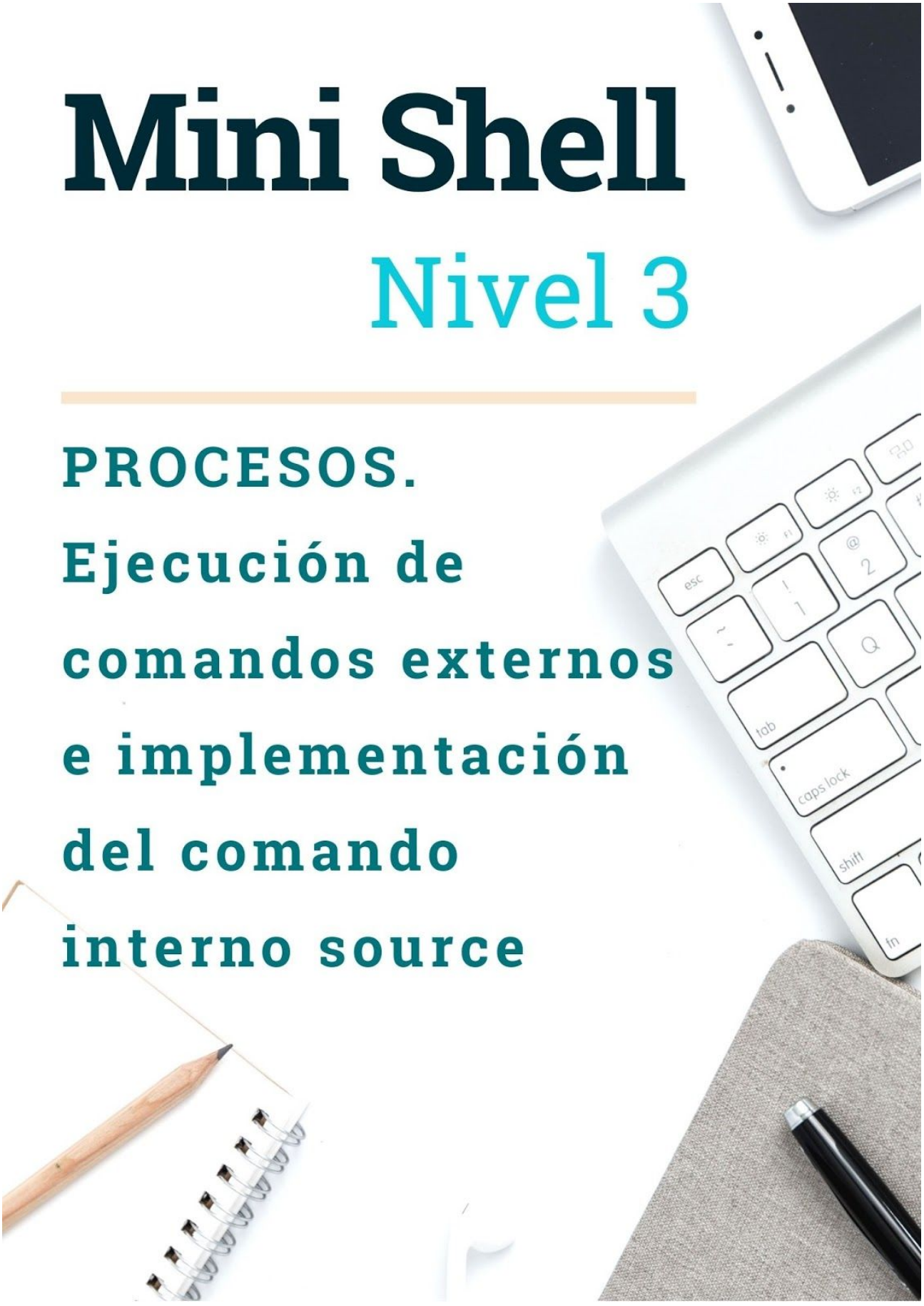
Mini Shell

Nivel 3

PROCESOS.

