. NO hay
garantía; ni siquiera para MERCANTIBILIDAD o IDONEIDAD PARA UN PROPÓSITO EN
PARTICULAR

rodrigo@fedora:~/Documentos/Practica4$ gcc actividad1.c -o Actividad1
rodrigo@fedora:~/Documentos/Practica4$ ./Actividad1
Proceso hijo PID = 4025
Proceso padre PPID = 3451
rodrigo@fedora:~/Documentos/Practica4$
```

Análisis:

El código presentado ilustra cómo obtener y mostrar el identificador de proceso (PID) del proceso hijo y el identificador de proceso del padre (PPID).

Al inicio, se incluyen bibliotecas esenciales como `unistd.h`, que permite el uso de `getpid()` y `getppid()`, y `stdio.h` para la impresión de resultados en la consola. La función principal del programa ejecuta dos impresiones: la primera muestra el PID del proceso actual, mientras que la segunda muestra el PPID del proceso padre.

Al finalizar, el programa se termina de manera correcta utilizando `exit(0)`. Al ejecutarlo, se espera como salida el PID del proceso hijo junto con el PPID del proceso padre, lo que proporciona una visión clara de la relación jerárquica entre procesos en un sistema operativo Unix/Linux.

10.2. Actividad 2

Analice el siguiente programa. Una vez que se ejecute, ¿cuántos procesos se crean?, ¿qué realiza cada uno de estos procesos?, ¿existe algún tipo de relación entre estos procesos?. Justifique y evidencie sus respuestas.

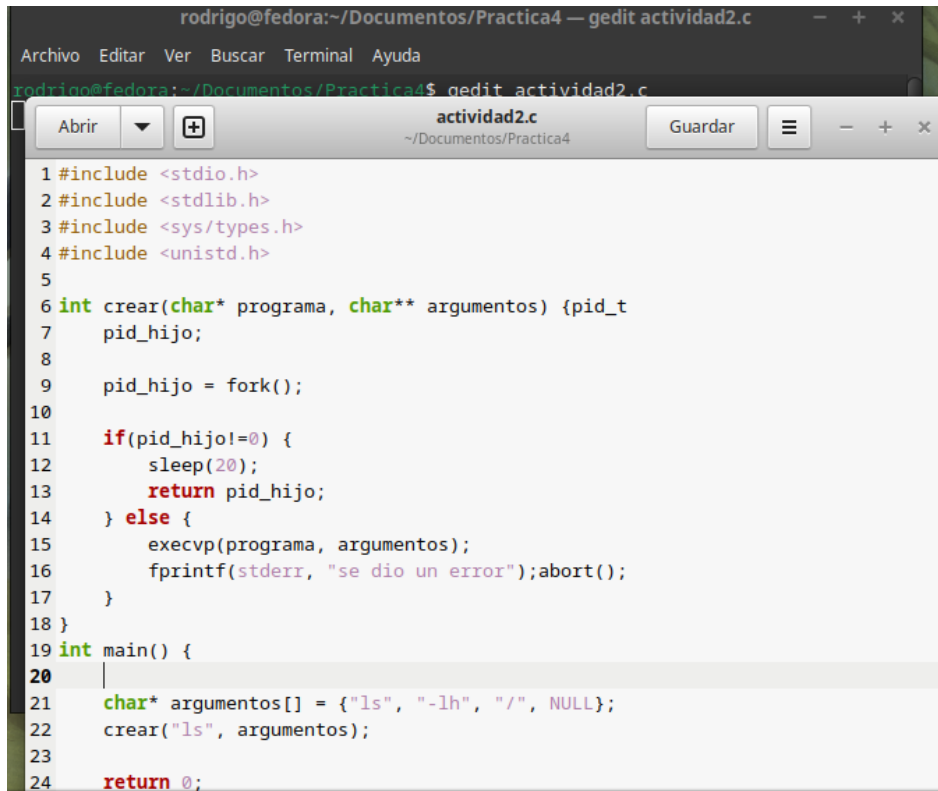
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int crear(char* programa, char** argumentos) {pid_t
7     pid_hijo;
8
9     pid_hijo = fork();
10
11     if(pid_hijo!=0) {
12         sleep(20);
```



```
13     return pid_hijo;
14 } else {
15     execvp(programa, argumentos);
16     fprintf(stderr, "se dio un error"); abort();
17 }
18 }
19 int main() {
20
21     char* argumentos[] = {"ls", "-lh", "/", NULL};
22     crear("ls", argumentos);
23
24     return 0;
25 }
```

Resolución:

Escribimos el código:



```
rodrigo@fedora:~/Documentos/Practica4 — gedit actividad2.c
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit actividad2.c
Actividad2.c
~/Documentos/Practica4
Guardar

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int crear(char* programa, char** argumentos) {pid_t
7     pid_hijo;
8
9     pid_hijo = fork();
10
11     if(pid_hijo!=0) {
12         sleep(20);
13         return pid_hijo;
14     } else {
15         execvp(programa, argumentos);
16         fprintf(stderr, "se dio un error"); abort();
17     }
18 }
19 int main() {
20
21     char* argumentos[] = {"ls", "-lh", "/", NULL};
22     crear("ls", argumentos);
23
24     return 0;
```

Compilamos y ejecutamos el código:

```
rodrigo@fedora:~/Documentos/Practica4
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit actividad2.c
rodrigo@fedora:~/Documentos/Practica4$ gcc actividad2.c -o Actividad2
rodrigo@fedora:~/Documentos/Practica4$ ./Actividad2
total 20K
dr-xr-xr-x.  1 root root    0 ene 23  2024 afs
lrwxrwxrwx.  1 root root    7 ene 23  2024 bin -> usr/bin
dr-xr-xr-x.  6 root root 4,0K oct  5 23:06 boot
drwxr-xr-x. 19 root root 3,8K oct  6 11:57 dev
drwxr-xr-x.  1 root root 4,6K oct  6 12:56 etc
drwxr-xr-x.  1 root root   14 sep 18 22:08 home
lrwxrwxrwx.  1 root root    7 ene 23  2024 lib -> usr/lib
lrwxrwxrwx.  1 root root    9 ene 23  2024 lib64 -> usr/lib64
drwx-----.  1 root root    0 abr 14 18:55 lost+found
drwxr-xr-x.  1 root root    0 ene 23  2024 media
drwxr-xr-x.  1 root root    0 ene 23  2024 mnt
drwxr-xr-x.  1 root root    0 ene 23  2024 opt
dr-xr-xr-x. 321 root root    0 oct  6 11:57 proc
dr-xr-x---.  1 root root 238 sep 19 00:21 root
drwxr-xr-x. 52 root root 1,4K oct  6 11:57 run
lrwxrwxrwx.  1 root root    8 ene 23  2024/sbin -> usr/sbin
drwxr-xr-x.  1 root root    0 ene 23  2024 srv
dr-xr-xr-x. 13 root root    0 oct  6 15:57 sys
drwxrwxrwt. 22 root root 520 oct  6 17:56 tmp
drwxr-xr-x.  1 root root 168 abr 14 18:58 usr
drwxr-xr-x.  1 root root 200 sep 18 23:58 var
rodrigo@fedora:~/Documentos/Practica4$
```

Análisis:

El código comienza incluyendo varias bibliotecas esenciales para la ejecución del programa, como `stdio.h`, `stdlib.h`, `sys/types.h`, y `unistd.h`. La función `crear()` está diseñada para crear un proceso hijo utilizando la función `fork()`. En `crear()`, se toma un programa (en este caso, "ls") y una lista de argumentos que serán pasados al programa a través de `execvp()`, lo cual reemplaza el código del proceso hijo con la ejecución de este programa.

