Ejercicio 4:

a) Tras el estudio del árbol de procesos tras la ejecución del programa y un análisis teórico, vemos claramente que todos los procesos salvo el padre quedarán o huérfanos o zombies, ya que ningún padre espera por la finalización de sus hijos.
b) Hay cuatro procesos que pueden quedar zombies durante unos instantes, pero el padre de cada uno de ellos los recogerá exitosamente gracias al wait. Sin embargo, el resto de los procesos salvo el primero quedarán huérfanos o zombies según la ejecución del programa.
   En ambos programas se generarán 16 procesos en total.

Ejercicio 5:

a) Al poner el break en el else del bucle, hacemos que cuando un proceso cree un hijo se salga del bucle, haciendo así que cada padre tenga como mucho un hijo. El cambio es de apenas dos líneas.
b) En este caso, hemos optado por que el padre vaya generando los hijos y los espere, es decir, los hijos no serán concurrentes, sino que se creará uno y el padre le esperará, luego se creará otro y el padre le esperará. Esto lo hemos hecho con el objetivo de minimizar los cambios, ya que no se nos pedía que los hijos fuesen concurrentes en el enunciado. En el caso de que quisiéramos hacer que los hijos fuesen concurrentes, tendríamos que sacar el wait del bucle for y, tras dicho bucle, crear un bucle while de la forma while(wait(NULL)! = -1). De esta forma, se esperará a que terminen todos los hijos del proceso principal, ya que wait solo devuelve -1 cuando el proceso no tiene hijos en ejecución.

Ejercicio 6:

Hemos optado por crear la estructura de forma implícita y luego escribir según la memoria de esta con fwrite para acortar el código lo más posible. Además, tras un análisis con sucesivas ejecuciones de valgrind, llegamos a la conclusión de que hay que liberar la memoria en todos los hijos si ha sido previamente creada por el padre, ya que se crea una copia de la memoria.

Ejercicio 8:

Como du se encuentra en un directorio distinto al resto, y ya que para las funciones execl y execv tenemos que dar el path de la función hemos optado por hacer que bash ejecute el programa y así evitamos tener que buscar el path de cada programa y que lo busque bash por nosotros, de la forma bash -c nombredelprograma. A nivel de resultados es el mismo, pero conseguimos un programa más general que si ejecutáramos una sentencia de ifs concreta para el path de cada programa.

Ejercicio 9:

Generamos una pareja de pipes por cada hijo, una para que el padre le pase información al hijo y otra para que el hijo le pase la información al padre. Tras ello, en caso de que sea uno de los hijos, definimos un switch en el que según que hijo sea, se ejecute una de las cuatro funciones requeridas, leyendo primero los operandos desde la pipe del padre y enviando la información al padre por la otra. Nos aseguramos de que el padre reciba dicha información antes de leerla e imprimirla por pantalla con la línea while(wait(NULL) != -1), que hará que el padre espere por la terminación de sus cuatro hijos.

De los tres ejemplos tenemos:

```
./build/ejercicio9 -2 -2

Datos enviados a través de la tubería por el proceso PID =
14185. Operando 1: -2. Operando 2: -2. Potencia: 0

Datos enviados a través de la tubería por el proceso PID =
14186. Operando 1: -2. Operando 2: -2. Factorial: 1

Datos enviados a través de la tubería por el proceso PID =
14187. Operando 1: -2. Operando 2: -2. Combinaciones: 1

Datos enviados a través de la tubería por el proceso PID =
14188. Operando 1: -2. Operando 2: -2. Suma de absolutos: 4
```

Donde vemos que la potencia es 0 con exponente negativo, el segundo ejemplo:

```
./build/ejercicio9 1 3

Datos enviados a través de la tubería por el proceso PID =
14454. Operando 1: 1. Operando 2: 3. Potencia: 1

Datos enviados a través de la tubería por el proceso PID =
14455. Operando 1: 1. Operando 2: 3. Factorial: 1

Datos enviados a través de la tubería por el proceso PID =
14456. Operando 1: 1. Operando 2: 3. Combinaciones: 0

Datos enviados a través de la tubería por el proceso PID =
14457. Operando 1: 1. Operando 2: 3. Suma de absolutos: 4
```

Aqui vemos que el numero de combinaciones es 0 cuando el segundo operando es mayor que el primero, y finalmente:

```
./build/ejercicio9 4 0

Datos enviados a través de la tubería por el proceso PID =
14509. Operando 1: 4. Operando 2: 0. Potencia: 1

Datos enviados a través de la tubería por el proceso PID =
14510. Operando 1: 4. Operando 2: 0. Factorial: 0

Datos enviados a través de la tubería por el proceso PID =
14511. Operando 1: 4. Operando 2: 0. Combinaciones: 1

Datos enviados a través de la tubería por el proceso PID =
14512. Operando 1: 4. Operando 2: 0. Suma de absolutos: 4
```

En este caso, el segundo operando es 0, así que decidimos que el factorial de la división sea 0 por convención, para no tener que dividir por 0.

Ejercicio 12:

En este caso hemos creado una función general calcular_primos que nos servirá para calcularlos tanto para el programa de procesos como función para el programa de hilos. Usamos la función clock_gettime en vez de clock() ya que en el programa de hilos estos avanzan el reloj de forma conjunta lo que falsea resultados. Además, nos hemos dado cuenta de que en uno de nuestros ordenadores personales el tiempo de ejecución del programa de hilos y de procesos es prácticamente el mismo. Sin embargo, en otro de nuestros PC's sí que existe una diferencia notoria entre los dos programas (siendo el de hilos claramente inferior) cuando ejecutamos el mismo código fuente. Si hablamos de tiempos, en el procesador que ejecutaba ambos programas a un tiempo similar obteníamos un tiempo de alrededor de 0.56 s, mientras que en el otro obteníamos 1.1 s para el de procesos y 0.76s para el de hilos.

Ejercicio 13:

En estos programas hemos creado funciones específicas para intentar repetir la menor cantidad de código posible. Además, hemos creado una struct con diferentes campos para el manejo de los hilos con campos como la id de los hilos a la hora de imprimir la salida esperada. En el ejercicio b específicamente pensamos que una forma de solucionar la impresión sería pasar en la estructura que contiene la información del hilo, un puntero a memoria estática donde actualizamos la fila de la matriz en la que se encuentra cada hilo, de forma que podemos acceder a ella desde el hilo contrario para su impresión.

Sin embargo, al ser más rápida la multiplicación que el manejo de strings y la impresión en pantalla puede darse el caso en que el hilo 2 imprima que el hilo 1 no ha empezado por que cuando obtuvo esa información el hilo 1 no había empezado la fila 1 aunque ya la haya terminado.