Lab shell → intérprete o línea de comandos.

La principal tarea de una shell es ejecutar comandos. Provee un montón de facilidades para ejecutar programas, combinarlos. Es un programa cuya función es ejecutar otros programas. En sistemas donde no hay una interfaz gráfica, la única manera de interactuar con el sistema es mediante la shell.

Una de las shells más conocidas es **Bash**, en el entorno de Unix.

Diferencia entre la terminal y la shell → La shell es un programa que ejecuta otros programas. Una terminal es la combinación entre monitor y teclado.

En el lab shell debemos implementar nuestra propia shell, con funcionalidad limitada. Va a ser una shell creada desde cero que corre en Unix. Esta shell tiene lo básico para parsear entrada y salida.

La implementación de la shell se asemeja a la de xargs con la diferencia que entre fork y exec va a hacer lo que el usuario ingrese. Una vez que no haya más que leer o el usuario llama a exit, la shell cierra su estado.

Orden de ejecución
sh.c

readline.c

runcmd.c → recibe la cadena de la entrada estándar, trata de interpretar algunas de las palabras claves reservadas estándar (una de esas es cd), si no es ninguna de esas, entonces parsea (se mete en el comando y busca que binario ejecutar y que redireccion tiene que hacer) y pasa a hacer la ejecución (fork, exec). Se encarga de read, parse y exec.

parsing.c → interpreta los comandos y devuelve un struc_cmd (abstracción que tiene el esqueleto y podemos encontrar en types.h) que representa el comando a correr. El struct_cmd tiene varias representaciones de lo que es un command. Tiene un struct genérico que guarda un tipo (característica del comando que voy a ejecutar).

echo hi | *sleep 5* → es un comando para la shell que resulta ser un comando compuesto por dos procesos y un pipe. Es la simbología que tiene la shell para representar un pipe.

Observación → El tipo EXEC va a ejecutar un comando, REDIC si necesita redirecciones, BACK cuando necesita un segundo plano, PIPE requiere usa pipes. Cada comando va a tener un tipo definido. A su vez, cada tipo de comando va a estar representado por un tipo de dato específico.

El hecho de que los distintos tipos de comandos (EXEC, REDIR, BACK, PIPE) tengan los mismos campos que al estructura genérica struct_cmd permite que cada struct particular (excec_cmd, back_cmd, pipe_cmd) se pueda castear como cmd.Por lo que el parseo de la línea me devuelve un struct cmd.

exec.c → hace una cosa u otra dependiendo de qué comando sea.

Al implementar pipe vamos a tener que hacer fork para ejecutar los procesos en procesos hijos pero el fork inicial que realiza la terminal ya está implementado en el esqueleto.

echo hi → imprime hi en la terminal (salida estándar)

echo hi > *prueba* → escribe hi en el archivo prueba y en caso de que no existe lo crea. Es decir, redirige lo que va a stdoutput a un archivo. Puedo comprobarlo con cat prueba.
Observación (1) :De esta manera puedo redirigir la entrada estándar o la salida del error.
Observación (2) :El proceso echo no se entera que se escribe en un archivo. El proceso siempre escribe sobre el file descriptor 1 (salida estándar) pues está compilado para escribir sobre el fd[1]. La terminal es quien hace que ese fd[1] en lugar de apuntar al monitor, apunte a un archivo.

wc -l → cuenta la cantidad de líneas que se ingresan en la pantalla. Una vez de ingresar el comando en la terminal puedo escribir las líneas y corta con control D indicando un end of file.

wc -l <prueba → redirigo la entrada estándar al archivo prueba. Devuelve la cantidad de líneas que tiene el archivo prueba.

ls /no-existe → se muestra un mensaje de error en la terminal.

Mensaje de error →Si quiero escribir el mensaje de error en un archivo, no lo podemos hacer mediante ls /no-existe >prueba porque el pico > se usa para redirigir salida estándar (fd[1]) y ese mensaje lo imprime strderror (fd[2]). Por lo tanto, lo tengo que hacer de la siguiente manera: ls /no-existe 2>prueba. Esto le indica a la shell que corra el ls pero que rediriga la salida de strderror en el archivo prueba.
De esta manera podemos redirigir los 3 fds: stdin, stdout y stderror.

