UNIVERSIDAD DE ALCALÁ

Departamento de Automática

Grado en Ingeniería de Computadores

Curso 2016/2017

Práctica 3: Herramientas de desarrollo y Servicios POSIX para la gestión de procesos

Sistemas Operativos

Índice

1. Competencias asociadas a la práctica

1. Ser capaz de comprender el funcionamiento de los servicios POSIX para la gestión de procesos vistos en las clases teóricas.
2. Ser capaz de comprender cómo se modifica la jerarquía de procesos en UNIX/Linux cuando se utiliza cada uno de los servicios POSIX de gestión de procesos.
3. Ser capaz de aplicar los diferentes servicios POSIX para la gestión de procesos.
4. Comprender el funcionamiento del intérprete de órdenes.
5. Aplicar los conocimientos sobre los servicios POSIX de procesos y su funcionalidad en el desarrollo de una aplicación que cree procesos, como el intérprete de órdenes.
6. Aplicar los conocimientos vistos en teoría sobre las herramientas de desarrollo clásicas en Unix: `gcc`, uso de bibliotecas estáticas, `make`, y `gdb`.
7. Ser capaz de desarrollar aplicaciones modulares.
8. Ser capaz de trabajar en equipo en el desarrollo de aplicaciones.

2. Introducción

En prácticas anteriores se ha introducido al alumno en el uso de un intérprete de órdenes, *bash*, como interfaz de usuario en Linux. Esta práctica tiene como objetivo principal introducir al alumno en la interfaz de aplicaciones, comenzando con la aplicación, a alto nivel, de las llamadas al sistema para gestionar procesos. Para ello, se realizarán varios ejercicios junto con un análisis más en detalle de la funcionalidad del intérprete de órdenes. El alumno aplicará los conceptos vistos en las clases teóricas y en el laboratorio mediante la implementación parcial de un intérprete de órdenes (al que denominaremos *minishell*).

Asimismo, esta práctica servirá para que el alumno utilice las herramientas de desarrollo clásicas empleadas en los sistemas Unix. Además, un sub-objetivo importante dentro de la práctica es la adquisición por parte del alumno de buenas prácticas de codificación y diseño modular de aplicaciones.

Hablar de programación bajo Unix es hablar de C. C se creó con el único objetivo de recodificar Unix (que en sus orígenes estaba programado en ensamblador) en un lenguaje de alto nivel, decisión que, por cierto, fue revolucionaria en su época. Además, todo el desarrollo de Arpanet y su sucesora, Internet, se basó en la utilización de máquinas Unix. Por lo tanto, C ha tenido un impacto decisivo en el desarrollo de la informática, hasta el punto de que la práctica totalidad de los lenguajes más extendidos en la actualidad han tomado elementos de C, o directamente son una evolución del mismo.

3. Ciclo de creación de un programa

Como en cualquier otro entorno, crear un programa bajo UNIX requiere una serie de pasos, que deben ser ya conocidos por el alumno:

- *Creación y edición* del programa o **código fuente** sobre un archivo de texto, empleando para ello una herramienta denominada **editor**. En caso de programar en lenguaje C, el convenio es que dicho archivo tenga extensión "`.c`".

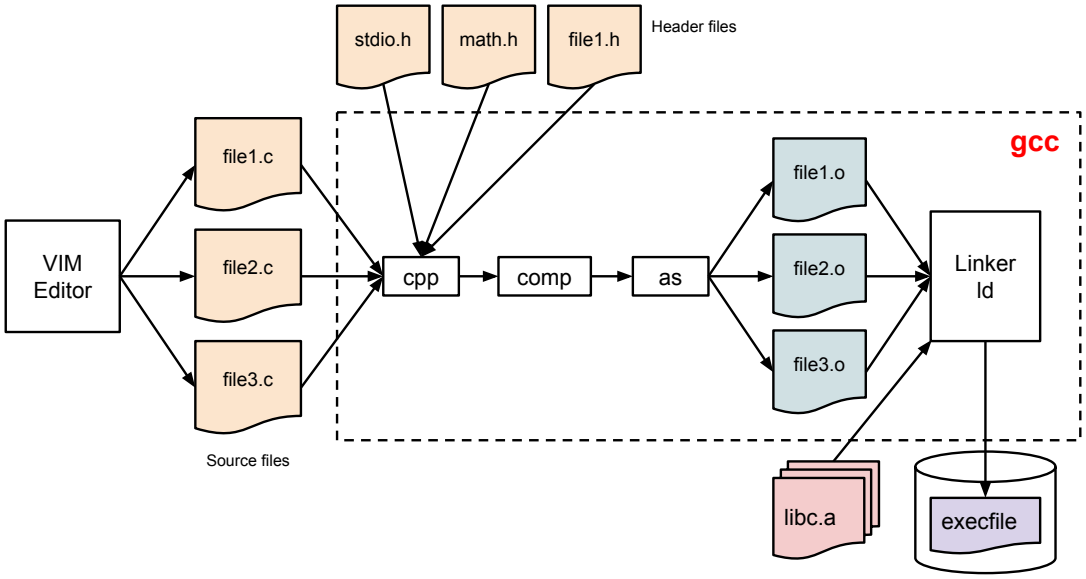

Figura 1: Generación de un archivo ejecutable en lenguaje C.