Dentro de la función `crear()`, se llama a `fork()`, que crea un nuevo proceso. Si `pid_hijo` (el valor de retorno de `fork()`) no es cero, significa que estamos en el proceso padre, donde el código hace una pausa de 20 segundos utilizando `sleep(20)` y luego regresa el `pid_hijo`. Si `pid_hijo` es igual a 0, significa que estamos en el proceso hijo, y se llama a `execvp()` para ejecutar el programa (en este caso, ls) con los argumentos especificados. Si `execvp()` falla, se imprime un mensaje de error y el proceso hijo se termina con `abort()`.

Finalmente, el programa principal invoca a la función `crear()` con los argumentos necesarios para ejecutar el comando `ls -lh /`. El proceso padre espera 20 segundos antes de continuar, mientras que el proceso hijo ejecuta el comando ls para listar los archivos del directorio raíz en formato detallado. Si todo funciona correctamente, el resultado es la ejecución del comando ls desde el proceso hijo. Si hay un error en la ejecución, se imprime un mensaje de error.

¿Cuántos procesos se crean?

Se crean dos procesos: el proceso padre, que invoca `crear()` y espera 20 segundos antes de finalizar, y el proceso hijo, que ejecuta el comando `ls` utilizando `execvp()`.

¿Qué realiza cada uno de estos procesos?

El proceso padre sigue su ejecución normal, pausándose durante 20 segundos antes de finalizar. Mientras tanto, el proceso hijo ejecuta el comando `ls -lh /` para listar el contenido del directorio raíz / en formato detallado. Si el comando se ejecuta con éxito, el proceso hijo termina; si `execvp()` falla, imprime un mensaje de error y aborta.

¿Existe algún tipo de relación entre estos procesos?

Existe una relación jerárquica entre el proceso padre y el hijo, creado mediante `fork()`. Inicialmente comparten el mismo contexto, pero el hijo ejecuta un programa diferente con `execvp()`. El padre espera 20 segundos, permitiendo que el hijo complete el comando `ls` antes de finalizar.

11. Ejercicios propuestos

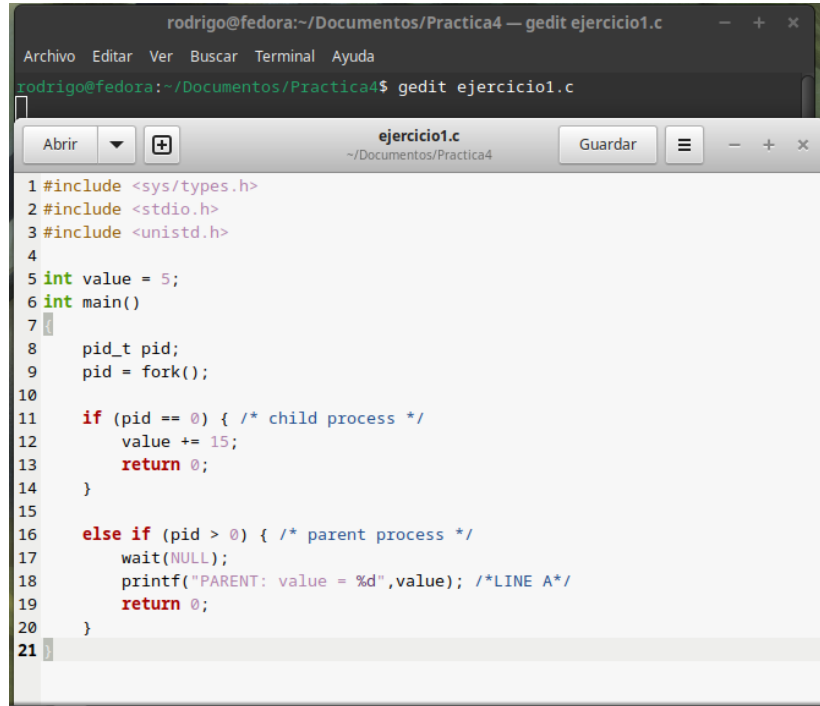
11.1. Ejercicio 1

El siguiente código crea un proceso hijo, realice un seguimiento de la variable `value` y describa el por qué tiene ese comportamiento.

```
1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 int value = 5;
6 int main()
7 {
8     pid_t pid;
9     pid = fork();
10
11     if (pid == 0) { /* child process */
12         value += 15;
13         return 0;
14     }
15
16     else if (pid > 0) { /* parent process */
17         wait(NULL);
18         printf("PARENT: value = %d",value); /*LINE A*/
19         return 0;
20     }
21 }
```

Resolución:

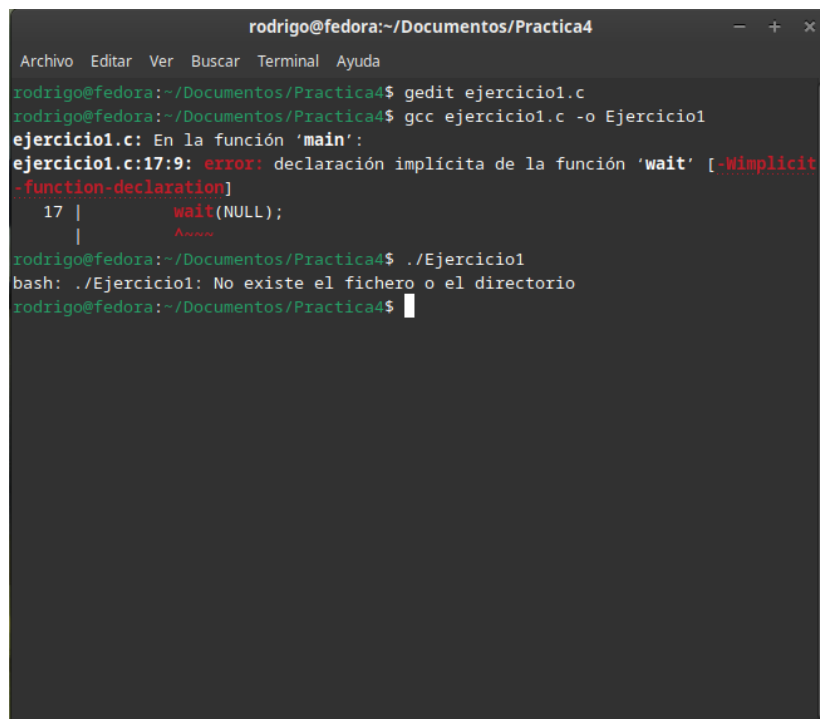
Escribimos el código:



```
rodrigo@fedora:~/Documentos/Practica4 — gedit ejercicio1.c
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio1.c
ejercicio1.c
~/Documentos/Practica4
Guardar

1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 int value = 5;
6 int main()
7 {
8     pid_t pid;
9     pid = fork();
10
11     if (pid == 0) { /* child process */
12         value += 15;
13         return 0;
14     }
15
16     else if (pid > 0) { /* parent process */
17         wait(NULL);
18         printf("PARENT: value = %d",value); /*LINE A*/
19         return 0;
20     }
21 }
```

Compilamos y ejecutamos el código:



```
rodrigo@fedora:~/Documentos/Practica4
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio1.c
rodrigo@fedora:~/Documentos/Practica4$ gcc ejercicio1.c -o Ejercicio1
ejercicio1.c: En la función 'main':
ejercicio1.c:17:9: error: declaración implícita de la función 'wait' [-Wimplicit
-function-declaration]
   17 |         wait(NULL);
      |         ^~~~~
rodrigo@fedora:~/Documentos/Practica4$ ./Ejercicio1
bash: ./Ejercicio1: No existe el fichero o el directorio
rodrigo@fedora:~/Documentos/Practica4$
```

The screenshot shows two windows from a Linux environment.