Ejecución de comandos externos e implementación del comando interno source



Indice

Mini Shell
Nivel 3

PROCESOS. Ejecución comandos externos e implementación comando interno source

En este nivel prepararemos nuestro minishell para que sea capaz de ejecutar, a través de la llamada al sistema **execvp()**, los comandos que no sean internos. Cada vez que detectemos que se trata de un comando externo, el minishell (proceso padre) creará un hijo que se encargue de él.
También seguiremos implementando comandos internos, en concreto el **source** y además conseguiremos salir del minishell pulsando Ctrl+D.

¿Cuáles son los retos de este nivel?

1. Ampliar la función **execute_line**() para permitir ejecutar también los comandos externos.
2. Implementar la función **internal_source**() para el comando interno **source** que ejecutará las líneas de comandos de un fichero tipo script.
3. Poder salir del minishell pulsando Ctrl+D.

¿Qué conceptos trabajamos con estos retos?

- Conceptos de proceso, proceso zombie y proceso huérfano
- El funcionamiento de las llamadas al sistema **fork**(), **wait**(), **exec**(), **getpid**(), **getppid**() y **exit**(), y el comando **ps** del shell
- (Gestión de procesos)

¿Cuáles son los detalles de implementación?

Hay que ampliar las siguientes funciones:

int execute_line(char *line);
En caso de que no se trate de un comando interno (cd, export, source, jobs, exit, fg o bg), se crea un hijo con **fork**() quien realizará la llamada al sistema **execvp**()[1] para ejecutar el comando externo solicitado. Si la ejecución del comando ha fallado entonces mostrar el error por la salida *stderr* y realizar un **exit**().

[1] La ejecución de los comandos NO se hará a través de la llamada al sistema **system**(). Tanto para los comandos que procedan del script, como para los que se introduzcan directamente en la línea de comandos, utilizaremos nuevos procesos creados con **fork**() y enviaremos las órdenes a **execvp**().

En este nivel, como todavía no sabemos lanzar procesos en segundo plano, existe un solo hijo y se ejecuta en primer plano. Utilizaremos un **wait**() para que el padre espere a recibir la información del cambio de estado del hijo. El cambio de estado puede indicar que el hijo ha finalizado, que ha sido detenido por una señal o que ha sido reanudado por una señal. En caso de finalización del hijo, ejecutar el wait permite al sistema liberar los recuros asociados al hijo y que éste no quede en estado zombie.

Ej: **wait(&status)**[2] o **wait(NULL)**

A modo de testeo en este nivel, mostraremos por pantalla el PID del padre y del hijo, y también informaremos cuando el hijo finaliza y con qué estado.

int internal_source(char **args);
Se comprueban los argumentos y se muestra la sintaxis en caso de no ser correcta.
Mediante la función **fopen**()[3] se abre en modo lectura el fichero de comandos especificado por consola.
Se indica error si el fichero no existe.
Se va leyendo línea a línea el fichero mediante **fgets**() y se pasa la línea leída a nuestra función execute_line(). Hay que realizar un **fflush** del stream del fichero tras leer cada línea.
Se cierra el fichero de comandos con **fclose**().

char *read_line(char *line);
Cuando se pulsa Ctrl+D al comienzo de una línea significa el final de la entrada stdin o sea EOF (no es una señal en el sentido Unix). Podemos averiguar si se ha producido tal situación utilizando la función feof().

Hay que controlar que si el resultado de fgets es NULL, y se ha producido un EOF (o sea feof(stdin)), hay que salir con exit.

```
ptr = fgets(line, COMMAND_LINE_SIZE, stdin); // leer linea
```

[2] Así podemos utilizar la macro **WIFEXITED**(status) para saber si un proceso ha finalizado con exit() y en tal caso mostrar el estado de finalización con la macro **WEXITSTATUS**(status), y en caso contrario, la macro **WIFSIGNALED**(status) nos indicará si el proceso ha finalizado por una señal, y podremos mostrar cual imprimiendo el resultado de la macro **WTERMSIG**(status).
[3] En este caso nos interesa más emplear fopen() que open() ya que no necesitamos el descriptor del fichero devuelto por open() sino el puntero al stream de datos que devuelve el fopen(), para utilizarlo en la llamada posterior a fgets()

```
if (!ptr) {  //ptr==NULL
    printf("\r");
   if (feof(stdin)) { //feof(stdin)!=0 significa que se ha activado el indicador EOF
       printf("Bye bye\n");
       exit(0);
   }
}

return ptr;
```

¿Qué recursos metemos en nuestras mochilas?