- *Compilación* del código fuente mediante otra herramienta denominada **compilador**, generándose un **archivo objeto**, que en UNIX suele tener extensión "`.o`". A veces, en lugar de generarse el archivo objeto directamente se genera un archivo intermedio en ensamblador (en UNIX típicamente con extensión "`.s`") que, a continuación, es necesario ensamblar con un **ensamblador** para obtener el archivo objeto. Además, en C existe una etapa previa a la compilación en la que el archivo fuente pasa por otra herramienta denominada **preprocesador**.

- *Enlazado* del archivo objeto con otros archivos objeto necesarios, así como con las bibliotecas que sean necesarias para así obtener el **archivo de programa ejecutable**. Esta labor es llevada a cabo por un programa denominado **enlazador**.

- Finalmente, *ejecutar* el programa tal cual o, en caso de ser necesario, *depurar* el programa con una herramienta denominada **depurador** o *debugger*. Si se detectan problemas en la ejecución, será necesario editar el programa fuente y corregir los fallos, reiniciándose el ciclo de desarrollo hasta obtener un programa sin errores.

El proceso anterior queda ilustrado en la Figura 1. En un sistema Linux, como el empleado en el laboratorio, las herramientas clásicas encargadas de cada etapa son las siguientes:

- **Editor**: existe gran variedad de ellos, siendo los más populares **`emacs`** y **`vim`**. En el laboratorio se recomienda el uso del editor **`vim`**.

- **Preprocesador**, **compilador**, **ensamblador**, y **enlazador**: estas herramientas pueden encontrarse de forma individual, pero en un sistema Linux típicamente encontramos la herramienta **`cc`** (realmente la herramienta que invoca la compilación en C y C++ de GNU, `gcc`), que se encarga de llamar al preprocesador, compilador, ensamblador, y enlazador, según se necesiten. En cualquier caso, y aunque por lo general no las utilizaremos de forma individual, las herramientas son:

  - Preprocesador: `cpp`.
  - Compilador: `comp`.

- Ensamblador: `as`.
- Enlazador: `ld`.

- **Depurador**: el depurador por excelencia en el entorno Linux (y en UNIX en general) es `gdb`.

Vamos a analizar cada fase con más detalle.

3.1. Edición del archivo fuente

La edición del código fuente se puede realizar con cualquier programa capaz de editar archivos en texto plano. Por su amplia difusión, gran versatilidad y velocidad de edición se utiliza mucho el editor `vim`, o versiones gráficas como `gvim`.

El editor `vim` es mucho más potente y rápido que los editores típicos de MS-DOS o Windows, como el `Bloc de Notas` o el `edit`, y otros editores de entorno de ventanas, como `gedit`. Sin embargo, el uso de `vim` al principio puede ser un poco frustrante, por lo que si el alumno no conoce el manejo de esta herramienta es recomendable seguir las notas que se pueden descargar desde la página web de la asignatura..

La forma de editar el programa será:

```
user@host:$ vim programa.c
```

Para salir del editor hay que pulsar escape y posteriormente teclear "`q!`" si no se desea guardar los cambios, o bien "`:wq`" si se quieren guardar los cambios.

3.2. Compilación y enlazado

El programa `gcc` es el encargado de compilar, enlazar y generar el archivo ejecutable a partir del archivo fuente. La forma más sencilla de invocarlo es:

```
user@host:$ gcc programa.c
```

Esta orden preprocesa, compila, ensambla y enlaza, generando el archivo de salida ejecutable `a.out`. Típicamente no se desea que esto sea así, por lo que se emplea la opción `-o` para establecer el nombre del archivo generado:

```
user@host:$ cc programa.c -o programa
```

Para ejecutar el programa es necesario invocarlo de la forma siguiente:

```
user@host:$ ./programa
```

Es necesario el empleo del `./` antes del nombre de programa para indicarle al intérprete de órdenes que el programa reside en el directorio actual de trabajo (`.`), ya que en caso de no especificarlo, sólo se buscaría el programa en aquellos directorios del sistema especificados en la variable de entorno `PATH`.

```
user@host:$ echo $PATH /usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

El programa `gcc` es muy flexible y soporta un gran cantidad de opciones que le permiten, por ejemplo, detenerse tras un determinado paso del proceso (por ejemplo, de la compilación) o aceptar varios archivos de entrada incluso de diferentes tipos (fuente, objeto o en ensamblador), siendo capaz de procesar cada uno de ellos de la manera adecuada para generar el archivo de salida que se le pide.

Por ejemplo, la siguiente orden compila el archivo `programa.c` y el objeto resultante lo enlaza con el objeto `funciones.o`, dando como resultado el programa de nombre `programa`:

```
user@host:$ gcc programa.c funciones.o -o programa
```

La siguiente orden compila los archivos fuente indicados, generando los archivos objeto de cada uno de ellos pero no continúa con el enlazado y generación del ejecutable final:

```
user@host:$ gcc -c programa.c funciones.c
```

A continuación se muestra un resumen de las opciones más frecuentes con las que se suele invocar `gcc`.

Opción	Descripción
`-E`	Parar tras el preprocesado.
`-S`	Parar tras el compilado (no ensamblar).
`-c`	Parar antes de enlazar.
`-o` nombre	Especificar el nombre del archivo de salida.
`-g`	Incluir información de depuración.
`-Wall`	Mostrar todos los avisos de compilación.
`-pedantic`	Comprobar que el programa es C estándar.
`-On`	Grado de optimización de código, donde n es un entero desde $n = 0$ (ninguna) a $n = 3$ (máxima).
`-D macro[=valor]`	Definir una macro (`#define`) y su valor (1 si se omite).
`-I directorio`	Indica un directorio en donde buscar archivos de cabecera (`.h`).
`-L directorio`	Indica un directorio en donde buscar bibliotecas compartidas.
`-lbiblioteca`	Utilizar la biblioteca compartida `lbiblioteca.a`.
`-static`	Enlazar bibliotecas estáticamente.

3.3. Depuración

El depurador estándar de UNIX es `gdb`. Se trata de un depurador muy potente y extendido en la industria, pero sólo admite una interfaz de línea de órdenes que, a priori, puede resultar engorrosa y difícil de aprender. Existen *frontends* gráficos para poder trabajar con él de un modo más cómodo, como por ejemplo la aplicación `ddd`.

El uso del depurador es necesario para la rápida detección de errores en los programas, y por lo tanto debe ser una prioridad para el alumno. El ahorro de tiempo que conlleva su aprendizaje supera con mucho el tiempo perdido en perseguir errores difíciles de localizar de forma visual o mediante el uso de otras "técnicas" de depuración como sembrar los programas de llamadas a `printf()`.

Como material de apoyo de la práctica se ha incluido un breve tutorial sobre el uso de `gdb`.

A pesar de la gran utilidad y potencia de `gdb`, no hay que perder la perspectiva de que la depuración engloba toda una serie de actividades que van más allá del uso de la herramienta, sin olvidar que el depurador es una herramienta fundamental. La mejor depuración es la que no es necesario hacer, para ello se puede recurrir a muchos "trucos" que se aprenden con la experiencia, como desarrollar de manera incremental, añadiendo poco a poco funciones a nuestro programa, verificar la corrección de cada funcionalidad nueva, abordar un único problema a la vez, etc. Las actividades propuestas en esta práctica están pensadas en esta línea; se sugiere observar cómo se aborda la construcción de los programas.

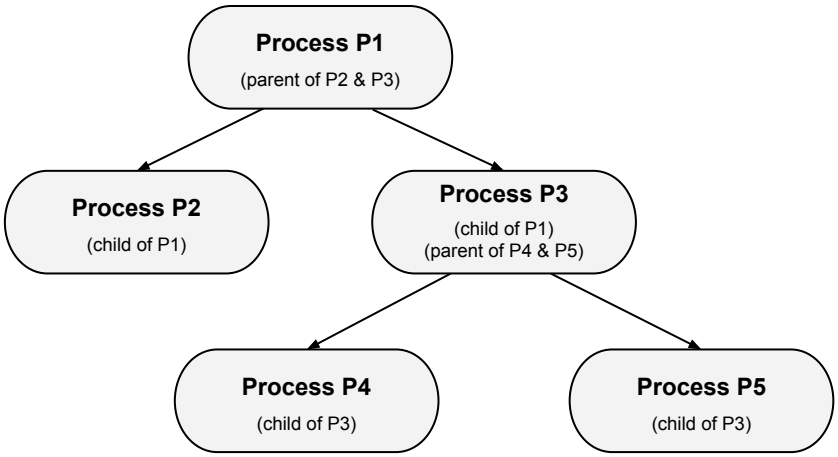

Figura 2: Jerarquía de procesos en UNIX

4. Automatización del proceso de desarrollo de aplicaciones: `make`

El desarrollo de un programa es una actividad cíclica en la que puede ser necesario recompilar muchas veces múltiples archivos fuente dependientes unos de otros. Esto, aparte de la incomodidad, puede ser causa de un desarrollo lento y sensible a errores.

Para evitarlo en lo posible, en los entornos UNIX suele ser habitual el uso de la herramienta `make` para automatizar las reconstrucciones del código y minimizar el tiempo de compilación. Para ello, sólo es necesario crear un archivo llamado `Makefile`, que especifica la forma en que deben construirse los objetos, ejecutables o bibliotecas, y las dependencias entre ellos. Además, `make` también es capaz de hacer otras cosas como ejecutar incondicionalmente conjuntos de órdenes.

Puede encontrar numerosa documentación en Internet acerca del uso del `make`. En la página web de la asignatura de Sistemas Operativos, en la sección *Material complementario* de la práctica 3, se han incluido algunos enlaces que se consideran de interés para el estudio de esta herramienta.

Aunque el uso de `make` pueda parecer innecesario en proyectos pequeños, su uso fuerza a adquirir buenas prácticas y hábitos de desarrollo. Por ello, **su uso es obligatorio en todas las prácticas** de Sistemas Operativos y Sistemas Operativos Avanzados.

5. Modelo de los procesos en UNIX

En Unix todo proceso es creado por el núcleo del SO, previa petición de otro proceso, estableciéndose una relación jerárquica entre el proceso que realiza la petición de creación, conocido como **proceso padre**, y el nuevo proceso, denominado **proceso hijo**. Un proceso padre puede tener varios hijos y todo hijo tiene únicamente un padre, tal y como se puede apreciar en la Figura 2.