¿Cómo lo podemos implementar? Cuando hacemos fork, el proceso hijo puede hacer lo que quiera con sus files descriptors y no va a modificar al padre. Pero tiene una ventana de tiempo antes de llamar a exec para poder tunear cosas.

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]){
     if (argc<2){
          printf("se necesita un argumento");
           exit(-1);
     }
     close(1);

     int fd = open("hola.txt",O_CREAT | O_RDWR, 0644);
     printf("Archivo abierto en %d\n,fd);
```

```
    execvp(argv[1],argv+1);
    printf("Terminando: %d\n", getpid());
```

Cierra el fd[1], crea un archivo (por defecto nos devuelve el más chico) y llama a execvp que imprime lo que hay en el archivo pensando que el mismo es salida estándar. ***Este es el concepto de la implementación de la redirección.***

Entonces, tenemos que cerrar los files descriptors que no vamos a utilizar dependiendo lo que nos haya ingresado el usuario y abrir las direcciones de los archivos, luego pisar los files descriptors. Hay que tener cuidado en como abrir los archivos.

Observaciones:

- ¿Qué pasa si quiero redirigir un archivo que no existe?
    - Bash me lo indica
- ¿Qué pasa si escribo un archivo que no existe?
    - Bash me lo crea

Esas dos sutilezas en principio con open(2) y close(2) se pueden resolver.

Prueba 0:

1. Close(1)
2. Open(“file-out.txt”)

O también:

1. Close(0)
2. open(“file-in.txt”)

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]){
    close(1);
    int fd = open("file-out", O_CREAT | O_RDWR, 0644);

    printf("archivo abierto en %d\n", fd);
    printf("terminando: %d\n",getpid());
}
```

Una pequeña mejora que podemos introducir con dup(2) que nos permite duplicar esto. Tiene la misma semántica de open pero la diferencia es que copia un file descriptor que ya estaba abierto. Me va a desenvolver el primer file descriptor que encuentre desocupado.

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
```

```
int main(int argc, char * argv[]){
      int fd = open("file-out", O_RDONLY);
      printf("archivo abierto en %d\n", fd);

      size_t n=0;
      char * line= null;
      getline(&line, &n,stdin);
      printf("lei: %d\n",line);

      printf("terminando: %d\n",getpid());
      free(line);
}
```

Open(2) y dup(2) funcionan igual. La diferencia es que con open podes tener un archivo abierto desde antes y después copiarlo con dup.

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]){
      int fd = open("file-out",O_CREAT | O_RDWR,0644);
      close(1);
      int dupfd = dup(fd);
      printf("archivo abierto en %d, dup devuelve %d\n",
fd,dupfd);
      printf("terminando: %d\n",getpid());
}
```

fd=3 y dupfd=1. Por lo que dupfd es la salida estándar.
Si corremos valgrind obtenemos 4 archivos abiertos. Por lo que lo correcto es cerrar fd.

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]){
      int fd = open("file-out",O_CREAT | O_RDWR,0644);
      close(1);
      int dupfd = dup(fd);
      close(fd);
      printf("archivo abierto en %d, dup devuelve %d\n",
fd,dupfd);
      printf("terminando: %d\n",getpid());
}
```

Los files descriptors 0, 1 y 2 los cierra por defecto la librería de C al salir. El 3 puede ser un problema sobre todo si hacemos pipes anidados. Este si lo tenemos que cerrar.

Todo proceso al ser creado espera 3 files descriptors abiertos (0, 1 y 2). Por lo que la idea es que todos los programas tengan únicamente 3 files descriptors abiertos.

Prueba 1:

1. fd = open("file-out.txt")
2. close(1)
3. dup (fd) //debería devolver 1
4. close(fd)

O también:

1. fd = open ("file-in.txt")
2. close(0)
3. dup(fd) //debería devolver 0

Sin embargo, si hago un open y después un dup siempre tengo que hacer un open - close. Por lo que una mejor opción es usar dup2().