The top window is a terminal with the prompt `rodrigo@fedora:~/Documentos/Practica4`. It displays the following commands and output:

```
rodrigo@fedora:~/Documentos/Practica4$ gcc ejercicio1.c -o Ejercicio1  
ejercicio1.c: En la función 'main':  
ejercicio1.c:17:9: error: declaración implícita de la función 'wait' [-Wimplicit-function-declaration]  
17 |         wait(NULL);  
    |         ^~~~~
```

The bottom window is a code editor titled "ejercicio1.c" at the path `~/Documents/Practica4`, containing the following C code:

```
#include <sys/types.h>  
#include <stdio.h>  
#include <unistd.h>  
#include sys/wait.h  
int value = 5;  
int main()  
{  
    pid_t pid;  
    pid = fork();  
  
    if (pid == 0) { /* child process */  
        value += 15;  
        return 0;  
    }  
  
    else if (pid > 0) { /* parent process */  
        wait(NULL);  
        printf("PARENT: value = %d", value); /*LINE A*/  
    }
```

A blue arrow points to the fourth line of the code, `#include sys/wait.h`.

```

rodrigo@fedora:~/Documentos/Practica4
Archivo Editar Ver Buscar Terminal Ayuda

rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio1.c
rodrigo@fedora:~/Documentos/Practica4$ gcc ejercicio1.c -o Ejercicio1
ejercicio1.c: En la función 'main':
ejercicio1.c:17:9: error: declaración implícita de la función 'wait' [-Wimplicit
-function-declaration]
    17 |         wait(NULL);
        |         ^~~~~
rodrigo@fedora:~/Documentos/Practica4$ ./Ejercicio1
bash: ./Ejercicio1: No existe el fichero o el directorio
rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio1.c
rodrigo@fedora:~/Documentos/Practica4$ gcc ejercicio1.c -o Ejercicio1
rodrigo@fedora:~/Documentos/Practica4$ ./Ejercicio1
PARENT: value = 5rodrigo@fedora:~/Documentos/Practica4$ █

```

Sistemas Operativos

Seguimiento de la variable value:

La variable value se inicializa en 5 antes de llamar a `fork()`, creando así copias independientes en el proceso padre y en el proceso hijo. Cuando el hijo se ejecuta, incrementa su propia copia de value en 15, resultando en 20. Sin embargo, esta modificación no afecta al padre, que sigue trabajando con su copia original de value, que permanece en 5.

El proceso padre espera a que el hijo termine utilizando `wait()`, y al imprimir el valor de value, muestra 5. Este comportamiento se debe a la separación de memoria establecida por `fork()`, donde cada proceso opera en su propio espacio de memoria, asegurando que las modificaciones en uno no influyan en el otro.

11.2. Ejercicio 2

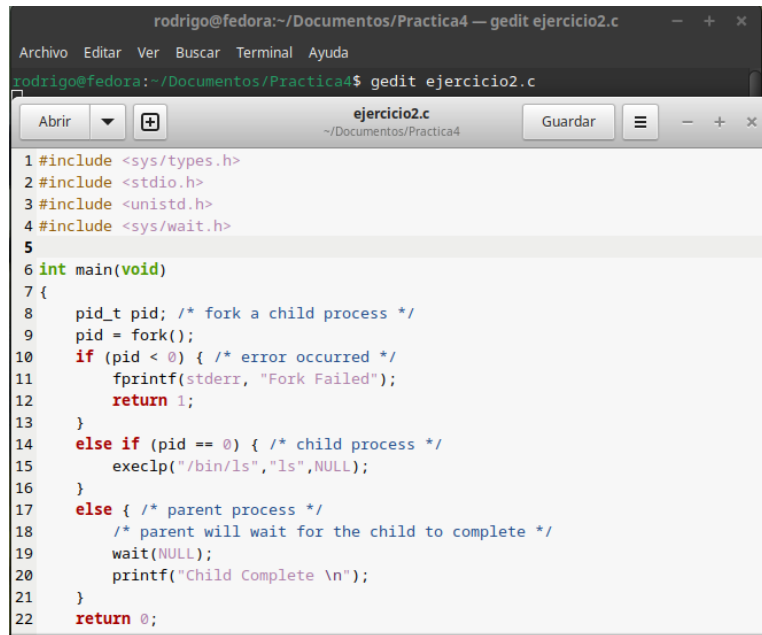
En el siguiente código, detalle que parte del código es ejecutada por el proceso padre y que porción del código es ejecutada por el proceso hijo. Describa la actividad decada uno:

```
1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 int main(void)
6 {
7     pid_t pid; /* fork a child process */
8     pid = fork();
9     if (pid < 0) { /* error occurred */
10         fprintf(stderr, "Fork Failed");
11         return 1;
12     }
13     else if (pid == 0) { /* child process */
14         execlp("/bin/ls", "ls", NULL);
15     }
16     else { /* parent process */
17         /* parent will wait for the child to complete */
18         wait(NULL);
19         printf("Child Complete \n");
20     }
21     return 0;
22 }
```

Resolución:

Como vemos que no posee la librería `<sys/wait.h>` la agregamos para que no exista errores de compilación:

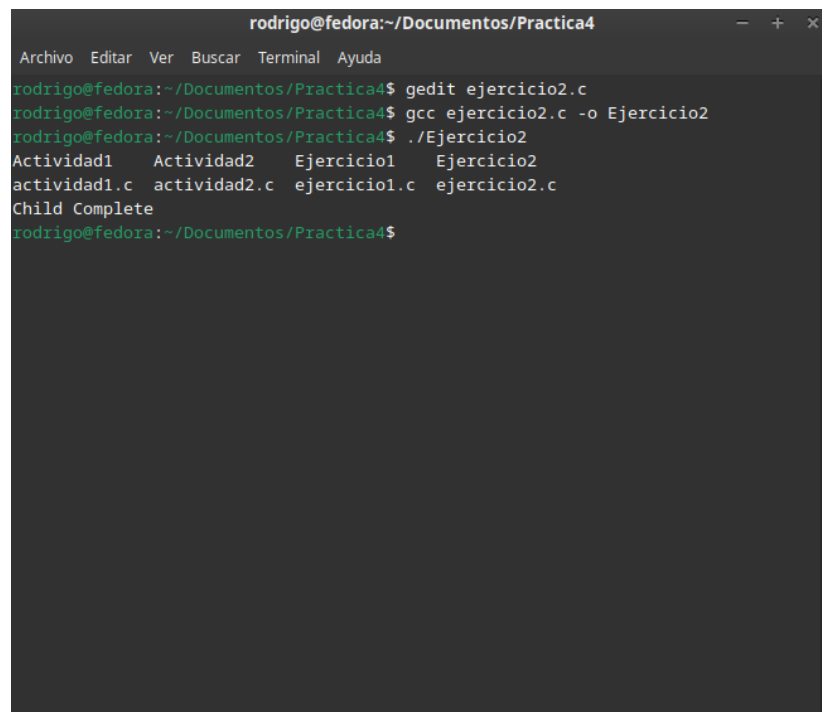
Escribimos el código:



```
rodrigo@fedora:~/Documentos/Practica4 — gedit ejercicio2.c
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio2.c
ejercicio2.c
~/Documentos/Practica4
Guardar

1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int main(void)
7 {
8     pid_t pid; /* fork a child process */
9     pid = fork();
10    if (pid < 0) { /* error occurred */
11        fprintf(stderr, "Fork Failed");
12        return 1;
13    }
14    else if (pid == 0) { /* child process */
15        execlp("/bin/ls", "ls", NULL);
16    }
17    else { /* parent process */
18        /* parent will wait for the child to complete */
19        wait(NULL);
20        printf("Child Complete \n");
21    }
22    return 0;
```

Compilamos y ejecutamos el código:



```
rodrigo@fedora:~/Documentos/Practica4
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio2.c
rodrigo@fedora:~/Documentos/Practica4$ gcc ejercicio2.c -o Ejercicio2
rodrigo@fedora:~/Documentos/Practica4$ ./Ejercicio2
Actividad1 Actividad2 Ejercicio1 Ejercicio2
actividad1.c actividad2.c ejercicio1.c ejercicio2.c
Child Complete
rodrigo@fedora:~/Documentos/Practica4$
```

Ejecución del proceso padre:

Código ejecutado:

```
1 else { /* parent process */
2     wait(NULL);
3     printf("Child Complete \n");
4 }
```

Actividades del proceso padre:

- El proceso padre se identifica al recibir un valor de pid mayor que 0, que es el PID del proceso hijo.
- Llama a `wait(NULL)`; lo que hace que el padre se bloquee hasta que el hijo termine su ejecución.
- Una vez que el hijo ha completado su tarea, el padre imprime `Child Complete`, indicando que el hijo ha terminado.