Hay una cuestión importante que cabe mencionar: si todo proceso necesita tener un proceso padre, £cuál es el primer proceso del que se originan todos los procesos en UNIX? La respuesta se encuentra en el proceso *init*, que es un proceso lanzado por el SO durante el arranque del sistema y es el encargado de iniciar los procesos necesarios para poder operar. De hecho, todos los procesos que hay en el sistema descienden de una manera u otra del proceso *init*. La jerarquía de procesos existente en una máquina puede consultarse por medio de la orden `pstree`, si bien no está presente por defecto en todas las instalaciones de Linux.

Por lo general, en un SO no todos los procesos tienen la misma prioridad. En el caso de Unix, a cada proceso se le puede asociar a una prioridad distinta. La prioridad es un mecanismo que

permite al SO dar un trato de privilegio a ciertos procesos a la hora de repartir la utilización de la CPU. Por ejemplo, resultaría poco eficiente asignar la misma prioridad a un proceso que se ejecuta en *background* que a un proceso interactivo, que tiene a un usuario pendiente de obtener su respuesta. Normalmente, es el propio SO quien se encarga de asignar prioridades. Sin embargo, en Unix es posible que un usuario modifique dichas prioridades por medio de la orden `nice`.

Internamente, un SO necesita guardar información de todos los elementos del sistema, y para poder localizar dicha información necesita manejar una identificación de dichos elementos, como sucede con el DNI y las personas. La forma más cómoda para el computador es trabajar con números, y de hecho, estos identificadores son tan importantes que se les ha dado un nombre propio. A los identificadores de los archivos se les denomina *número de nodo índice*, mientras que los identificadores de los procesos se llaman **PID** (*Process IDentificator*). Todo proceso en el sistema tiene un único PID asignado, y es necesario para poder identificar a dicho proceso en algunas órdenes. Asimismo, para que el SO pueda localizar con facilidad al proceso padre, todo proceso tiene asignado un segundo número conocido como **PPID** (o *Parent Process IDentificator*). Se puede conocer el PID y PPID de los procesos que se están ejecutando en el sistema por medio de dos órdenes muy útiles para obtener información sobre procesos: *ps* y *top*.

6. Llamadas al sistema y servicios POSIX

Las llamadas al sistema son el mecanismo proporcionado por el SO utilizado para que los desarrolladores puedan, en última instancia, acceder a los servicios ofrecidos por el SO. Estrictamente hablando, las llamadas al sistema se definen en ensamblador, y por lo tanto no son portables entre distintas arquitecturas. Esta situación hace que la programación sea dificultosa y no portable. Por este motivo, las llamadas al sistema se envuelven con un envoltorio (o rutinas `wrapper` en Linux) en forma de función en lenguaje de alto nivel, típicamente C. De este modo, el programador va a percibir la llamada al sistema como una mera llamada a una función y el código que desarrolle será portable a otras plataformas que soporten estas rutinas `wrapper`.

El estándar POSIX precisamente define la firma de las funciones que componen dicho envoltorio en sistemas UNIX. La presente práctica tiene como objeto introducir al alumno en la programación de llamadas al sistema por medio de la interfaz POSIX. Para ello, se utilizarán servicios POSIX relacionados con la gestión de procesos.

7. Servicios POSIX para la gestión de procesos

Entre los aspectos más destacados de la gestión de procesos en UNIX/Linux se encuentra la forma en que éstos se crean y cómo se ejecutan nuevos programas. En esta sección se describen los principales servicios proporcionados por POSIX para el manejo de procesos.

7.1. Servicio POSIX `fork()`

El servicio POSIX `fork ()` permite crear un proceso. El sistema operativo trata este servicio llevando a cabo una clonación del proceso que lo invoca, conocido como proceso padre del nuevo proceso creado, denominado proceso hijo. Todos los procesos se crean a partir de un único proceso padre lanzado en el arranque del sistema, el proceso *init*, cuyo PID es 1 y que, por lo tanto, está situado en lo más alto en la jerarquía de procesos de UNIX, como ya se ha mencionado en la sección 5.

El servicio POSIX `fork ()` duplica el contexto del proceso padre y se le asigna este nuevo contexto al proceso hijo. Por lo tanto, se hace una copia del contexto de usuario del proceso padre -de su código, datos y pila, y de su contexto de núcleo -entrada del bloque de control de procesos del proceso padre. Ambos procesos se diferenciarán esencialmente en el PID asociado a cada uno de ellos. Si la función se ejecuta correctamente, retorna al proceso padre el identificador (PID) del

proceso hijo recién creado, y al proceso hijo el valor 0. Si, por el contrario, la función falla, retorna -1 al padre y no se crea ningún proceso hijo. Su sintaxis es la siguiente:

```
pid_t fork()
```

7.2. Servicio POSIX `exec()`

El servicio POSIX `exec()` permite cambiar el programa que se está ejecutando, reemplazando el código y datos del proceso que invoca esta función por otro código y otros datos procedentes de un archivo ejecutable. El contenido del contexto de usuario del proceso que invoca a `exec()` deja de ser accesible si la función se ejecuta correctamente, y es reemplazado por el del nuevo programa. Por lo tanto, en estas condiciones, el programa antiguo es sustituido por el nuevo, y nunca se retornará a él para proseguir su ejecución. Si la función falla, devuelve -1 y no se modifica la imagen del proceso. La declaración de la familia de funciones `exec` es la siguiente:

```
int execl (const char *camino, const char *arg0, ...)
int execlp (const char *archivo, const char *arg0, ...)
int execle (const char *camino, const char *arg0, ... ,
            char *envp[])
int execv (const char *camino, char *const argv[])
int execvp (const char *archivo, char *const argv[])
int execve (const char *archivo, const char *argv[],
            char *envp[])
```

Parámetros

camino ruta completa del nuevo programa a ejecutar.

archivo se utiliza la variable de entorno `PATH` para localizar el programa a ejecutar. No es necesario especificar la ruta absoluta del programa si éste se encuentra en alguno de los directorios especificados en `PATH`.

argi argumento `i` pasado al programa para su ejecución.

argv[] array de punteros a cadenas de caracteres que representan los argumentos pasados al programa para su ejecución. El último puntero debe ser `NULL`.

envp[] array de punteros a cadenas de caracteres que representan el entorno de ejecución del nuevo programa.

7.3. Servicio POSIX `exit()`

El servicio POSIX `exit()` termina la ejecución del proceso que lo invoca. Como resultado, se cierran todos los descriptores de archivos abiertos por el proceso y, a través de su parámetro, proporciona una indicación de cómo ha finalizado. Esta información podrá ser recuperada por el proceso padre a través del servicio POSIX `wait()`. Su sintaxis es la siguiente:

```
void exit(int status)
```

Parámetros

status almacena un valor que indica cómo ha finalizado el proceso: 0 si el proceso terminó correctamente, y distinto de 0 en caso de finalización anormal.

7.4. Servicios POSIX `wait()` y `waitpid()`

Ambos servicios esperan la finalización de un proceso hijo y además permiten obtener información sobre su estado de terminación. Si se ejecuta correctamente, `wait()` retorna el PID (identificador) del proceso hijo cuya ejecución ha finalizado y el código del estado de terminación del proceso hijo en el parámetro del servicio (si éste no es `NULL`). Por el contrario,el servicio devuelve -1 si el proceso no tiene hijos o estos ya han terminado.

Un ejemplo de uso de este servicio es cuando un usuario escribe una orden en el intérprete de órdenes de UNIX. El intérprete crea un proceso (*shell* hija) que ejecuta la orden (el programa) correspondiente. Si la orden se ejecuta en primer plano, el padre esperará a que finalice la ejecución de la *shell* hija. Si no, el padre no esperará y podrá ejecutar otros programas.

Un proceso puede terminar y su proceso padre no estar esperando por su finalización. En esta situación especial el proceso hijo se dice que está en estado *zombi*; ha devuelto todos sus recursos excepto su correspondiente entrada en la tabla de procesos. En este escenario, si el proceso padre invoca a `wait()`, se eliminará la entrada de la tabla de procesos correspondiente al proceso hijo muerto.

La sintaxis del servicio `wait()` es la siguiente:

```
pid_t wait(int *status)
```

Parámetros

status si no es `NULL`, almacena el código del estado de terminación de un proceso hijo: 0 si el proceso hijo finalizó normalmente, y distinto de 0 en caso contrario.

`waitpid()` es un servicio más potente y flexible de espera por los procesos hijos ya que permite esperar por un proceso hijo particular. La sintaxis del servicio `waitpid()` es la siguiente:

```
pid_t waitpid(pid_t pid, int*status,int options)
```

Este servicio tiene el mismo funcionamiento que el servicio `waitpid()` si el argumento `pid` es -1 y el argumento `status` es cero.

Parámetros

pid Si es `-1`, se espera la finalización de cualquier proceso. Si es `>0`, se espera la finalización del proceso hijo con identificador `pid`. Si es 0, se espera la finalización de cualquier proceso hijo con el mismo identificador de grupo de proceso que el del proceso que realiza la llamada. Si es `<-1`, se espera la finalización de cualquier proceso hijo cuyo identificador de grupo de proceso sea igual al valor absoluto del valor de `pid`.

status Igual que el servicio `wait()`.

options Se construye mediante el OR binario ('|') de cero o más valores definidos en el archivo de cabecera `sys/wait.h`. Es de especial interés el valor de `options` definido con `WNOHANG`. Este valor especifica que la función `waitpid()` no suspenderá (no bloqueará) al proceso que realiza este servicio si el estado del proceso hijo especificado por `pid` no se encuentra disponible. Por lo tanto, esta opción permite que la llamada `waitpid()` se comporte como un servicio *no bloqueante*. Si no se especifica esta opción, `waitpid` se comporta como un servicio *bloqueante* .

8. Servicios POSIX para comunicación entre procesos

Los procesos no son entes aislados, sino que es necesario que sean capaces de relacionarse con otros procesos para lograr un objetivo común. A continuación, se describen algunos servicios POSIX que posibilitan esta comunicación.

8.1. Tuberías sin nombre (*pipes*)

Las tuberías sin nombre o pipes (|) son familiares para la mayoría de los usuarios de UNIX, shell. Así, por ejemplo, para imprimir los nombres de que las han utilizado directamente en la los usuarios conectados en el sistema por orden alfabético se utilizaría:

```
user@host:$ who | sort | lp
```

En dicha orden aparecen tres procesos conectados mediante tuberías. Aunque en este ejemplo el ujo de datos es en una única dirección (de `who` a `sort` y de `sort` a `lp`), es posible conectar procesos de manera que el ujo de datos sea bidireccional o, incluso, formando anillos de comunicación. Sin embargo, estas formas de utilizar las tuberías sin nombre sólo están disponibles a través del interfaz de llamadas al sistema, por lo que algunos usuarios de UNIX desconocen estas posibilidades.

La llamada al sistema `pipe()` crea un canal de comunicaciones representado por dos descriptores de archivo retornados en el array que se le pasa como argumento a la llamada. Este canal de comunicaciones funciona como una estructura FIFO (*First In, First Out*).