```
int dup2(int oldfd, int newfd)        //Le paso 2 files descriptors. Cierra 1
                                      //y te lo duplica en esa posición.
```

Un flag de open() es O_CLOEXEC → el sistema operativo se va acordar que abriste un file descriptor con este flag y en el momento que hagas un exec te lo va a cerrar por vos.

<u>Comparación entre dup y dup2</u>

Código con dup

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]){
      int fd = open("file-out",O_CREAT | O_RDWR,0644);
      close(1);
      int dupfd = dup(fd);
      close(fd);
      printf("archivo abierto en %d, dup devuelve %d\n",
fd,dupfd);
      printf("terminando: %d\n",getpid());
}
```

Código con dup2

```
#include <fcntl.h>
#include <stdio.h>
```

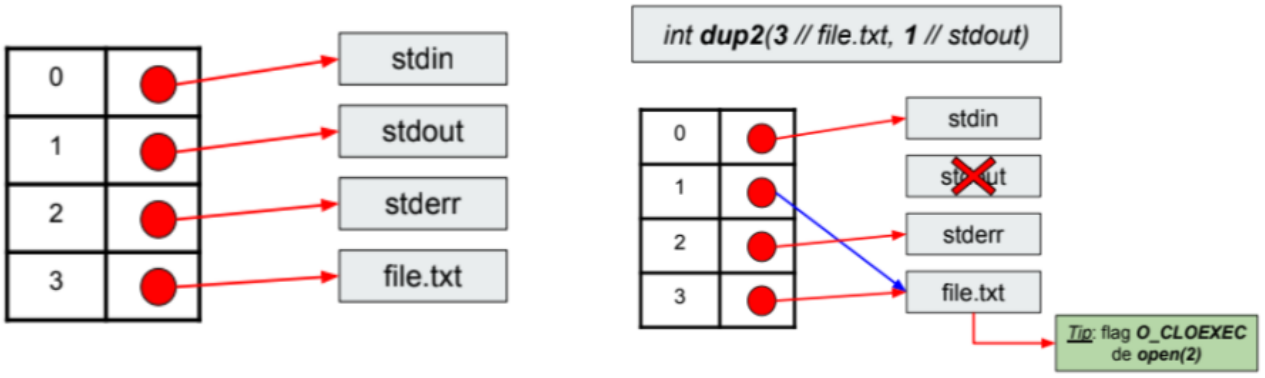
```
#include <unistd.h>

int main(int argc, char * argv[]){
      int fd = open("file-out",O_TRUNC | O_CREAT | O_RDWR,0644);
      int dupfd = dup2(fd,1);
      close(fd);
      printf("archivo abierto en %d, dup2 devuelve %d\n",
fd,dupfd);
      printf("terminando: %d\n",getpid());
}
```

Dup2() me cierra el destino no el viejo. En este caso, agarra a fd y lo pisa en la posición 1.

La ventaja de dup2() es que poder indicar a qué file descriptor va a parar, a diferencia de lo que ocurre con open() que esto no es posible,sino que agarra el file descriptor cerrado más chico lo cual resulta anti pattern aunque es posible.

De todas formas, si bien usar dup2() es más prolijo, no tiene ventaja sobre open() + close().

Dup2: como se comporta gráficamente



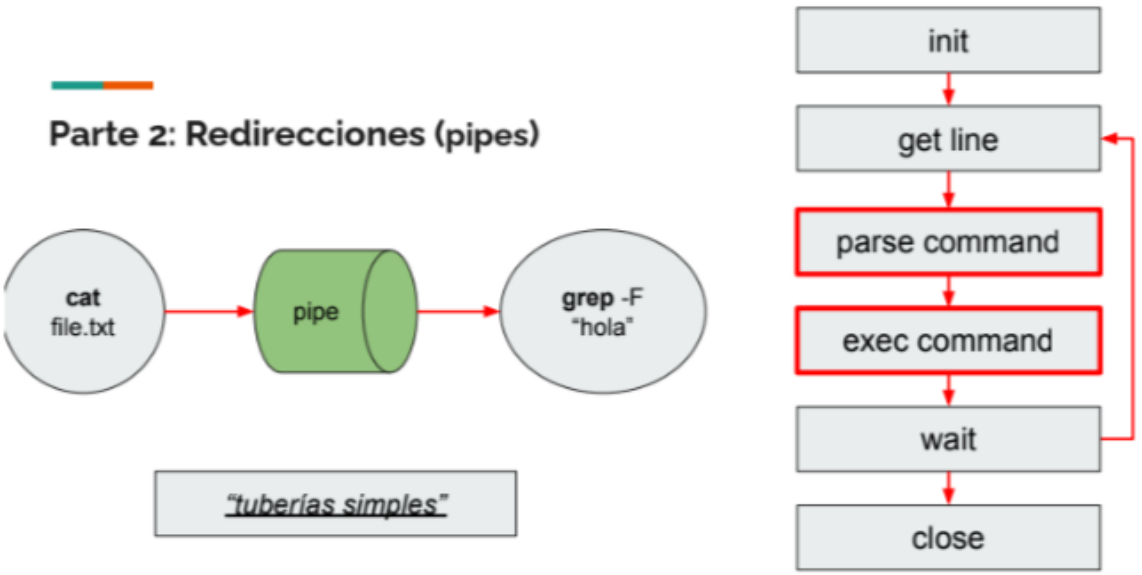
Pipes → es una forma de redirigir salida estándar de un problema a entrada estándar.