Ejecución del proceso hijo:

Código ejecutado:

```
1 else if (pid == 0) { /* child process */
2     execlp("/bin/ls", "ls", NULL);
3 }
```

Actividades del proceso hijo:

- El proceso hijo se identifica al recibir un valor de pid igual a 0 tras la llamada a `fork()`.
- Llama a `execlp("/bin/ls", "ls", NULL)`; lo que reemplaza el código del proceso hijo con el programa `ls`, que lista el contenido del directorio actual.
- Si `execlp()` se ejecuta correctamente, el hijo termina después de listar el contenido. Si `execlp()` falla, el hijo no tiene un manejo de errores y terminaría sin imprimir nada.

11.3. Ejercicio 3

Analice el siguiente código, explique cómo se da el flujo del código desde el proceso padre y como procede con los procesos hijos:

Nota: verificando los códigos, completaré las librerías restantes para poder evitar inconvenientes al compilar y ejecutar en linux.

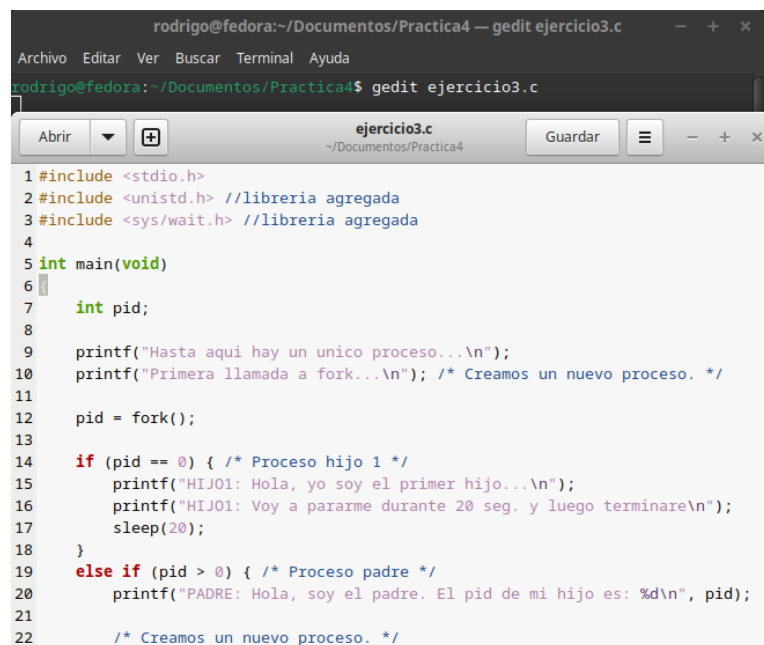
```
1 #include <stdio.h>
2 #include <unistd.h> //libreria agregada
3 #include <sys/wait.h> //libreria agregada
4
5 int main(void)
6 {
7     int pid;
8     printf("Hasta aqui hay un unico proceso...\n");
9     printf("Primera llamada a fork...\n"); /* Creamos un nuevo proceso. */
10
11     pid = fork();
12
13     if (pid == 0) { /* Proceso hijo 1 */
```



```
14     printf("HIJ01: Hola, yo soy el primer hijo...\n");
15     printf("HIJ01: Voy a pararme durante 20 seg. y luego terminare\n");
16     sleep(20);
17 }
18 else if (pid > 0) { /* Proceso padre */
19     printf("PADRE: Hola, soy el padre. El pid de mi hijo es: %d\n", pid);
20     /* Creamos un nuevo proceso. */
21     pid = fork();
22
23     if (pid == 0) { /* Proceso hijo 2 */
24         printf("HIJ02: Hola, soy el segundo hijo...\n");
25         printf("HIJ02: El segundo hijo va a ejecutar la orden 'ls'...\n");
26         execlp("ls", "ls", NULL);
27         printf("HIJ02: Si ve este mensaje, el execlp no funciono...\n");
28     }
29     else if (pid > 0) { /* Proceso padre */
30         printf("PADRE: Hola otra vez. Pid de mi segundo hijo: %d\n", pid);
31         printf("PADRE: Voy a esperar a que terminen mis hijos...\n");
32         printf("PADRE: Ha terminado mi hijo %d\n", wait(NULL));
33         printf("PADRE: Ha terminado mi hijo %d\n", wait(NULL));
34     }
35     else {
36         printf("Ha habido algun error al llamar a fork\n");
37     }
38 }
39 return 0; //return 0 agregado
40 }
```

Resolución:

Escribimos el código:



```
rodrigo@fedora:~/Documentos/Practica4 - gedit ejercicio3.c
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio3.c

ejercicio3.c
~/Documentos/Practica4

1 #include <stdio.h>
2 #include <unistd.h> //libreria agregada
3 #include <sys/wait.h> //libreria agregada
4
5 int main(void)
6 {
7     int pid;
8
9     printf("Hasta aqui hay un unico proceso...\n");
10    printf("Primera llamada a fork...\n"); /* Creamos un nuevo proceso. */
11
12    pid = fork();
13
14    if (pid == 0) { /* Proceso hijo 1 */
15        printf("HIJ01: Hola, yo soy el primer hijo...\n");
16        printf("HIJ01: Voy a pararme durante 20 seg. y luego terminare\n");
17        sleep(20);
18    }
19    else if (pid > 0) { /* Proceso padre */
20        printf("PADRE: Hola, soy el padre. El pid de mi hijo es: %d\n", pid);
21
22        /* Creamos un nuevo proceso. */
```

Compilamos y ejecutamos el código:

```
rodrigo@fedora:~/Documentos/Practica4
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio3.c
rodrigo@fedora:~/Documentos/Practica4$ gcc ejercicio3.c -o Ejercicio3
rodrigo@fedora:~/Documentos/Practica4$ ./Ejercicio3
Hasta aqui hay un unico proceso...
Primera llamada a fork...
HIJ01: Hola, yo soy el primer hijo...
PADRE: Hola, soy el padre. El pid de mi hijo es: 8740
HIJ01: Voy a pararme durante 20 seg. y luego terminare
PADRE: Hola otra vez. Pid de mi segundo hijo: 8741
PADRE: Voy a esperar a que terminen mis hijos...
HIJ02: Hola, soy el segundo hijo...
HIJ02: El segundo hijo va a ejecutar la orden 'ls'...
Actividad1  Actividad2  Ejercicio1  Ejercicio2  Ejercicio3
actividad1.c actividad2.c ejercicio1.c ejercicio2.c ejercicio3.c
PADRE: Ha terminado mi hijo 8741
PADRE: Ha terminado mi hijo 8740
rodrigo@fedora:~/Documentos/Practica4$
```

Flujo del código:

- 1.- **Inicio del proceso principal:** se imprime un mensaje inicial que indica que solo hay un proceso en ejecución.
- 2.- **Primera llamada a fork():** se realiza la primera llamada a `fork()`. El proceso padre crea un nuevo proceso hijo (Hijo 1). El valor de `pid` será 0 en el hijo y el PID del hijo en el padre.
- 3.- **Código en el proceso hijo 1:** si `pid == 0`, estamos en el contexto del primer hijo, que imprime un mensaje y se detiene durante 20 segundos.
- 4.- **Código en el Proceso Padre:** si `pid > 0`, estamos en el contexto del padre, que imprime el PID de su primer hijo y realiza otra llamada a `fork()` para crear un segundo hijo.
- 5.- **Segunda llamada a fork():** el padre ejecuta otra llamada a `fork()` y crea un segundo proceso hijo (Hijo 2).
- 6.- **Código en el proceso hijo 2:** si `pid == 0`, estamos en el contexto del segundo hijo, que intenta ejecutar el comando `ls`.
- 7.- **Código en el proceso padre (de nuevo):** si `pid > 0`, el padre imprime el PID de su segundo hijo y un mensaje indicando que va a esperar a que ambos hijos terminen.
- 8.- **Espera a los hijos:** el padre utiliza `wait(NULL)` para esperar la finalización de sus hijos y muestra el PID de cada uno que termina.