```
int pipe(int pipefd[2])
```

Suponiendo que el array pasado se denomina `fd`, escribiendo en el descriptor `fd[1]` se introducen datos en la tubería, y leyendo del descriptor `fd[0]` se extraen datos de la misma. Una vez creada la tubería, se pueden emplear las llamadas al sistema generales para el manejo de archivos. La llamada al sistema `open()`, utilizada de forma general para abrir un archivo y obtener su descriptor, no se usa en este caso, ya que `pipe()` es la encargada de crear las tuberías. El resto de las funciones tienen un comportamiento equivalente:

`read()` Los datos se leen de una tubería en el orden de llegada, es decir, en el mismo orden en el que se escribieron. No es posible realizar operaciones de lectura de naturaleza no secuencial sobre una tubería sin nombre. Una vez que los datos son leídos se eliminan de la tubería, de forma que no pueden ser vueltos a leer (lectura destructiva). Normalmente, la lectura es bloqueante, esto significa que si se realiza una llamada `read()` sobre una tubería vacía, el proceso que realiza la llamada se bloquea hasta que otro proceso introduzca datos en dicha tubería, salvo que todos los descriptores de escritura de la tubería sin nombre estén cerrados, en cuyo caso read() devuelve 0. El proceso se desbloquea en cuanto se introduzcan datos en la tubería.

`write()` De manera similar a `read()` los datos se introducen en la tubería según su secuencia de llegada. Cuando ésta se llena, `write()` se bloquea hasta que una llamada a `read()` retire los datos suficientes como para poder completar la llamada.

`close()` Su importancia es mayor que para los archivos normales, puesto que cerrar el extremo de escritura es la única forma de indicar una condición de fin de archivo para los procesos lectores de la tubería. De forma similar, si se cierra el extremo de lectura, cualquier intento de escritura producirá un error.

Para conectar dos o más procesos a través de una tubería sin nombre, la técnica habitual consiste en crear la tubería antes que los procesos hijos, ya que éstos, al heredar la tabla de descriptores de archivos abiertos por el padre, obtendrán la copia de los descriptores asociados a la tubería. Es posible así conectar procesos padre con sus hijos (e incluso nietos), y procesos hijos entre sí; la única condición es que hayan obtenido una copia de los descriptores de la tubería. Sin embargo, no hay forma de conectar dos procesos no emparentados mediante tuberías sin nombre.

Uno de los problemas más habituales cuando se trabaja con tuberías sin nombre es la aparición de abrazos mortales (o interbloqueos) como consecuencia de una mala gestión de las tuberías. Estos interbloqueos suelen surgir cuando un proceso lee de una tubería en la cual nunca se escribe (o viceversa). El siguiente programa es un ejemplo de ello:

Redireccionando la entrada/salida

Cada proceso, en el momento de su creación, dispone de tres archivos abiertos con unos descriptores de archivo específicos. Los archivos concretos apuntados por los descriptores, ya sean regulares o de otro tipo, se heredan en un principio del proceso padre, y pueden ser modificados por el propio proceso durante su ejecución.

`stdin` (0) descriptor correspondiente a la entrada estándar. Este descriptor es el empleado por el proceso para obtener información del exterior. Por defecto, al leer de este descriptor se obtienen caracteres introducidos por el usuario a través del teclado. Las funciones de entrada de la biblioteca estándar de C, como `scanf()`, leen de este descriptor para obtener los caracteres.

`stdout` (1) descriptor correspondiente a la salida estándar. Este descriptor es el empleado por el proceso para enviar información hacia el exterior. Por defecto, al escribir en este descriptor, se envían caracteres al terminal que inició el programa. Las funciones de salida de la biblioteca estándar de C, como `printf()`, escriben en este descriptor para imprimir los datos.

`stderr` (2) descriptor correspondiente a la salida estándar de error. Este descriptor es el empleado por el proceso para enviar información hacia el exterior de posibles errores que ocurran durante la ejecución. Se emplea un descriptor aparte para facilitar la redirección de estos datos y poder diferenciarlos de la salida normal del programa. Por defecto, al escribir en este descriptor, y al igual que ocurre con la salida estándar, se envían caracteres al terminal que inició el programa.

Para modificar el archivo apuntado por cada uno de los tres descriptores, el sistema operativo proporciona dos llamadas al sistema:

```
int dup(int oldfd)
int dup2(int oldfd, int newfd)
```

Estas llamadas se utilizan para crear una copia de un descriptor en otro. De esta forma, se puede establecer dos identificadores para el mismo archivo abierto (o tubería). En realidad no se trata de una copia como tal, sino de un alias. La primera de las funciones copia el descriptor pasado como parámetro en la primera entrada de la tabla de archivos abiertos que no esté ocupada. La segunda copia el descriptor pasado como primer parámetro en el descriptor correspondiente al segundo parámetro. Si el segundo descriptor está abierto previamente, el archivo original se cierra antes de hacer la copia.

Mediante el uso de las funciones `dup()` y `dup2()`, se pueden redireccionar los descriptores correspondientes a la entrada/salida de una forma similar a ésta:

```
int fd = open(‘‘output.txt’’);
dup2(fd, 1);
```

En este ejemplo, primero se realiza una llamada al sistema `open()` para abrir un archivo (en este caso, “output.txt”). Esta llamada devolverá en `fd` el descriptor correspondiente a la posición del archivo en la tabla de archivos abiertos. A continuación, con la llamada al sistema `dup2()`, se establece que el descriptor 1, correspondiente a la salida estándar, pasará a apuntar al mismo archivo apuntado por el descriptor `fd`. A partir de ese momento, todas escrituras que se realicen al descriptor 1, se almacenarán en el archivo “output.txt”, en vez de ser enviadas al terminal. Si se desea utilizar `dup()` en vez de `dup2()`, es necesario primero cerrar (empleando la llamada al sistema `close()`) el descriptor de destino, de tal forma que quede disponible. Una vez cerrado, la llamada al sistema `dup()` lo seleccionará como destino del descriptor que se desea duplicar. El siguiente código es equivalente al anterior:

```
int fd = open(‘‘output.txt’’);
close(1);
dup(fd);
```

Una combinación de tuberías y redirecciones permite establecer una comunicación entre procesos de la misma forma que cuando utilizamos el carácter (|) en el terminal. Cuando se desea comunicar dos procesos de tal forma que la salida estándar de uno de ellos esté directamente conectada a la entrada estándar del otro, es necesario emplear una tubería creada previamente mediante la llamada al sistema `pipe()`. Esta llamada tendrá que ser realizada por el proceso padre de los dos procesos que se desea intercomunicar, ya que ambos tienen que heredar los descriptores correspondientes a los extremos de la tubería. Dichos procesos deberán realizar llamadas a `dup()` o `dup2()` de forma independiente para poder redireccionar la entrada y salida estándar y que los descriptores 0 y 1 apunten respectivamente a los extremos de lectura y escritura de la tubería. El proceso que generará los datos deberá redireccionar la salida estándar, y el proceso que esperará recibirlos deberá hacer lo propio con la entrada. El siguiente código es un ejemplo de este procedimiento:

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

struct s_command
{
  char ** argv;
  int argc;
};

int main()
{
    int res;
    int pid_cat, pid_wc;
    int pipefd[2];

    // Array of commands
    struct s_command commands[2];

    // Fill command[0] with "cat /etc/passwd"
    commands[0].argv = (char **)malloc(3 * sizeof(char *));
    commands[0].argv[0] = "cat";
    commands[0].argv[1] = "/etc/passwd";
    commands[0].argv[2] = NULL;
    commands[0].argc = 2;

    // Fill command[1] with "wc"
    commands[1].argv = (char **)malloc(2 * sizeof(char *));
    commands[1].argv[0] = "wc";
    commands[1].argv[1] = NULL;
    commands[1].argc = 1;

    // Create the pipe
    res = pipe(pipefd);
    if (res < 0)
    {
        perror("Error creating the pipe");
        exit(-1);
    }

    // Create processes
    if ((pid_cat = fork()) == 0)
    {
        // I'm the first child and I will execute "cat"
        dup2(pipefd[1], 1); // pipe "write" end -> stdout

        close(pipefd[0]); // Close all the pipes
        close(pipefd[1]); // so that we don't hung ourselves

        execvp(commands[0].argv[0], commands[0].argv); // Execute command

        // If we passed the "exec", something wrong happened
        perror("error when executing cat");
        exit(-1);
    }
    else if ((pid_wc = fork()) == 0)
    {
```

```
59          // I'm the third child and I will execute "wc"
60          dup2(pipefd[0], 0); // pipe "read" end -> stdin
61
62          close(pipefd[0]); // Close all the pipes
63          close(pipefd[1]); // so that we don't hung ourselves
64
65          execvp(commands[1].argv[0], commands[1].argv); // Execute command
66
67          // If we passed the "exec", something wrong happened
68          perror("error when executing wc");
69          exit(-1);
70      }
71      else
72      {
73          // I'm the father
74          close(pipefd[0]); // Close all the pipes
75          close(pipefd[1]); // so that we don't hung ourselves
76
77          // And then I'll wait for my children
78          waitpid(pid_cat, NULL, 0);
79          waitpid(pid_wc, NULL, 0);
80      }
81  }
```

En el código anterior se crean dos procesos. El primero de ellos ejecuta el programa "`cat /etc/passwd`" mientras que el segundo ejecuta el programa "`wc`" sin argumentos. El proceso principal crea primero la tubería y, posteriormente, crea los dos procesos. El primero de ellos, al comenzar su ejecución, redirecciona la salida estándar hacia el extremo de escritura de la tubería. El segundo, en cambio, redirecciona la entrada estándar hacia el extremo de lectura. El proceso padre, una vez que ha creado los dos procesos, espera a que estos finalicen antes de terminar su ejecución.

8.2. Servicio POSIX `sigaction()`

Las señales son un mecanismo de comunicación asíncrono gestionado por el sistema operativo y muy utilizado para la notificación **por software** de eventos y situaciones especiales a los procesos. Tiene una gran utilidad de cara a afrontar situaciones en las cuales se producen eventos en momentos del tiempo sin determinar, de manera que se interrumpe el flujo secuencial dentro de nuestra aplicación para activar la tarea asociada a la señal.

El funcionamiento es muy similar al de las interrupciones pero la notificación del evento, a diferencia de las interrupciones que se lleva a cabo por hardware (se activa una determinada entrada de la CPU), en el caso de las señales es un mecanismo generado por el propio sistema operativo en función del evento asociado. Una vez que el sistema operativo (motivado por cuestiones internas o por otro proceso) genera una señal, un proceso la recibe y se ejecuta una rutina de tratamiento de esa señal (manejador de señal).

Cuando un proceso que está en ejecución recibe una señal, detiene su ejecución en la instrucción máquina actual y, si existe una rutina de tratamiento de la señal, se ejecuta. Si la rutina de tratamiento no termina el proceso, retorna al punto en que se recibió la señal.

