

Part II

Procesos

DRAFT

2

Procesos

CONTENTS

2.1	Crear procesos	11
2.1.1	La llamada a sistema <i>fork()</i>	14
2.1.2	función <i>getpid()</i> y <i>getppid()</i>	15
2.1.3	Estado de terminación	17
2.1.3.1	Procesos Huérfanos y procesos Zombis	18
2.1.3.2	Procesos huérfanos	19
2.1.3.3	Procesos Zombis	19
2.1.4	función <i>wait(...)</i> y <i>waitpid(...)</i>	19
2.1.5	Ejemplos:	22
2.1.5.1	Ejemplo del uso genérico de <i>wait</i> para evitar procesos huérfanos.	22
2.1.5.2	Ejemplo del uso de <i>wait</i> para múltiples procesos hijos	23
2.1.5.3	Ejemplo del uso <i>WIFEXITED</i> y <i>WEXITSTATUS</i>	24
2.1.6	Ejercicios propuestos:	25
2.2	Conclusiones	30
2.2.1	Ventajas	30
2.2.2	Desventajas	30

2.1 Crear procesos

Como se ha discutido anteriormente en el desarrollo de las sesiones, la función *fork()* permite crear un nuevo proceso que constituye una copia completa del proceso en ejecución. Los segmentos de datos, de código y de pila (*DS: Data Segment, CS: Code Segment, SS: Stack Segment*) son copiados exactamente en el momento de la ejecución de la función *fork()*.

En Linux, *fork()* se implementa utilizando la técnica COW *copy-on-write*

que pospone la duplicación de los datos en memoria hasta que alguno de los procesos ejecute una operación de escritura. Realizar la duplicación de recurso en el momento de la ejecución de la orden *fork()* es ineficiente debido a que, en principio, muchos recursos pueden ser compartidos entre los procesos padre e hijo. Sin embargo, debido a la naturaleza propia de los sistemas operativos derivados de Unix, la ejecución de la orden *fork()* está acompañada de la orden para cambiar la imagen de memoria del proceso, esto es, alguna de las funciones de la familia *exec..()* (se describirá más adelante).

Así, Copy-on-write (o COW) es una técnica que busca hacer eficiente el proceso de creación del proceso hijo, retrasando o evitando la copia de los datos innecesaria. En lugar de duplicar la imagen de memoria del proceso padre, el padre y el hijo pueden compartir una única imagen en memoria. Sin embargo, los espacios de memoria se marcan como de solo lectura de forma que si se escriben en ellos se realiza un duplicado y cada proceso recibe una copia única. En general, la duplicación de recursos durante la creación del proceso hijo ocurre sólo cuando alguno ejecuta una operación de escritura; antes de eso, comparten los bloques de memoria en modo de solo lectura.

Este enfoque moderno de la implementación de la función *fork()*, heredada de la función *vfork()* de los sistemas operativos BSD, resulta ser una implementación eficiente evitando duplicaciones innecesarias. Esto es, en el caso de que los bloques de memoria no sean escritos, por ejemplo, si se llama a la función *exec()* inmediatamente después de *fork()*, realizar la duplicación resultaría en un costo innecesario. Así, el costo de la ejecución de *fork()* es la duplicación de las tablas de páginas del padre y la creación de un descriptor de proceso único para el hijo.

Un caso general para la creación de procesos hijos implica el reemplazo de la imagen de memoria del proceso hijo por la de un programa diferente al del proceso padre. Esta es la forma para ejecutar procesos externos desde un proceso en ejecución: *fork() + exec()*. La figura 3.1 muestra el diagrama de la ejecución del llamado *fork() + exec()*.

Dentro de las características principales del nuevo proceso, al que llamaremos proceso hijo, se encuentran:

- El proceso hijo tiene su propio identificador de proceso *pid*, y este *pid* no coincide el de ningún proceso existente.
- El identificador del proceso padre *ppid* del proceso hijo es *pid* del proceso que ejecutó *fork()*.
- Su ejecución inicia en la instrucción siguiente a la llamada *fork()* que lo creó.
- Con respecto al padre, la ejecución se considera asíncrona, en el sentido que no es posible asumir un orden relativo entre los procesos padre e hijo.