Documentos de apoyo:

- Procesos. Llamadas al sistema con ejemplos prácticos.
- Tutorial - Write a shell in C

¿Cómo comprobamos que hemos superado este nivel?

- Con vuestras propias pruebas, mediante los mensajes por pantalla provisionales, que muestran la escisión de proceso padre e hijo, la finalización del hijo y su estado, las líneas ejecutadas del fichero externo
- Y también podéis ejecutar vuestro programa **nivel3.c** e ir escribiendo en la línea de comandos los ejemplos que se exponen a continuación, comparando vuestros resultados con los mostrados aquí

```
uib:$ ./nivel3
$ pws
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6945]
pws: no se encontró la orden
[execute_line()→ Proceso hijo 6945 finalizado con estado: 1]
$ pwd
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6947]
```

```
/home/uib/practicasSO/SOI/miP2/niveles
[execute_line()→ Proceso hijo 6947 finalizado con estado: 0]
$ cd ..
$ pwd
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6948]
/home/uib/practicasSO/SOI/miP2
[execute_line()→ Proceso hijo 6948 finalizado con estado: 0]
$ cd niveles
$ cd nuevo
chdir: No such file or directory
$ printenv USER
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6952]
uib
[execute_line()→ Proceso hijo 6952 finalizado con estado: 0]
$ export USER=Yo
$ printenv USER
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6958]
Yo
[execute_line()→ Proceso hijo 6958 finalizado con estado: 0]
$ export
Error de sintaxis. Uso: export Nombre=Valor
$ cat comandos1.sh [4]
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6960]
ls
date
file nivel3.c #muestra tipo
ps
hola
sleep 1
[execute_line()→ Proceso hijo 6960 finalizado con estado: 0]
$ source
Error de sintaxis. Uso: source <nombre_fichero>
$ source comandos1.sh
[internal_source()→ LINE: ls
]
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6966]
nivel1.c  nivel1.o  nivel1  nivel2.c  nivel2.o  nivel2  nivel3.c  nivel3.o  nivel3
```

[4] Hay que tenerlo creado previamente y con ese contenido. Cuando implementemos la redirección podremos crearlo desde nuestro mini shell.

```
comandos1.sh   Makefile
[execute_line()→ Proceso hijo 6966 finalizado con estado: 0]

[internal_source()→ LINE: date
]
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6967]
Mon Nov  6 12:40:06 CET 2017
[Proceso hijo 6967 finalizado con exit(), estado: 0]

[internal_source()→ LINE: file nivel3.c #muestra tipo
]
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6968]
nivel3.c: C source, UTF-8 Unicode text
[execute_line()→ Proceso hijo 6968 finalizado con estado: 0]

[internal_source()→ LINE: ps
]
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6969]
  PID TTY          TIME CMD
 2556 pts/2    00:00:00 bash
 6943 pts/2    00:00:00 nivel3
 6969 pts/2    00:00:00 ps
[execute_line()→ Proceso hijo 6969 finalizado con estado: 0]

[internal_source()→ LINE: hola
]
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6970]
hola: no se encontró la orden
[Proceso hijo 6970 finalizado con exit(), estado: 255]

[internal_source()→ LINE: sleep 1
]
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6971]
[execute_line()→ Proceso hijo 6971 finalizado con estado: 0]
$ source comandos #no existe
fopen: No such file or directory
$ sleep 60
[execute_line()→ PID padre: 6943]
[execute_line()→ PID hijo: 6976]
^C
```

```
uib:$
```

Observación: Al pulsar CTRL+C en este nivel el minishell aborta y ya no sale el prompt de nuestro programa. En el siguiente nivel **el minishell no deberá abortar con CTRL+C**, tan sólo interrumpir el proceso que se ejecuta en primer plano.