Tuberías simples

Convertir proceso hijo en exec en 2 concatenados por 1 pipe.

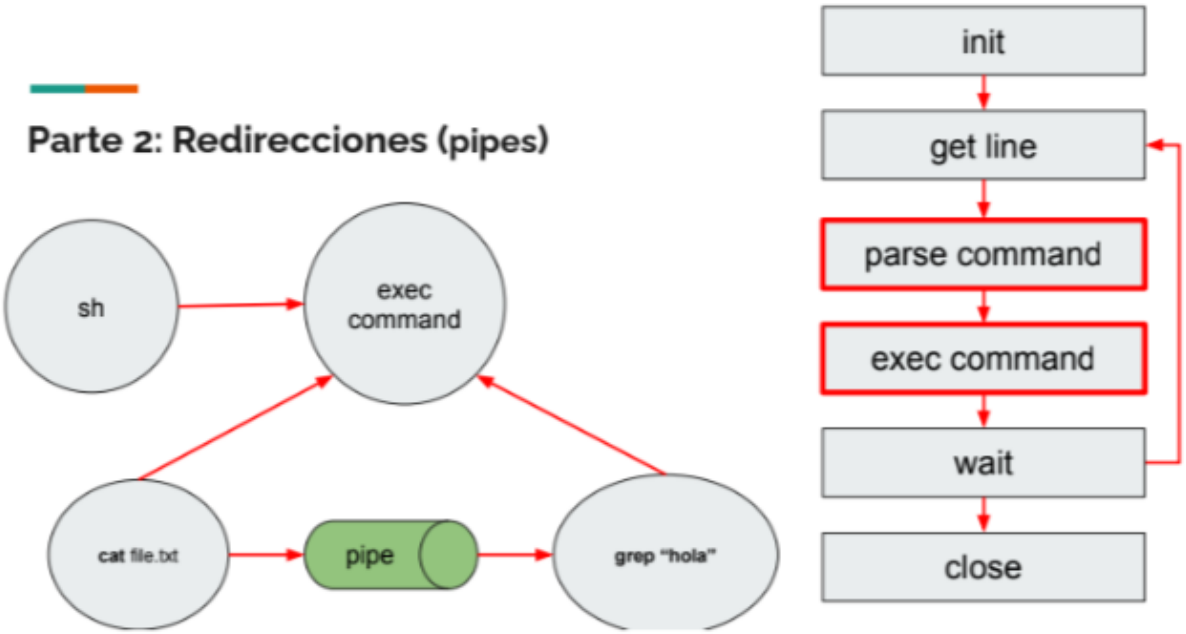
- ¿Cuánto tarda echo hi | sleep 5 en ejecutarse? 5 segundos.
- ¿Qué ocurre con sleep 5 | sleep 2? Tarda 5 segundos. No tarda 2 porque el comando completo es sleep 5 | sleep 2 por lo que la shell va a esperar que ese comando termine, no espera a que el último proceso del pipe termine sino a que todos los procesos del pipe terminen. No tarda 7 porque los procesos de pipe se ejecutan en paralelo para que a medida que uno escribe haya otro que lea sino corremos el riesgo que el pipe se cree y se nos quede todo bloqueado.

Observación: Puedo tener n pipes concatenados y siempre todos se ejecutan en paralelo.