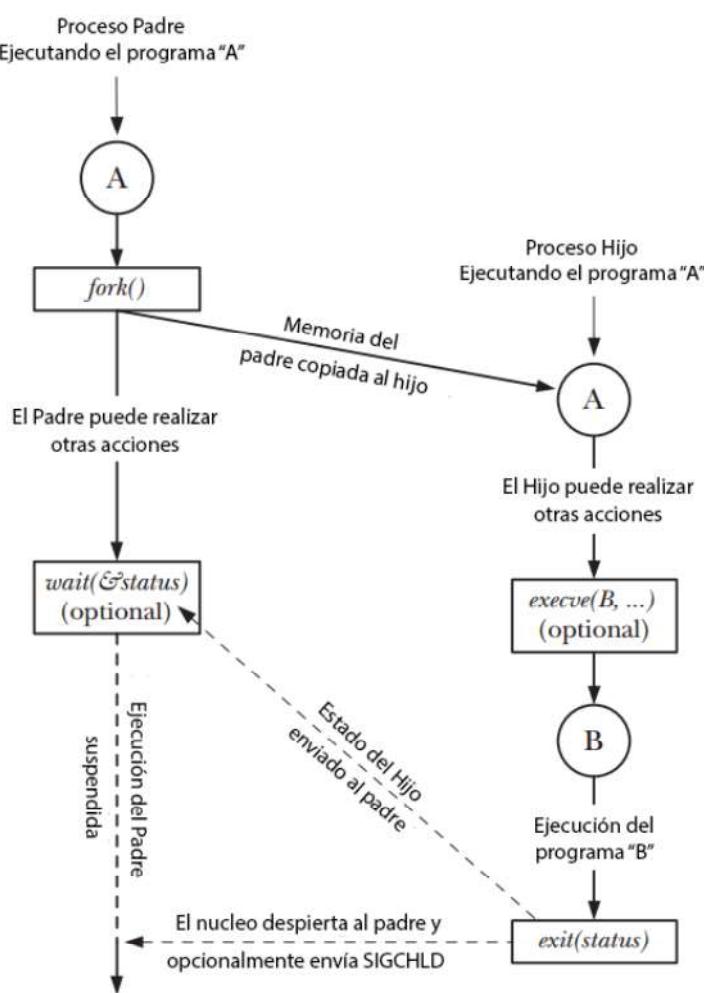


Figure 2.1: Diagrama de la ejecución de las funciones `fork()`, `exec()` y `wait()`.
 [Modificado de: The Linux Programming Interface]

- Los contadores de tiempo de CPU y de recursos utilizados se establecen a cero en proceso hijo.
- Hereda información del padre tales como las variables de entorno, descriptores de archivos abiertos, entre otros.
- El conjunto de señales pendientes inicia vacío.
- El proceso hijo no hereda los temporizadores ni las alarmas que haya programado el proceso padre.
- El proceso hijo no hereda operaciones de E/S pendientes.

2.1.1 La llamada a sistema *fork()*

La función *fork()* está disponible en la librería *unistd.h*. El valor devuelto es de tipo *pid_t* que es en realidad una redefinición del tipo *unsigned int*. Este valor corresponde al identificador de proceso (*pid*) del proceso hijo en el proceso que llama la función. Para el proceso creado el valor devuelto es *0*. En caso de error, se devuelve *-1* en el padre, no se crea ningún proceso y la variable *errno* se carga con el valor adecuado al error. La función puede fallar por diferentes razones, entre otras está la falta de permisos o la capacidad excedida del sistema.

```

1 #include <sys/types.h>
2 #include <unistd.h>
3
4 pid_t fork(void);
```

La principal forma de diferenciar los flujos de ejecución de cada uno de los procesos es mediante la evaluación del valor de retorno de la función. Esto es, si el valor returnedo es >0 el código esta siendo ejecutado por el proceso padre. Si el valor returnedo es *0* el programa está siendo ejecutado por el proceso hijo. En el programa 2.1 se muestra el uso del valor de retorno de *fork()*.