Anexo 1: Concepto de proceso, proceso zombie y proceso huérfano

Proceso:
Programa en ejecución. Un **programa** será un archivo que reside en el disco de sistema y que se crea con otros programas, como por ejemplo el compilador de C o el editor nano, mientras que un **proceso** es una copia del programa en la memoria que está ejecutándose. Un proceso comprende todo el **entorno de ejecución** del programa, es decir, desde las variables, los ficheros abiertos, el directorio en el que reside, información acerca de usuario que ejecuta el proceso y el terminal donde lo ejecuta así como el código del programa.
Características:

- Un proceso consta de código, datos, pila y montón.
- Los procesos existen en una jerarquía de árbol (UNIX).
- El sistema asigna un identificador de proceso (PID) único al iniciar el proceso.
- El planificador de tareas asigna un tiempo compartido para el proceso según su prioridad (sólo *root* puede cambiar prioridades).

Proceso zombie (defunct):
Un proceso zombi o "defunct" es un proceso que ha completado su ejecución (toda la memoria y recursos asociados con dicho proceso han sido liberados) pero aún tiene una entrada en la tabla de procesos, la cual no se eliminará hasta que el proceso que lo ha creado lea el estado de su salida mediante la llamada al sistema wait()[5].

Proceso huérfano:
Proceso en ejecución cuyo padre ha finalizado. El nuevo identificador de proceso padre (PPID), en Linux, coincide con el identificador del proceso **init** que es 1 (en Ubuntu fue sustituído por **systemd**[6], y en el caso de procesos que desciendan del bash si se quedan huérfanos adoptan el PPID del proceso **upstart**).

[5] Cuando un proceso tiene un hijo mediante una llamada al sistema fork, el valor que devuelve la función main ya no es devuelto al Sistema Operativo, sino que es devuelto a su proceso padre. Para que el proceso padre pueda recibir ese código de retorno del hijo, ha de efectuar una llamada al sistema wait
[6] https://es.wikipedia.org/wiki/Systemd

Anexo 2: Funcionamiento de las llamadas al sistema fork(), wait(), exec(), getpid(), getppid() y exit(), y el comando ps del shell

- La llamada al sistema **fork**() cambia un proceso en 2 procesos idénticos, conocidos como el padre y el hijo. Los procesos tienen un único identificador, que será diferente en cada ejecución. Es imposible indicar con antelación qué proceso obtendrá la CPU pero se puede detectar fácilmente si el proceso en ejecución es el hijo o el padre ya que **fork() devuelve 0 al proceso hijo y el PID *(process identification)* del hijo al proceso padre** (-1, si error).
- El proceso hijo es una copia exacta del proceso padre, de hecho el proceso hijo tiene una copia del espacio de datos, heap (memoria dinámica) y de la pila. Sin embargo, el espacio de direcciones de datos del proceso padre será distinto del espacio de direcciones del proceso hijo, lo cual implica que **ambos procesos no comparten los datos**, sólo se hace un copia para el hijo. Habitualmente el padre y el hijo compartirán el segmento de código ya que éste es de sólo lectura.
- El PPID *(parent process identification)* del hijo coincide con el PID del padre, a menos que quede huérfano. Podemos obtener el PID y el PPID de un proceso mediante **getpid**() y **getppid**() respectivamente.
- En realidad, todos los procesos en algún momento son hijos, todos menos el **proceso init** (o **systemd**). En el caso de que un proceso sea creado mediante el shell (ejecutado desde éste), el shell será el padre.
- El **comando ps** de UNIX lista los procesos que se están ejecutando, mostrando también el PID y el nº de terminal, con `$ ps -f` (formato completo) veremos también el PPID.
    - Su versión gráfica es pstree, ejecutando `$ pstree -p` veremos el árbol de procesos en ejecución junto con su nº de PID. Si queremos ver sólo la rama de un proceso desde su padre, podemos ejecutar `$ pstree <PPID>`
    - Con `$ ps f` podemos ver el estado de los procesos[7] y un árbol jerárquico con caracteres ASCII.

```
uib:$ ./nivel3
```

[7] D uninterruptible sleep (usually IO)
R runnable (Ejecutando)
S sleeping (Bloqueado)
T traced or stopped
Z a defunct ("zombie") process