Esto es conceptualmente cómo queremos que los procesos se vean.
Se interpreta a la flecha que entra al pipe, como escribo en el pipe y a la que sale, como leo del pipe.
Estamos en el punto en que la shell tiene un hijo, llama a exec_ cm y queremos saber qué hacemos con un comando que es un pipe.
La shel hace un fork() origina, ese fork() se convierte en cat y hago otro fork() para convertirlo en grep(). Esta es una forma engorrosa de implementarlo cuando tenemos muchos pipes por lo que no se recomienda.

Lo más óptimo es seguir la siguiente estructura:



Todos los círculos (sh, exec command, cat file.txt, grep “hola”) son procesos. La shell expoinear un proceso y en vez de llamar a exec se convierte en el proceso coordinador de

pipes que va a expoinear otros 2 procesos y los va a concatenar con pipes. Esta es una posible implementación y es la recomendada.

¿Qué ventaja tiene? La shell va a hacer wait al coordinador y este puede hacer wait a los 2 procesos hijo y crea el pipe. Esta forma es la recomendada porque la shell tiene que esperar a un solo proceso. Además permite implementar de manera sencilla pipes anidados.

A **diferencia de lo que hacíamos en lab fork()** ahora vamos a usar el pipe para pisar los files descriptors de entrada y salida de los procesos que vamos a invocar. Para conectar el pipe con los files descriptors de los procesos vamos a usar dup(). Vemos otra utilidad de dup() además de open(). Nos garantizamos que el proceso coordinador espere a sus 2 procesos hijos mediante 2 wait().

Para **esperar a múltiples hijos** tenemos que hacer múltiples llamadas a wait(). Se llama a fork()-exec() (para cada proceso) y después a los wait(). Si se invoca todo junto no funciona ya que se ejecuta secuencialmente y no queremos esto dado que se bloquea la tubería.
Entonces el coordinador hace fork() y exec() dos veces para generar ambos procesos, crear el pipe, hacer las redirecciones hacia ese pipe y después hacer el wait() para ambos.

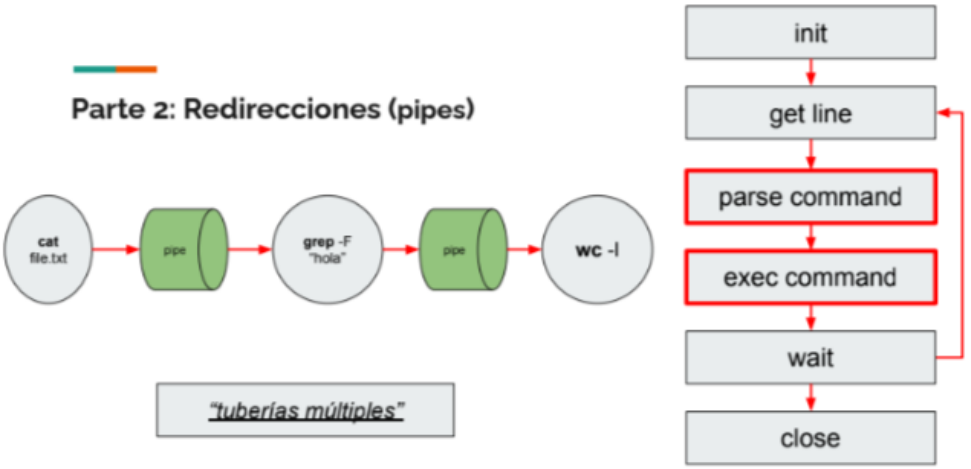
Si creo un pipe ahora el file descriptor 3 va a ser la entrada de escritura del pipe, haciendo dup() redireccionamos los files descriptors según corresponda y finalmente cerramos los que correspondan. Recordar que dup() tiene 2 files descriptors, usa solo 1 y cierra el que usa por lo que hay que cerrar el otro.

<u>Tuberías múltiples</u>

Se espera ejecutar múltiples pipes anidados.

Hay 2 formas de implementarlos.

1) Un coordinador con n hijos → forma en cadena
2) Uno de los extremos del coordinador se convierte en el segundo coordinador → forma recursiva (ésta es la recomendada ya que resulta simple hacer los wait() y que todos se esperen entre sí y es elegante de implementar ya que se puede usar recursividad.)