Programa 2.1: Evaluando el valor de retorno para diferenciar la ejecución.

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6
7 int main( void ) {
8     pid_t pidchild = fork();
9     switch (pidchild)
```

```
10  {
11      case -1: /* Error, manejar el error */
12          perror("Error fork");
13          exit(EXIT_FAILURE);
14      case 0: /* Bloque proceso hijo */
15          printf( "Proceso hijo\n" );
16          break;
17      default: /* Bloque proceso padre */
18          printf( "Proceso padre\n" );
19      }
20      return 0;
21 }
```

En el programa 2.1 crea un proceso hijo, tanto el proceso padre como el proceso hijo imprimen una cadena correspondientemente. La ejecución de la función *fork()* de la linea 9 podría almacenar en la variable *pidchild* diferentes valores.

Si al ejecutar la función falla el valor retornado será -1, en este caso no se crea un proceso adicional y debe ser manejada adecuadamente la condición de error. En este caso el manejador es sólo una advertencia mediante la salida de una cadena de texto al flujo *stderr* (generalmente *stderr* está direcccionado por defecto a *stdout*). Así, la cadena final mostrada es complementada con el contenido del valor actual de la variable *errno*. El valor en *errno* es significativo solo cuando el valor de retorno de la llamada de una función indicó un error, es decir: -1 de la mayoría de las llamadas del sistema y -1 o NULL de la mayoría de las funciones de la biblioteca. la variable *errno* está definida en el estándar ISO C como un valor modificable de tipo *int*, y no debe declararse explícitamente.

Si el valor retornado en la variable *pidchild* es diferente de -1, indica que la función fue ejecutada correctamente y un proceso hijo fue creado. En este caso el programa es ejecutado por dos procesos. Si el multiplexor de la linea 9 es ejecutado por el proceso hijo, el programa cambiara el flujo hacia el bloque de sentencias indicado en el caso 0 en la linea 14. Podemos decir entonces que el proceso hijo ejecutará el bloque asociado al caso 0. En este caso, será una sentencia de impresión con la cadena *Proceso hijo* ejecutará el *break* y finalmente la sentencia de la linea 20 indicando una terminación correcta.

Si el bloque es ejecutado por el proceso padre el bloque que se ejecutará será en definido en la entrada *default* del multiplexor *switch*. Lo que hará que el proceso padre imprima la cadena *Proceso padre* y termine ejecutando la sentencia de la linea 20. sin embargo, el valor almacenado en la variable *pidchild* será el correspondiente al *pid* del proceso hijo.

2.1.2 función *getpid()* y *getppid()*

El sistema operativo incluye funciones para obtener la información referente a los identificadores de procesos asignados a los procesos en ejecución. De igual forma es posible obtener la información de identificación del proceso padre de un proceso. En algunos escenarios como en el de comunicación mediante señales, es necesario tener la información de identificación del proceso padre para poder realizar la señalización. Las funciones disponibles para obtener esta información son:

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4
5 pid_t getpid(void);
6 pid_t getppid(void);
```

Las funciones retornan un entero sin signo *pid_t*. En el caso de *getpid()* indica el *pid* del proceso que lo ejecuta. En el caso de *getppid()* (*get parent pid*) devuelve el *pid* del proceso padre del proceso que lo ejecuta. Estas funciones siempre son exitosas y no retornan condición de error.

Ahora escribiremos una versión más completa del programa 2.1 incluyendo las llamadas anteriores.