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
uib       5070  5053  0 11:16 pts/18   00:00:00 bash
uib       6841  5070  0 12:28 pts/18   00:00:00 ./nivel3
uib       6872  6841  0 12:28 pts/18   00:00:00 ps -f
$ pstree 5053
gnome-terminal-┬─┬─bash─────nivel3─────pstree
               ├─{dconf worker}
               ├─{gdbus}
               └─{gmain}
$ ps f
  PID    TTY      STAT   TIME    COMMAND
 5070    pts/18   Ss     0:00    bash
 6841    pts/18   S+     0:00    \_ ./nivel3
 6877    pts/18   R+     0:00    \_ ps f
$ exit
uib:$ ps f
  PID    TTY      STAT   TIME    COMMAND
 5070    pts/18   Ss     0:00    bash
```

  - Además podemos ver los procesos que se ejecutan en un terminal concreto mediante `$ ps -t <num_terminal>`
  - Si queremos ver nuestra lista de procesos asociados a algún terminal, en formato orientado a usuario, podemos obtenerla con `$ ps -u`
  - y también podemos obtener toda la lista de procesos de la máquina, con todos los detalles, mediante `$ ps -ef`
  - Mediante la línea de comandos `$ ps -el | grep 'Z'` podemos listar los procesos zombies
- La llamada al sistema wait() fuerza al proceso padre a esperar a que un proceso hijo se detenga o termine, permitiendo obtener su estado de salida. Devuelve el PID del hijo, o -1 si se produce un error. Si un proceso solicita información mediante wait y esta información no está disponible, el proceso abandona el estado de listo y pasa al estado de dormido (bloqueado). Este estado se caracteriza porque el proceso no consume CPU, es decir, no sustrae tiempo de cálculo a otros procesos. Cuando la información está disponible, el proceso se desbloquea y continúa su ejecución. A efectos del programador, el programa se para en la llamada al sistema hasta que la información esté disponible.

```
wait(&status); //o también wait(NULL)
```

Existe otra llamada al sistema similar, waitpid(), que tiene una versión bloqueante:

```
waitpid(-1,&estado,0)
```

que es equivalente a wait(&estado), y otra no bloqueante:

```
while (waitpid(-1,&estado,WNOHANG)==0)
      /* espera activa */;
```

Normalmente se usan las llamadas no bloqueantes en lo que se denomina espera activa. En esta espera, se sondea continuamente mediante la llamada no bloqueante para ver si la información está disponible. En sistemas multiprogramados esta técnica consume innecesariamente tiempo de CPU que podrían aprovechar otros procesos.

- La familia de llamadas al sistema **exec** reemplaza (muta) el espacio de direcciones del proceso con un nuevo programa (el proceso es el mismo).Todo el contenido del espacio de direcciones cambia: código, datos, pila, etc. Se reinicia el contador de programa a la primera instrucción (main). Se mantiene todo lo relacionado con la identidad del proceso (el PID del proceso no varía). Esta familia de funciones devuelve -1 en caso de error, en caso contrario **no retorna** (las instrucciones que pongamos a continuación sólo se ejecutarán en caso fallido!!!). Utilizando execvp() busca dentro del PATH si no se especifica una ruta.
- exit() termina la ejecución de un proceso.

Ejemplos de uso de las funciones fork(), wait(), exec(), getpid(), getppid() y exit()

```
// uso_fork.c - Funciones getpid() y getppid() - Adelaida Delgado
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main() {
  pid_t pid;

  pid = fork();
  if (pid == 0) { // hijo
     printf("HIJO: getpid(), o sea PID del proceso hijo: %d\n", getpid());
     printf("HIJO: getppid(), o sea PID del proceso padre: %d\n", getppid());
     exit(EXIT_SUCCESS); // IMPORTANTE!!!
  } else if (pid >0) { // padre
     printf("PADRE: pid recibido de fork(), o sea PID del proceso hijo: %d\n", pid);
```

```
    printf("PADRE: getpid() o sea PID del proceso padre: %d\n", getpid());
    printf("PADRE: getppid() o sea PID del proceso padre del padre: %d\n", getppid());
}
else { //pid <0 error
    perror("fork");
    exit(EXIT_FAILURE);
}
return EXIT_SUCCESS;
}
```

Ejemplos de ejecuciones de uso_fork.c:[8]