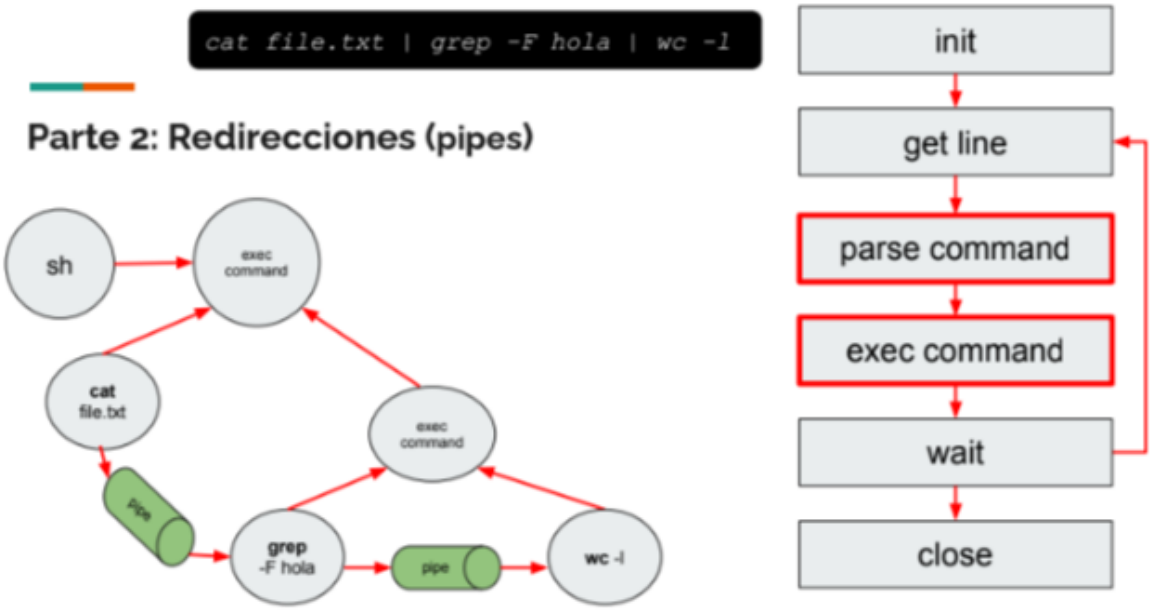


Creamos el pipe que va a tener 2 files descriptors. Antes de llamar a exec() pero antes de llamar a fork(), vamos a usar dup() para que el file descriptor que era de escritura del pipe() sea pisado por el fd[1] del proceso izquierdo.

Ejemplo

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]){
      if(argc<2){
            printf("Se necesita un argumento\n");
            exit(-1);
      }
      int fds[2];
      pipe(fds);
      close(fds[0]);
      int dupfd = dup2(fds[1],1);
      printf("archivo abierto en %d, dup2 devuelve
%d\n",fd,dupfd);
      execvp(argv[1],argv+1);
      printf("terminando: %d\n",getpid());
}
```

Esto me pisó la salida estándar por un extremo de escritura de un pipe. Después se llama a exec y el proceso ve un 0, 1 y 2 y no sabe que el 1 apunta a un extremo de un pipe.



Si queremos ejecutar el siguiente comando. Necesitamos 3 procesos conectados por 2 pipes.
El hijo derecho en lugar de convertirse en el grep, se va a convertir en otro coordinador que va a expoinar 2 procesos.

Ese nuevo coordinador va a redireccionar el pipe a otros 2 procesos. Al nuevo hijo izquierdo le va a conectar el pipe original y después crea otro pipe para sus 2 hijos. De esta forma el nuevo hijo izquierdo se convierte en el grep y el nuevo hijo derecho en el wc.
Si hay más pipes anidados, recursivamente, podríamos hacer que el hijo derecho se convierta en otro coordinador y así ir concatenando N procesos. En este esquema, cada proceso coordinador espera a sus 2 procesos con lo cual va a haber una cadena de wait() entre todos los coordinadores hasta esperar al último proceso.

Entonces el hijo izquierdo hace fork() y exec(), el derecho hace fork() y en lugar de hacer exec() va a llamar a la función que lo convierte en coordinador. El hijo derecho no va a hacer exec() pero si dos fork() para generar procesos que si hacen exec().