Programa 2.2: Usando *getpid()* y *getppid()*

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6
7 int main( void ) {
8     pid_t pidchild = fork();
9     switch (pidchild)
10    {
11        case -1:
12            perror("Error fork");
13            exit(EXIT_FAILURE);
14        case 0:
15            printf( "Proceso hijo: pid=%d y
16 ppid=%d\n", getpid(), getppid() );
17            break;
18        default:
19            printf( "Proceso padre: pid=%d y
ppid=%d\n", getpid(), getppid() );
20    }
21 }
```

```
20     return 0;  
21 }
```

Al ejecutar esta versión del programa una posible salida sería:

```
1 $> ./ejemplo1  
2 Proceso padre: pid=5277 y ppid=5272  
3 Proceso hijo: pid=5278 y ppid=5277
```

Cada vez que usted ejecute este programa, los valores numéricos de la salida van a variar, esto porque cada vez que se ejecuta este debe ser un nuevo proceso y los procesos hijos creados en cada ejecución también serán diferentes. Note que el *ppid* del proceso hijo debe coincidir con el *pid* del proceso padre. El *ppid* del proceso padre hace referencia al proceso que lo creó. En general, hace referencia a la ventana del *shell* donde se introdujo la orden de la linea [1](#).

Un comportamiento posible del programa [2.2](#) podría ser una salida como:

```
1 $> ./a.out  
2 Proceso padre: pid=5277 y ppid=5272  
3 Proceso hijo: pid=5278 y ppid=1
```

Aunque esta salida es poco probable por la sencillez del programa utilizado como ejemplo, es una salida posible bajo algunas circunstancias. Note que salida de la linea [3](#) ahora es un poco diferente, en esta ocasión el valor de retorno asociado a la ejecución de *getppid()* es 1. Lo anterior sucede debido a que cuando el proceso hijo ejecuta *getppid()* el proceso padre ha ejecutado la ultima linea de código (linea [20](#)), esto es debido a que el proceso padre termina primero que el proceso hijo. En este caso el proceso hijo es marcado por el sistema operativo como un *Proceso Huérfano* debido a que durante un periodo de tiempo, desde que el proceso padre terminó hasta que él termina, no tiene un proceso padre.

En principio, el sistema operativo espera que el proceso que crea otro proceso mediante *fork()* sea el responsable de pedir el *estado de terminación* del proceso hijo. Debido a que es necesario que algún proceso solicite el *estado de terminación* del proceso hijo, el sistema operativo permite que el proceso *init* adopte al proceso hijo. El proceso *init* es el proceso inicial ejecutado por el kernel del sistema operativo durante la etapa de *booteo* y es considerado el padre de todos los procesos. Por ser el primer proceso que se crea este tiene asignado el valor de *pid* 1.

El *estado de terminación* del proceso es un valor numérico que indica como fue la ejecución de un proceso. En el programa [2.2](#), las lineas [13](#) y la linea [20](#) son las encargadas de devolver el valor que informa el estado de terminación. La primera indica un valor de error mediante el retorno de la constante *EXIT_FAILURE* (definida en *stdlib.h*) y la segunda indica una terminación exitosa.

2.1.3 Estado de terminación

Cuando un programa termina su ejecución puede informar la causa de terminación devolviendo al proceso padre una cantidad limitada de información. Esta información es limitada debido que sólo puede informar la forma en la que terminó la ejecución y no es posible devolver la causa o las causas de la terminación en caso de que esta haya sido por una condición de error.

Específicamente, este información corresponde a un valor numérico entre 0 y 255. Existe algunas convenciones relacionadas con los valores de retorno, por ejemplo se considera que un programa que retorna 0 indica una condición de éxito en su ejecución y cualquier otro valor indica una condición de error, como por ejemplo -1. Es posible que la explicación de esta convención esté relacionada con la emulación de la lógica booleana de las primeras implementación del lenguaje C, en el cual un valor de 0 indica un valor de verdad *false* y cualquier otro valor, positivos o negativos, indica un valor de verdad *true*.

Sin embargo, de acuerdo con el documento del estándar C99 ISO/IEC 9899:C2007 los valores de retorno, de por ejemplo, la función *exit(int status)* deben corresponder a *formas definidas en las implementaciones*. Esto es, si un programa retorna 0, mediante por ejemplo una sentencia *return 0;*, el valor returnedo no es necesariamente un 0 sino un valor que indica una correcta terminación y que depende de la implementación particular que se este usando. Por portabilidad, es recomendable utilizar las macros *EXIT_SUCCESS* y *EXIT_FAILURE* definidas en *stdlib.h*, para indicar una ejecución exitosa o una terminación por alguna falla, respectivamente.

- **EXIT_SUCCESS:** Esta macro se puede usar con la función de salida para indicar la finalización exitosa del programa. En los sistemas POSIX, el valor de esta macro es 0.
- **EXIT_FAILURE:** Esta macro se puede usar con la función de salida para indicar la finalización incorrecta del programa en un sentido general. En los sistemas POSIX, el valor de esta macro es 1. En otros sistemas. Otros valores de estado distintos de cero también indican fallas.

Hasta este punto hemos sugerido que el estado de terminación del programa es necesariamente el estado de terminación del proceso. Pero es posible que un proceso termine por razones diferentes a la terminación del programa, como por ejemplo, el envío de una señal de terminación externa (*SIGINT*, *SIGTERM*, *SIGKILL*). Por lo tanto, el estado de terminación de un programa, o el valor que retorna, es diferente, conceptualmente, al estado de terminación del proceso. Sin embargo, cuando el proceso termina porque el programa que ejecutaba terminó o retornó, entonces el estado de terminación del proceso coincide en parte con el estado de finalización del proceso.

2.1.3.1 Procesos Huérfanos y procesos Zombis

Hemos visto que para el sistema operativo es necesario que los procesos que crean procesos adicionales soliciten el *estado de terminación del proceso*, esto es debido a el mecanismo utilizado para administrar los recursos de la maquina asignados a cada proceso. Al terminar la ejecución de un proceso sus recursos asociados a los bloques de memoria son marcados como disponibles. Sin embargo, no todos los recursos utilizados son inmediatamente liberados. Específicamente, el *estado de terminación* del proceso es mantenido por el sistema operativo dentro de la entrada específica asignada para él en el BCP. Así, un proceso que haya finalizado la ejecución de su programa pero que no le hayan solicitado el estado de terminación permanecerá ocupando dicha entrada.

Cada proceso padre es el encargado de realizar dicha solicitud. Sin embargo, es posible que se den dos condiciones particulares. Si un proceso padre termina antes que el proceso hijo, esté ultimo al terminar será marcado como un proceso *huérfano* debido a que no tiene padre. La otra condición se da cuando un proceso hijo ha finalizado pero el padre no ha solicitado el estado de terminación, en este caso el proceso es *inactivo* y es marcado como un proceso *zombie*.

2.1.3.2 Procesos huérfanos

Cuando un proceso padre muere antes que un proceso hijo, el núcleo sabe que no recibirá una solicitud del estado de terminación del proceso, por lo que hace que estos procesos sean huérfanos y los pone bajo el cuidado de *init*. Este proceso realizará la llamada al sistema solicitando el estado de terminación y posibilitando la liberación de los recursos asignados.

2.1.3.3 Procesos Zombis

Un proceso zombi es un proceso que ha finalizado su ejecución y está esperando que su proceso padre lea su estado de terminación. Debido a que el proceso hijo ha finalizado, técnicamente es un proceso *inactivo*, sin embargo, dado que está esperando a su padre, todavía hay una entrada en la tabla de procesos, en el BCP (*Bloque de Control de Procesos*). Los recursos que el proceso hijo usó se liberan para otros procesos, específicamente los espacios de memoria. Puede ser malo tener demasiados procesos zombies, ya que ocupan espacio en la tabla de procesos, si se llena evitará que se ejecuten otros procesos.

2.1.4 función *wait(...)* y *waitpid(...)*

El mecanismo disponible dentro de los sistemas operativos derivados de *Unix* es un conjunto de llamadas al sistema de nominadas *wait(...)* y *waitpid(...)*. Los prototipos de las funciones son:

```

1
2 #include <sys/types.h>
3 #include <sys/wait.h>
4
5 pid_t wait(int *estado);
6 pid_t waitpid(pid_t hijo, int *estado, int opciones
    );
7
8 WIFEXITED(int estado)
9 WEXITSTATUS(int estado)
10 WIFSIGNALED(int estado)
11 WIFSTOPPED(int estado)
12 WSTOPSIG(int estado)

```

La función *wait* es una llamada bloqueante y espera hasta que un proceso hijo termine o se detenga. En el caso de que ningún proceso hijo haya terminado su ejecución. Si al momento de la llamada algún proceso ha terminado, estas funciones seleccionan uno de estos procesos y retorna el estado de terminación asociado al proceso. El valor de retorno es almacenado en la dirección de memoria apuntada por el parámetro *int *estado*.

La diferencia fundamental entre las dos funciones es que la función *wait* espera por cualquier proceso que haya finalizado mientras que la función *waitpid* espera por la terminación de un proceso específico definido en el parámetro *pid_t hijo*.

Si el proceso que llama *wait* no ha creado procesos hijos la llamada no bloqueará el proceso y retornará una condición de error. Si el proceso ha creado hijos y alguno de éstos ya han terminado, el proceso llamante no se bloqueará y el estado de terminación de alguno de los procesos hijos es devuelto.

Para ambas funciones si el parámetro *estado* es *NULL* no se almacenará el estado de terminación pero éste será solicitado al sistema operativo, lo que generará la liberación de los recursos asociados al proceso.

La función *waitpid* es más compleja en su funcionamiento debido a las variantes posibles en los parámetros *hijo* y *opciones*. Las variantes son:

Para el parámetro *pid_t hijo*:

- -1: *waitpid* actúa igual que *wait*, esperando cualquier hijo.
- mayor que 0: indica el *pid* de un proceso hijo determinado que se desea esperar.

- 0: para cualquier hijo con el mismo grupo de procesos que el padre.
- menor que -1: para cualquier hijo cuyo identificador de grupo *pgid* sea igual al valor absoluto de *pid*.

Para el parámetro *opciones* es en realidad una máscara de *bits* que pueden ser construidas con el operador *bitwise OR* (carácter |) o mediante el uso de macros como:

- WNOHANG: Indica que el proceso que llama no debe ser bloqueado si ninguno de los hijos ha terminado.
- WUNTRACED: Indica que el padre desea obtener más información si el proceso hijo sido detenido *SIGTIN*

El valor de retorno de las dos funciones en caso de error es -1 y la variable *errno* se establece en uno de los siguientes valores:

- ECHILD: Cuando no hay procesos hijos para esperar. Para *waitpid* esto también es retornado cuando el proceso hijo especificado en el parámetro *hijo* no se encuentra o cuando el *id* del grupo no existe.
- EINTR: La llamada fue interrumpida por una señal recibida al proceso llamante o porque el área de memoria apuntada por *estado* no puede ser escrita.
- EINVAL: Indicando que un valor invalido fue especificado en el argumento *opciones* de la función *waitpid*.

El valor de retorno almacenado en *estado* corresponde a la información de terminación del proceso que no necesariamente es el código de terminación del programa, como lo mencionamos anteriormente (sección 2.1.3), éste es una parte de estado de terminación del proceso. Para analizar este valor existen un conjunto de macros que permiten interpretar la información que el sistema operativo ha incluido en el *estado de terminación*.

- WIFEXITED (estado) devuelve un valor distinto de cero, verdadero, si el hijo terminó como resultado de la ejecución del programa asociado.
- WEXITSTATUS (estado) devuelve los ocho bits menos significativos de *estado*, que corresponden al estado de terminación del programa, es decir el valor de la función *exit* o de la sentencia *return 0;*. Esta macro sólo puede ser evaluado si WIFEXITED devuelve un valor distinto de cero.
- WIFSIGNALED (estado) devuelve verdadero si el proceso hijo salió debido a una señal que no fue capturada.

- WTERMSIG (estado) devuelve el número de la señal que causó el proceso hijo para terminar. Esta macro solo puede ser evaluado si WIFSIGNALED devuelve un valor distinto de cero.
- WIFSTOPPED (estado) devuelve verdadero si el proceso secundario que causó el retorno está actualmente detenido; esto sólo es posible si la llamada a *waitpid* se realizó con el argumento WUNTRACED.
- WSTOPSIG (estado) devuelve el número de la señal que causó que el proceso hijo se detuviera. Esta macro solo se puede evaluar si WIFSTOPPED devuelve un valor distinto de cero.

2.1.5 Ejemplos:

Ejemplos de las funciones definidas anteriormente.

2.1.5.1 Ejemplo del uso genérico de *wait* para evitar procesos huérfanos.

Una versión correcta para crear un proceso hijo garantizando que no se genere la condición de proceso *huérfano* se describe en el programa 2.3.

Programa 2.3: Ejemplo del uso de *fork()*.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 int main( void ) {
8     pid_t pidchild = fork();
9     switch (pidchild)
10    {
11        case -1:
12            perror("Error fork");
13            exit(EXIT_FAILURE);
14        case 0:
15            printf( "P. hijo: pid=%d y ppid=%d\n",
16                    getpid(), getppid() );
17            break;
18        default:
19            printf( "P. padre: pid=%d y ppid=%d\n",
20                    getpid(), getppid() );
21            wait(NULL);
22    }
23 }
```

```

20     }
21     return EXIT_SUCCESS;
22 }
```