11.4. Ejercicio 4

Analice el siguiente código. ¿Cuántos procesos se generan?

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     /* fork a child process */fork();
7     /* fork another child process */fork();
8     /* and fork another */fork();
9     return 0;
10 }
```

Resolución:

El código genera **8 procesos** en total, ya que cada llamada a `fork()` duplica todos los procesos existentes.

- En la primera llamada, se crea un nuevo proceso hijo, aumentando el total a 2.
- En la segunda llamada, ambos procesos existentes (el padre y el primer hijo) generan un nuevo hijo, resultando en 4 procesos.
- Finalmente, en la tercera llamada, todos los procesos (los 4 anteriores) se duplican, llevando el total a 8.

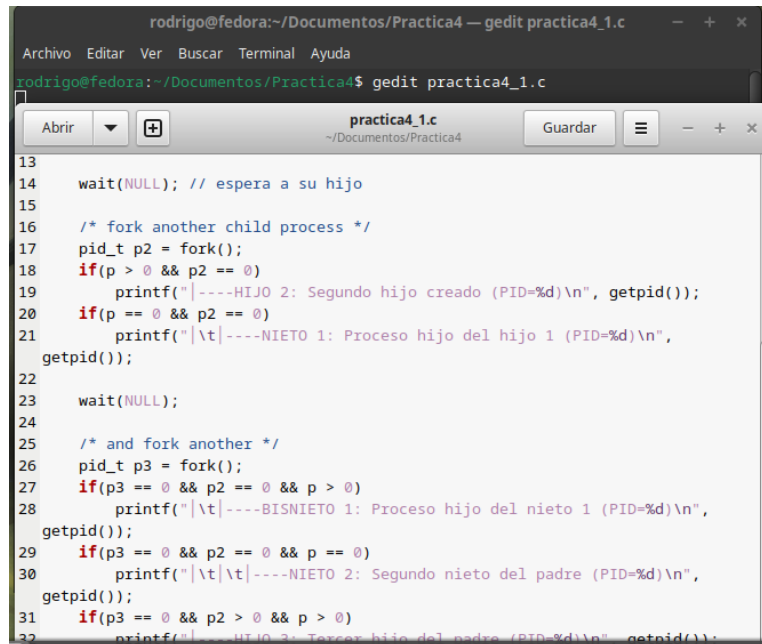
Así, después de 3 llamadas a `fork()`, la fórmula 2^n (donde n es el número de llamadas) explica que se generan $2^3 = 8$ procesos.

Código mejorado:

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4
5 int main() {
6     pid_t p = fork();
7     if(p > 0)
8         printf("PADRE: \n");
9     if(p == 0)
10         printf(" ----(HIJO 1)\n");
11
12     wait(NULL); // espera a su hijo
13
14     pid_t p2 = fork();
15     if(p > 0 && p2 == 0)
16         printf(" ----(HIJO 2)\n");
17     if(p == 0 && p2 == 0)
18         printf(" \ t ----(NIETO 1)\n");
19
20     wait(NULL);
21     pid_t p3 = fork();
22     if(p3 == 0 && p2 == 0 && p > 0)
23         printf(" \ t ----(BISNIETO 1)\n");
24     if(p3 == 0 && p2 == 0 && p == 0)
25         printf(" \ t \ t ----(NIETO 2)\n");
```

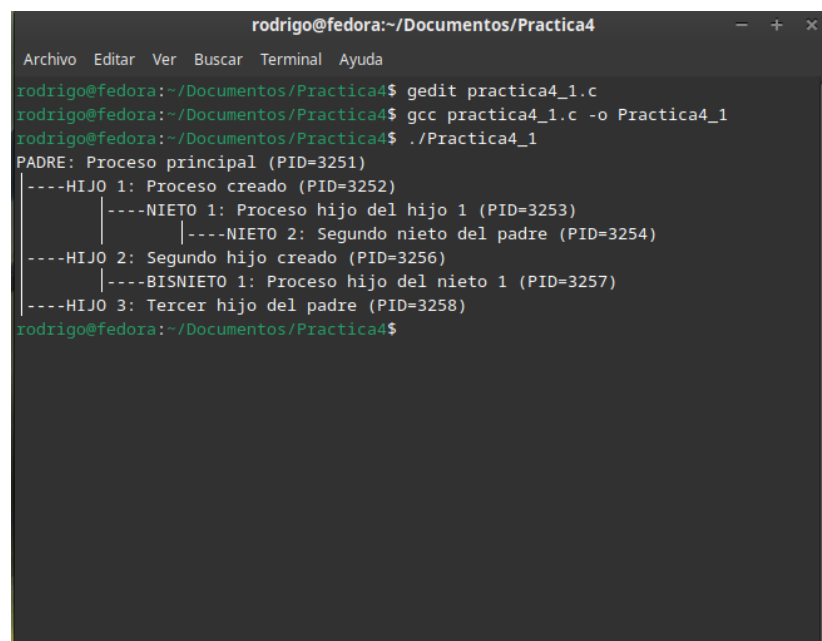
```
26     if(p3 == 0 && p2 > 0 && p > 0)
27         printf("    ----(HIJO 3)\n");
28
29     wait(NULL);
30     return 0;
31 }
```

Escribimos el código:



```
13
14     wait(NULL); // espera a su hijo
15
16     /* fork another child process */
17     pid_t p2 = fork();
18     if(p > 0 && p2 == 0)
19         printf("----HIJO 2: Segundo hijo creado (PID=%d)\n", getpid());
20     if(p == 0 && p2 == 0)
21         printf("\t|----NIETO 1: Proceso hijo del hijo 1 (PID=%d)\n",
22             getpid());
23     wait(NULL);
24
25     /* and fork another */
26     pid_t p3 = fork();
27     if(p3 == 0 && p2 == 0 && p > 0)
28         printf("\t|----BISNIETO 1: Proceso hijo del nieto 1 (PID=%d)\n",
29             getpid());
30     if(p3 == 0 && p2 == 0 && p == 0)
31         printf("\t| \t|----NIETO 2: Segundo nieto del padre (PID=%d)\n",
32             getpid());
33     if(p3 == 0 && p2 > 0 && p > 0)
34         printf("\t| \t|----HIJO 3: Tercer hijo del padre (PID=%d)\n", getpid());
```

Compilamos y ejecutamos el código:



```
rodrigo@fedora:~/Documentos/Practica4
rodrigo@fedora:~/Documentos/Practica4$ gedit practica4_1.c
rodrigo@fedora:~/Documentos/Practica4$ gcc practica4_1.c -o Practica4_1
rodrigo@fedora:~/Documentos/Practica4$ ./Practica4_1
PADRE: Proceso principal (PID=3251)
----HIJO 1: Proceso creado (PID=3252)
    |----NIETO 1: Proceso hijo del hijo 1 (PID=3253)
        |----NIETO 2: Segundo nieto del padre (PID=3254)
----HIJO 2: Segundo hijo creado (PID=3256)
    |----BISNIETO 1: Proceso hijo del nieto 1 (PID=3257)
----HIJO 3: Tercer hijo del padre (PID=3258)
rodrigo@fedora:~/Documentos/Practica4$
```

11.5. Ejercicio 5

Los siguientes códigos muestran un ejemplo del problema del productor-consumidor, donde un proceso se encarga de generar información, mientras que el segundo lo lee de la memoria. Ambos códigos utilizan funciones de la API POSIX para manejar memoria compartida. Analice el código y detalle las funciones que permiten compartir memoria entre procesos. (Para poder compilar el código, agregue `-lrt` a `gcc`, y ejecute los comandos `./productor && ./consumidor`.)