```
$ ./uso_fork
PADRE: pid recibido de fork(), o sea PID del proceso hijo: 8587
PADRE: getpid() o sea PID del proceso padre: 8586
HIJO: getpid(), o sea PID del proceso hijo: 8587
PADRE: getppid() o sea PID del proceso padre del padre: 7716
HIJO: getppid(), o sea PID del proceso padre: 8586
$ ./uso_fork
PADRE: pid recibido de fork(), o sea PID del proceso hijo: 8591
HIJO: getpid(), o sea PID del proceso hijo: 8591
PADRE: getpid() o sea PID del proceso padre: 8590
HIJO: getppid(), o sea PID del proceso padre: 8590
PADRE: getppid() o sea PID del proceso padre del padre: 7716
$ ./uso_fork
PADRE: pid recibido de fork(), o sea PID del proceso hijo: 8595
PADRE: getpid() o sea PID del proceso padre: 8594
PADRE: getppid() o sea PID del proceso padre del padre: 7716
HIJO: getpid(), o sea PID del proceso hijo: 8595
HIJO: getppid(), o sea PID del proceso padre: 8594
```

- Ejecutarlo varias veces y comprobar que unas veces acaba antes el padre y otras el hijo y cómo se intercala la ejecución de ambos[9].
- Ejecutar el **comando ps** y comprobar cuál es el PID del bash (compararlo con el PID del padre del padre).
- Comprobad si a veces el PID del proceso padre que muestra el hijo no concuerda con el PID del padre que había mostrado el padre. ¿Por qué? ¿A quien pertenece entonces ese PID? (averiguarlo realizando `$ ps <PID>` o `$ pstree <PID>`). Más información

[8] Fijarse que en cuanto se produce el fork() no se sabe quién (padre / hijo) tendrá el procesador en cada momento y que incluso el hijo puede quedar huérfano
[9] Se puede observar mejor el efecto poniendo un usleep(10) después de cada instrucción

Ejemplo de hijo zombie:

```c
/* ejzombies.c - ejemplo de hijo zombie - Adelaida Delgado*/

#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main (){
 pid_t pid;
 pid = fork ();
 if (pid > 0) { //proceso padre
   printf("Soy el padre y espero 20\" antes de terminar, no sé nada de mi hijo.\n");
   sleep (20);
 }
 else if (pid ==0){ // proceso hijo, acaba sin notificarlo al padre
   exit (0);
 }
 else { //pid <0 error
      perror("fork");
      exit(EXIT_FAILURE);
 }
 return EXIT_SUCCESS;
}
```

Lo ejecutamos y, **desde otro terminal**, comprobamos los procesos en ejecución en el terminal anterior (en el ejemplo es el 1), antes y después de finalizar el padre:

```
$ ps -t 1 f
    PID TTY      STAT   TIME COMMAND
   5005 pts/1    Ss     0:00 /bin/bash
   7717 pts/1    S+     0:00  \_ ./ejzombies
   7718 pts/1    Z+     0:00      \_ [ejzombies] <defunct>

$ ps -t 1 -f
    PID TTY      STAT   TIME COMMAND
   5005 pts/1    Ss+    0:00 /bin/bash
```

Cuando el padre de un zombi se termina, normalmente sus zombies se eliminan de la tabla de procesos.

Podemos evitar el hijo zombie usando wait():

```c
/* ejzombieevitado.c - ejemplo cómo evitar hijo zombie con wait - Adelaida Delgado */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
  int estado;
  pid_t pid;

  pid = fork();
  if (pid > 0) {
     wait(&estado);
     printf("Soy el proceso padre, mi proceso hijo ha terminado ");
     printf("y yo ahora espero 5 segundos antes de terminar\n");
     printf("estado: %d\n", estado);
     sleep(5);
  }
  else if (pid ==0){ // proceso hijo, acaba sin notificarlo al padre
     exit (0);
  }
  else { //pid <0 error
    perror("fork");
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

Vamos a ver ahora un uso combinado de fork(), execvp() y wait():

Recordemos que la familia de llamadas al sistema **exec** reemplazan la imagen del proceso en curso con una nueva. Todo el contenido del espacio de direcciones cambia: código, datos, pila, etc. Se reinicia el contador de programa a la primera instrucción (main). Se mantiene todo lo relacionado con la identidad del proceso (el PID del proceso no varía).