Note que se ha añadido el uso de la función *wait* en la linea 19. Esta instrucción se encuentra en el bloque del proceso padre, por lo que a pesar de la naturaleza asíncrona de la ejecución de los dos procesos el proceso padre no podrá terminar antes que el proceso hijo, evitando que éste último entre en la condición de huérfano. en este caso en particular, el código de terminación del proceso hijo no es almacenado dentro de los segmentos de memoria del proceso padre. Este efecto es debido al parámetro *NULL* de la función *wait*. Sin embargo, el comportamiento bloqueante de ésta se mantienen.

2.1.5.2 Ejemplo del uso de *wait* para múltiples procesos hijos

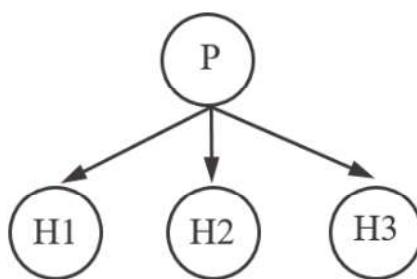


Figure 2.2: Diagrama de la jerarquía de procesos con múltiples hijos.

Programa 2.4: Usando *fork() + wait()* con múltiples hijos.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7
8 int main( void ) {
9     pid_t root = getpid();
10
11     for( int i=0; i<3; i++ ){
12         if( !fork() )
13             break;
14     }
```

```

15 if( root == getpid() ){
16     for(int j=0; j<3; j++)
17         wait(NULL);
18     printf("Finalizando padre %d\n", getpid());
19 }
20 else
21     printf("Finalizando hijo %d\n", getpid());
22
23 return EXIT_SUCCESS;
24 }
```

En general la idea es que el proceso padre deberá llamar a la función *wait* tantas veces como haya llamado a *fork()*. Para garantizar que el ciclo de llamadas a *wait* sólo sea ejecutado por el proceso padre se incluye la sentencia de la linea 15. Note que el contenido de la variable *root* se cargo inicialmente con el valor del proceso inicial (padre) en la linea 9. Así, la comparación sólo será verdadera para este proceso. Los procesos hijos que ejecuten la linea 15 contendrán en la variable *root* el *pid* del padre pero al ejecutar la llamada *getpid()* esta devolverá su *pid*, el cual no coincidirá con *root*.

Al ejecutar varias veces el programa notará que el proceso padre siempre imprimirá después de los procesos hijos. Ese, en particular, es el comportamiento deseado. Garantizar que el padre nunca termine antes que los procesos que creó permite una terminación adecuada de la ejecución de los procesos.

La ejecución del programa anterior genera una salida similar a:

```

1 $>a.out
2 Finalizando hijo 7648
3 Finalizando hijo 7649
4 Finalizando hijo 7650
5 Finalizando padre 7647
```

“nobreak

2.1.5.3 Ejemplo del uso *WIFEXITED* y *WEXITSTATUS*.

El programa 2.5 es un ejemplo de la forma de obtener el código de terminación de un proceso hijo.

Programa 2.5: Obtener el valor entero returnedo por un proceso hijo.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
```

```
7 int main( void ) {
8     int estado;
9     pid_t pidchild, pidwait;
10    pidchild = fork();
11    switch (pidchild){
12        case -1:
13            perror("Error fork"); exit(EXIT_FAILURE);
14        case 0:
15            exit(15);
16        default:
17            pidwait = wait(&estado);
18            if(pidwait != -1)
19                if(WIFEXITED(estado))
20                    printf("Hijo estado: %d\n",
21                         WEXITSTATUS(estado));
22    }
23    return EXIT_SUCCESS;
24 }
```

Para obtener el estado de terminación del programa a partir del estado del terminación del proceso es necesario solicitarlo a través de la instrucción de la linea 17 la cual almacena dicho valor en la variable *estado*, La linea 18 verifica el valor retornado por *wait* controlando que la ejecución no haya devuelto una condición de error ($\neq -1$). El siguiente paso es verificar que el proceso haya retornado por una condición de salida del programa. La linea 19 retorna verdadero si eso es así. Una vez que nos hemos asegurado que el valor de *estado* corresponde a un estado de terminación de un proceso hijo y que este termine por ejecución del programa se obtiene dicho valor con la macro *WEXITSTATUS* que devuelve los 8 bits menos significativos en los cuales esta almacenado el valor retornado en la linea 15.

La ejecución del programa genera una salida como la siguiente:

```
1 $> ./a.out
2 Hijo estado: 15
```

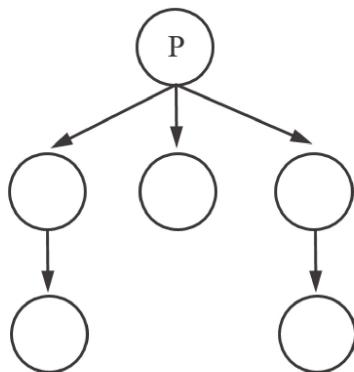
2.1.6 Ejercicios propuestos:

Figure 2.3: Ejercicio propuesto 1.

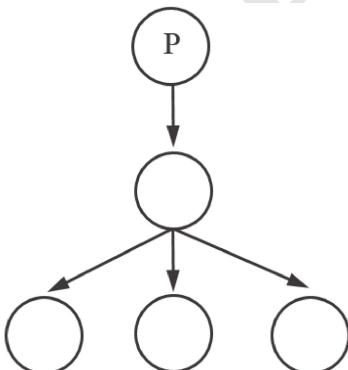


Figure 2.4: Ejercicio propuesto 2.

Las formas para verificar la jerarquía de los procesos creados varían desde diversos comandos en la terminal como *pstree*, *strace*, entre otros. Sin embargo, para poder visualizarlos a través de la linea de comando todos los proceso deben demorar lo suficiente como para poder introducir el comando y visualizar el árbol. Así, que haremos uso del comando *pstree* pero usado desde el código mediante la llamada al sistema *system*. No utilice esta función en el desarrollo de los programas futuros, en esta sección es solo para ilustrar el mecanismo.

Programa 2.6: Función *showtree* para visualizar el árbol de procesos.