Productor:

```
1 //Productor
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <fcntl.h>
6 #include <sys/shm.h>
7 #include <sys/stat.h>
8 #include <sys/mman.h>
9 #include <unistd.h>
10
11 int main() {
12     /* the size (in bytes) of shared memory object */
13     const int SIZE = 4096;
14     /* name of the shared memory object */
15     const char *name = "OS";
16     /* strings written to shared memory */
17     const char *message_0 = "Hello";
18     const char *message_1 = "World!";
19
20     /* shared memory file descriptor */
21     int fd;
22     /* pointer to shared memory object */
23     char *ptr;
24
25     /* create the shared memory object */
26     fd = shm_open(name, O_CREAT | O_RDWR, 0766);
27     /* configure the size of the shared memory object */
28     ftruncate(fd, SIZE);
29     /* memory map the shared memory object */
30     ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, fd, 0);
31     /* write to the shared memory object */
32     sprintf(ptr, "%s", message_0);
33     ptr += strlen(message_0);
34     sprintf(ptr, "%s", message_1);
35     ptr += strlen(message_1);
36
37     return 0;
38 }
```

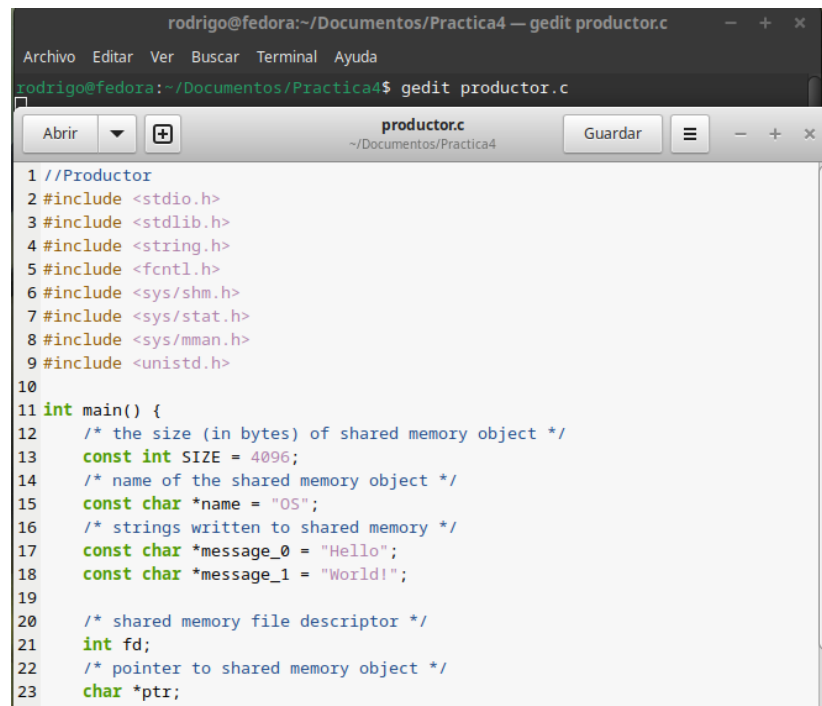
Consumidor:

```
1 //Consumidor
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <fcntl.h>
5 #include <sys/shm.h>
6 #include <sys/stat.h>
```

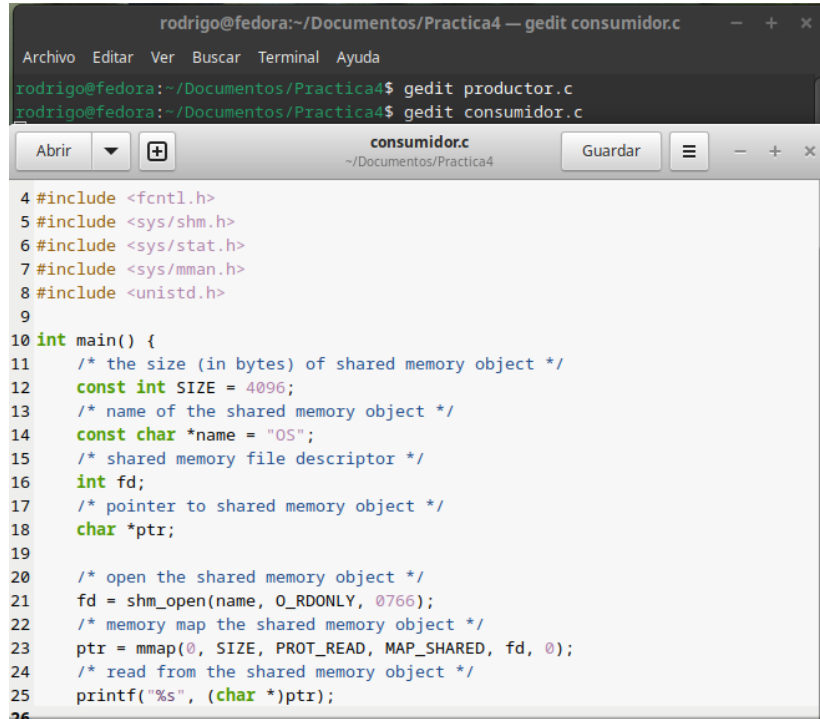
```
7 #include <sys/mman.h>
8 #include <unistd.h>
9
10 int main() {
11     /* the size (in bytes) of shared memory object */
12     const int SIZE = 4096;
13     /* name of the shared memory object */
14     const char *name = "OS";
15     /* shared memory file descriptor */
16     int fd;
17     /* pointer to shared memory object */
18     char *ptr;
19
20     /* open the shared memory object */
21     fd = shm_open(name, O_RDONLY, 0766);
22     /* memory map the shared memory object */
23     ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, fd, 0);
24     /* read from the shared memory object */
25     printf("%s", (char *)ptr);
26
27     /* remove the shared memory object */
28     shm_unlink(name);
29
30     return 0;
31 }
```

Resolución:

Escribimos el código de productor:



Escribimos el código de consumidor:

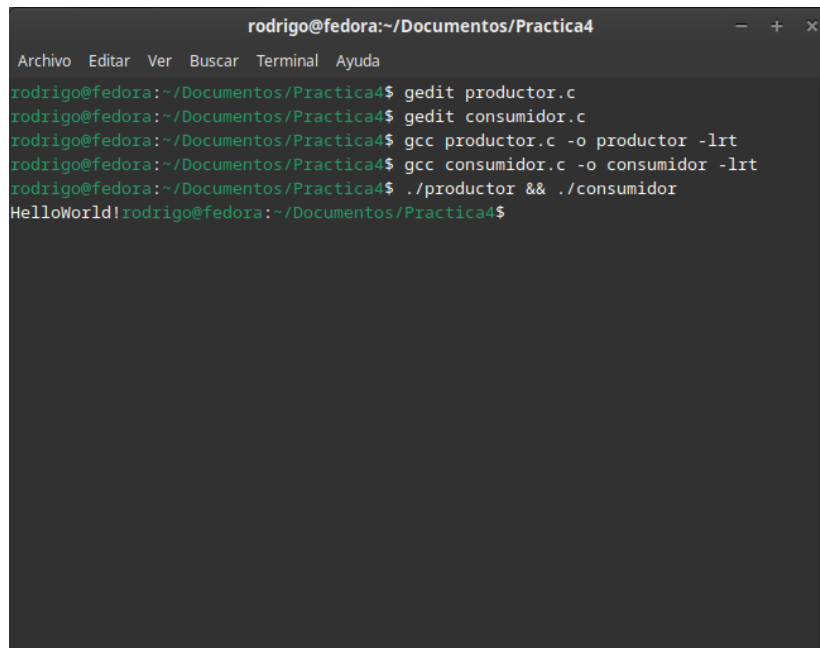


```
rodrigo@fedora:~/Documentos/Practica4 — gedit consumidor.c
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit productor.c
rodrigo@fedora:~/Documentos/Practica4$ gedit consumidor.c

consumidor.c
~/Documentos/Practica4

4 #include <fcntl.h>
5 #include <sys/shm.h>
6 #include <sys/stat.h>
7 #include <sys/mman.h>
8 #include <unistd.h>
9
10 int main() {
11     /* the size (in bytes) of shared memory object */
12     const int SIZE = 4096;
13     /* name of the shared memory object */
14     const char *name = "OS";
15     /* shared memory file descriptor */
16     int fd;
17     /* pointer to shared memory object */
18     char *ptr;
19
20     /* open the shared memory object */
21     fd = shm_open(name, O_RDONLY, 0766);
22     /* memory map the shared memory object */
23     ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, fd, 0);
24     /* read from the shared memory object */
25     printf("%s", (char *)ptr);
26 }
```

Compilamos y ejecutamos los códigos con `-lrt` y los comandos `./productor && ./consumidor`:



```
rodrigo@fedora:~/Documentos/Practica4
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit productor.c
rodrigo@fedora:~/Documentos/Practica4$ gedit consumidor.c
rodrigo@fedora:~/Documentos/Practica4$ gcc productor.c -o productor -lrt
rodrigo@fedora:~/Documentos/Practica4$ gcc consumidor.c -o consumidor -lrt
rodrigo@fedora:~/Documentos/Practica4$ ./productor && ./consumidor
HelloWorld!rodrigo@fedora:~/Documentos/Practica4$
```

Funciones que permiten compartir memoria entre procesos:

shm_open(): crea o abre un objeto de memoria compartida.

- **Productor:** `fd = shm_open(name, O_CREAT | O_RDWR, 0766);`
`O_CREAT` indica que se debe crear el objeto si no existe y `O_RDWR` permite lectura y escritura.
- **Consumidor:** `fd = shm_open(name, O_RDONLY, 0766);`
`O_RDONLY` permite solo lectura del objeto existente.

ftruncate(): establece el tamaño del objeto de memoria compartida.

- **Productor:** `ftruncate(fd, SIZE);`
define el tamaño del segmento de memoria compartida (4096 bytes).

mmap(): mapea el objeto de memoria compartida en el espacio de direcciones del proceso.

- **Productor:** `ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, fd, 0);`
`PROT_WRITE` permite escritura; `MAP_SHARED` asegura que las modificaciones son visibles para otros procesos.
- **Consumidor:** `ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, fd, 0);`
`PROT_READ` permite lectura.

shm_unlink(): elimina el objeto de memoria compartida, liberando recursos del sistema.

- **Consumidor:** `shm_unlink(name);`
se utiliza al final del programa del consumidor para limpiar el objeto de memoria compartida.

12. Cuestionario

12.1. ¿Cuál es la principal característica de crear un proceso utilizando la función FORK?

La principal característica de crear un proceso utilizando la función `fork()` es que genera un nuevo proceso hijo que es una copia del proceso padre.

Ambos procesos pueden ejecutarse simultáneamente y tienen espacios de memoria independientes, lo que significa que las modificaciones en uno no afectan al otro. `fork()` retorna 0 al hijo y el PID del hijo al padre, permitiendo identificar y gestionar los procesos de manera efectiva.

Esto es fundamental para poder implementar programación multitarea en diversos sistemas operativos en las cuales lo puedan permitir.

12.2. ¿Cuántos procesos FORK se pueden crear de forma secuencial? ¿Existe algún límite establecido por el sistema operativo?

El número de procesos que se pueden crear de forma secuencial con `fork()` no tiene un límite específico en el estándar de C o POSIX.

Las limitaciones están relacionadas con:

- **Recursos del sistema:** cada `fork()` consume memoria y descriptores de archivo.
- **Límite de procesos:** la mayoría de los sistemas operativos imponen un límite en el número de procesos que un usuario puede crear, que puede configurarse con comandos como `ulimit`.
- **Carga del sistema:** crear muchos procesos puede afectar el rendimiento del sistema.
- **Jerarquía de procesos:** cada `fork()` duplica el número de procesos existentes, lo que puede llevar a un crecimiento exponencial.

12.3. ¿Qué trabajo realiza la función EXECLP? Explique utilizando un ejemplo de utilización del comando.

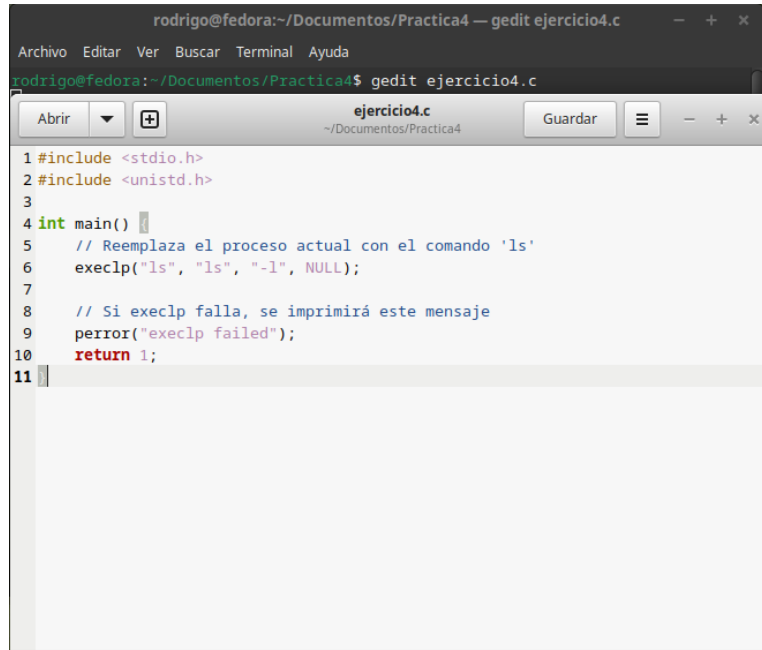
La función `execlp()` reemplaza el proceso actual con un nuevo programa. Cuando se invoca, el proceso que llama a `execlp()` se convierte en el programa especificado, y su código y espacio de datos se reemplazan por los del nuevo programa. Si la llamada a `execlp()` tiene éxito, no hay retorno al proceso original.