Esta familia de llamadas al sistema devuelve -1 en caso de error, en caso contrario **no retorna** (las instrucciones que pongamos a continuación de la llamada sólo se ejecutarán en caso fallido!!!). Utilizando execvp() busca dentro del PATH si no se especifica una ruta.

```c
/* uso_fork_exec_wait.c - Muestra el uso combinado de fork(), exec() y wait() */
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h> //pid_t

int main(void) {
  pid_t pid;
  char *args[3]={"ps", "f", NULL};

  printf("Hasta aquí hay un único proceso...\n");
  printf("Primera llamada a fork...\n");

 /* Creamos un nuevo proceso. */
  pid = fork();

  if (pid == 0) {
    printf("HIJO1: Hola, yo soy el primer hijo...\n");
    printf("HIJO1: Voy a pararme durante 10\" y luego terminaré...\n");
    sleep(10);
    exit(0);
  }
  else if (pid > 0) {
    printf("PADRE: Hola, soy el padre. Mi PID es: %d. ",getpid());
    printf("El PID de mi 1er hijo es: %d\n", pid);

    /* Creamos un nuevo proceso. */
    pid = fork();
    if (pid == 0) {
      printf("HIJO2: Hola, soy el segundo hijo...\n");
      printf("HIJO2: Esperaré 5\" y ejecutaré la orden 'ps f'\n");
      sleep(5);
      execvp(args[0], args);
      printf("HIJO2: Si ve este mensaje, el execvp no funcionó...\n");
      exit(-1);
    }
    else if (pid > 0) {
      printf("PADRE: Hola otra vez. Mi PID es: %d. ", getpid());
      printf("El PID de mi segundo hijo es: %d\n",pid);
      printf("PADRE: Voy a esperar a que terminen mis hijos...\n");
      printf("PADRE: Ha terminado mi hijo %d\n",wait(NULL));
      printf("PADRE: Ha terminado mi hijo %d\n",wait(NULL));
    }
   else {
      printf("Ha habido algún error al llamar por 2ª vez al fork()\n");
      exit(-1);
    }
  }
  else {
     printf("Ha habido algún error al llamar a fork()\n");
```

```
    exit(-1);
  }
  return 0;
}
```

Ejemplo de ejecución del programa anterior:

```
$ ./uso_fork_exec_wait
Hasta aquí hay un único proceso...
Primera llamada a fork...
PADRE: Hola, soy el padre. Mi PID es: 6530. El PID de mi 1er hijo es: 6531
HIJO1: Hola, yo soy el primer hijo...
HIJO1: Voy a pararme durante 10" y luego terminaré...
PADRE: Hola otra vez. Mi PID es: 6530. El PID de mi segundo hijo es: 6532
PADRE: Voy a esperar a que terminen mis hijos...
HIJO2: Hola, soy el segundo hijo...
HIJO2: Esperaré 5" y ejecutaré la orden 'ps -f'
  PID TTY      STAT   TIME COMMAND
 3480 pts/1    Ss     0:00 /bin/bash
 6530 pts/1    S+     0:00  \_ ./uso_fork_exec_wait
 6531 pts/1    S+     0:00      \_ ./uso_fork_exec_wait
 6532 pts/1    R+     0:00      \_ ps f
PADRE: Ha terminado mi hijo 6532
PADRE: Ha terminado mi hijo 6531
```

Paralelamente a la ejecución del programa se puede, **desde otro terminal**, comprobar los procesos en ejecución y sus PIDs:

```
$ ps -t 1 f
  PID TTY      STAT   TIME COMMAND
 3480 pts/1    Ss     0:00 /bin/bash
 6530 pts/1    S+     0:00  \_ ./uso_fork_exec_wait
 6531 pts/1    S+     0:00      \_ ./uso_fork_exec_wait
 6532 pts/1    S+     0:00      \_ ./uso_fork_exec_wait
```

```
$ ps -t 1 f
  PID TTY      STAT   TIME COMMAND
 3480 pts/1    Ss     0:00 /bin/bash
 6530 pts/1    S+     0:00  \_ ./uso_fork_exec_wait
 6531 pts/1    S+     0:00      \_ ./uso_fork_exec_wait
```

```
$ ps -t 1 f
  PID TTY      STAT   TIME COMMAND
 3480 pts/1    Ss+    0:00 /bin/bash
```