Ejemplo de utilización:

Supongamos que tenemos un programa en C que utiliza `execlp()` para ejecutar el comando `ls`. El código sería::

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
5     // Reemplaza el proceso actual con el comando 'ls'
6     execlp("ls", "ls", "-l", NULL);
7
8     // Si execlp falla, se imprimir este mensaje
9     perror("execlp failed");
10    return 1;
11 }
```

Escribimos el código:



```
rodrigo@fedora:~/Documentos/Practica4 — gedit ejercicio4.c
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio4.c
ejercicio4.c
~/Documentos/Practica4
Guardar
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
5     // Reemplaza el proceso actual con el comando 'ls'
6     execlp("ls", "ls", "-l", NULL);
7
8     // Si execlp falla, se imprimirá este mensaje
9     perror("execlp failed");
10    return 1;
11 }
```

Compilamos y ejecutamos el código:

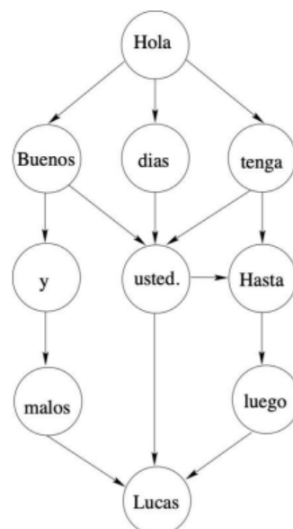
```
rodrigo@fedora:~/Documentos/Practica4
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit ejercicio4.c
rodrigo@fedora:~/Documentos/Practica4$ gcc ejercicio4.c -o Ejercicio4
rodrigo@fedora:~/Documentos/Practica4$ ./Ejercicio4
total 192
-rwxr-xr-x. 1 rodrigo rodrigo 16824 oct  6 13:01 Actividad1
-rw-r--r--. 1 rodrigo rodrigo  184 oct  6 12:59 actividad1.c
-rwxr-xr-x. 1 rodrigo rodrigo 16944 oct  6 17:56 Actividad2
-rw-r--r--. 1 rodrigo rodrigo  480 oct  6 17:55 actividad2.c
-rwxr-xr-x. 1 rodrigo rodrigo 16824 oct  6 20:18 consumidor
-rw-r--r--. 1 rodrigo rodrigo  773 oct  6 20:16 consumidor.c
-rwxr-xr-x. 1 rodrigo rodrigo 16800 oct  6 18:44 Ejercicio1
-rw-r--r--. 1 rodrigo rodrigo  388 oct  6 18:44 ejercicio1.c
-rwxr-xr-x. 1 rodrigo rodrigo 16912 oct  6 19:06 Ejercicio2
-rw-r--r--. 1 rodrigo rodrigo  523 oct  6 19:06 ejercicio2.c
-rwxr-xr-x. 1 rodrigo rodrigo 16920 oct  6 19:35 Ejercicio3
-rw-r--r--. 1 rodrigo rodrigo  1483 oct  6 19:35 ejercicio3.c
-rwxr-xr-x. 1 rodrigo rodrigo 16720 oct  6 20:44 Ejercicio4
-rw-r--r--. 1 rodrigo rodrigo  245 oct  6 20:44 ejercicio4.c
-rwxr-xr-x. 1 rodrigo rodrigo 16880 oct  6 20:18 productor
-rw-r--r--. 1 rodrigo rodrigo  1041 oct  6 20:15 productor.c
rodrigo@fedora:~/Documentos/Practica4$
```

Cuando ejecutamos `./listar`, el proceso actual se reemplazará con el comando `ls`, y observaremos la salida del listado de archivos y directorios en el directorio actual, similar a lo que obtendríamos al ejecutar `ls -l` en la terminal.

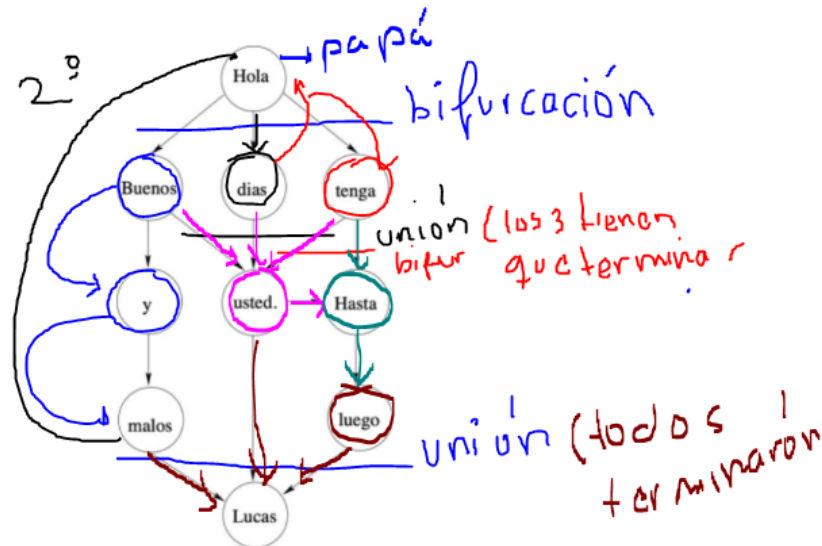
Si `execlp()` falla, el programa imprimirá un mensaje de error gracias a la función `perro()`, que nos dirá la razón del fallo.

12.4. Implementación

Implementar el grafo de precedencia de la figura utilizando las llamadas al sistema `fork` y `wait` (no utilizar `waitpid`). El grupo de sentencias a ejecutar en cada nodo del grafo se simularán mediante la sentencia `printf("cadena")`, donde "cadena" es la cadena de caracteres que contiene cada nodo de la figura. La frase deberá aparecer en una única línea:



Resolución:



Resultado: Hola Buenos y malos dias tenga usted. Hasta luego Lucas

El código sería:

```

1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <sys/wait.h>
5
6 void create() {
7     pid_t pid_hola, pid_buenos, pid_y, pid_malos, pid_dias, pid_tenga, pid_usted, pid_hasta,
8         pid_luego, pid_lucas;
9     // Proceso padre: "Hola"
10    if ((pid_hola = fork()) == 0) {
11        printf("Hola ");
12        fflush(stdout);
13        // Proceso hijo: "Buenos"
14        if ((pid_buenos = fork()) == 0) {
15            printf("Buenos ");
16            fflush(stdout);
17            // Proceso hijo: "y"
18            if ((pid_y = fork()) == 0) {
19                printf("y ");
20                fflush(stdout);
21                // Proceso hijo: "malos"
22                if ((pid_malos = fork()) == 0) {
23                    printf("malos ");
24                    fflush(stdout);
25                    // Proceso hijo: "dias"
26                    if ((pid_dias = fork()) == 0) {
27                        printf("das ");
28                        fflush(stdout);
29                        // Proceso hijo: "tenga"
30                        if ((pid_tenga = fork()) == 0) {

```

```
30         printf("tenga ");
31         fflush(stdout);
32         // Proceso hijo: "usted."
33         if ((pid_usted = fork()) == 0) {
34             printf("usted. ");
35             fflush(stdout);
36             // Proceso hijo: "Hasta"
37             if ((pid_hasta = fork()) == 0) {
38                 printf("Hasta ");
39                 fflush(stdout);
40                 // Proceso hijo: "luego"
41                 if ((pid_luego = fork()) == 0) {
42                     printf("luego ");
43                     fflush(stdout);
44                     // Proceso hijo: "Lucas"
45                     if ((pid_lucas = fork()) == 0) {
46                         printf("Lucas\n");
47                         fflush(stdout);
48                         exit(0); // Finaliza el proceso
49                     }
50                     wait(NULL); // Espera a "Lucas"
51                     exit(0);
52                 }
53                 wait(NULL); // Espera a "luego"
54                 exit(0);
55             }
56             wait(NULL); // Espera a "Hasta"
57             exit(0);
58         }
59         wait(NULL); // Espera a "usted."
60         exit(0);
61     }
62     wait(NULL); // Espera a "das"
63     exit(0);
64 }
65 wait(NULL); // Espera a "malos"
66 exit(0);
67 }
68 wait(NULL); // Espera a "y"
69 exit(0);
70 }
71 wait(NULL); // Espera a "Buenos"
72 exit(0);
73 }
74 wait(NULL); // Espera a "Hola"
75 exit(0);
76 }
77 wait(NULL); // Espera a "Hola"
78 }
79 int main() {
80     create();
81     return 0;
82 }
```

Compilamos y ejecutamos el código:

```
rodrigo@fedora:~/Documentos/Practica4
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica4$ gedit implementacion.c
rodrigo@fedora:~/Documentos/Practica4$ gcc implementacion.c -o Implementacion
rodrigo@fedora:~/Documentos/Practica4$ ./Implementacion
Hola Buenos y malos días tenga usted. Hasta luego Lucas
rodrigo@fedora:~/Documentos/Practica4$
```