Las señales se identifican mediante un número entero. Para facilitar su uso, todas las señales tienen un nombre simbólico que comienza por el prefijo `SIG`[1]. Ejemplos: un proceso padre recibe la señal `SIGCHLD` cuando termina un proceso hijo, `SIGKILL` (un proceso mata a otro proceso), `SIGALARM` (señal enviada por el sistema operativo a un proceso para indicar que vence un temporizador), etc.

El manejo de señales se realiza por medio del servicio POSIX `sigaction()`. El prototipo de este servicio es el siguiente:

```
int sigaction (int sig, const struct sigaction *act, struct sigaction
*oldact);
```

Servicio POSIX que permite configurar la señal, es decir, especificar un manejador para la señal `act`. El manejador es la función que se ejecutará cuando se reciba la señal (si no es ignorada). `sigaction()` permite tres opciones cuando llega una señal a un proceso:

[1] Los nombres simbólicos de las señales están definidos en el archivo `<sys/signal.h>`. Si nuestro programa maneja señales, basta con incluir el archivo `<signal.h>`, que incluye al anterior.

1. Ignorar a la señal: No se ejecuta ningún manejador cuando se entrega la señal al proceso de destino pero éste puede realizar alguna acción.
2. Llamar a la rutina de tratamiento de la señal por defecto.
3. Llamar a una rutina de tratamiento de la señal propia.

La estructura `sigaction` para señales estándar[2] es la siguiente:

```
struct sigaction
void(*sa_handler)();
sigset_t sa_mask;
int sa_flags;
```

donde:

sa_handler Es un puntero a la función manejador de la señal. Esta función tiene un único parámetro entero, el número de la señal. Existen dos valores especiales para este campo:

SIG_DFL Asigna un manejador por defecto.
SIG_IGN Ignora la señal (no se ejecuta ningún manejador).

sa_mask Especifica la máscara con las señales adicionales que deben ser bloqueadas (pendientes de ser recibidas) durante la ejecución del manejador. Normalmente, se asigna una máscara vacía.

sa_flags Especifica opciones especiales[3].

Parámetros

sig Es el identificador (número entero o nombre simbólico) de la señal que queremos capturar. Se puede consultar un listado completo de señales en la sección 7 de la página `man` de `sigaction`.

act Puntero a la estructura donde se debe especificar la configuración deseada de la señal de tipo `sigaction`, descrito previamente.

oldact Puntero a una estructura de tipo `sigaction` donde se devuelve la configuración previa a la ejecución de la función `sigaction()`. Generalmente, este parámetro se pone a NULL para indicar que no devuelva dicha configuración.

El servicio POSIX `signal()` devuelve `0` en caso de éxito o `-1` si hubo algún error.

Ejemplo: Imprimir un determinado mensaje cada 5 segundos e ignorar `SIGINT`[4].

```
#include <signal.h>
#include <stdio.h>
#include <string.h>

void tratar_alarma(int sennal) {
    printf("Se activa la alarma\n\n");
}

int main ()
{
```

[2]La estructura `sigaction` incluye información adicional para señales de tiempo real, pero está fuera del ámbito de esta práctica y de la asignatura.
[3]Se recomienda al alumno que consulte el `man` para `sigaction()`.
[4]La señal SIGINT se genera con la combinación de teclas <CTRL><C>.

```
    struct sigaction act;
    /* Establecer manejador para SIGALRM */
    /* Funcion manejadora de la alarma   */
    act.sa_handler = tratar_alarma;

    act.sa_flags = 0;
    /* ninguna acción concreta */
    sigaction(SIGALRM, &act, NULL);

    /* ignorar SIGINT */
    act.sa_handler = SIG_IGN;

    /* no hay función manejadora. Se ignora SIGINT*/
    sigaction(SIGINT, &act, NULL);
    for (;;) {
        alarm (5);
        /* servicio para detener el proceso hasta que reciba una señal */
        pause();
    }
    return 0;
}
```

Compruebe el resultado de este programa.

8.3. Servicio POSIX `kill()`

Un proceso puede enviar señales a otros procesos o incluso grupos utilizando el servicio POSIX `kill()`.

`int kill(pid_t pid, int sig)`
Parámetros

pid Identifica al proceso al que se envía la señal (su PID).

sig Número de la señal que se envía. Sus posibles valores son iguales que en el servicio POSIX `signal()`.

Como es habitual en el resto de servicios POSIX, un valor de retorno 0 indica que `kill()` se ejecuta correctamente, mientras que un -1 indica que ocurrió un error durante su ejecución.

9. Intérprete de órdenes

El intérprete de órdenes es la puerta de entrada tradicional a UNIX. Comprender el funcionamiento interno del intérprete de órdenes ayuda a comprender muchos de los conceptos básicos de interacción con el sistema operativo, diferenciar bien los espacios de usuario y de sistema, así como algunos mecanismos de comunicación entre procesos.

9.1. Ciclo de ejecución del intérprete de órdenes

Conviene conocer la secuencia de acciones que realiza el intérprete de órdenes, que consiste básicamente en los siguientes pasos:

1. *Imprimir del prompt.* El intérprete se encuentra a la espera de que el usuario introduzca una orden.

2. *Procesar la orden (parser).* Se realizan un conjunto de transformaciones sobre la orden del usuario. Por ejemplo, si el usuario ha introducido la orden `cat practica1.c`, el intérprete transforma la cadena anterior en una estructura que pueda ser manejada más fácilmente en su ejecución (como ya veremos, consiste, esencialmente, en la transformación de la cadena en un array de vectores a las cadenas“ `cat`” y “`practica1.c`”).

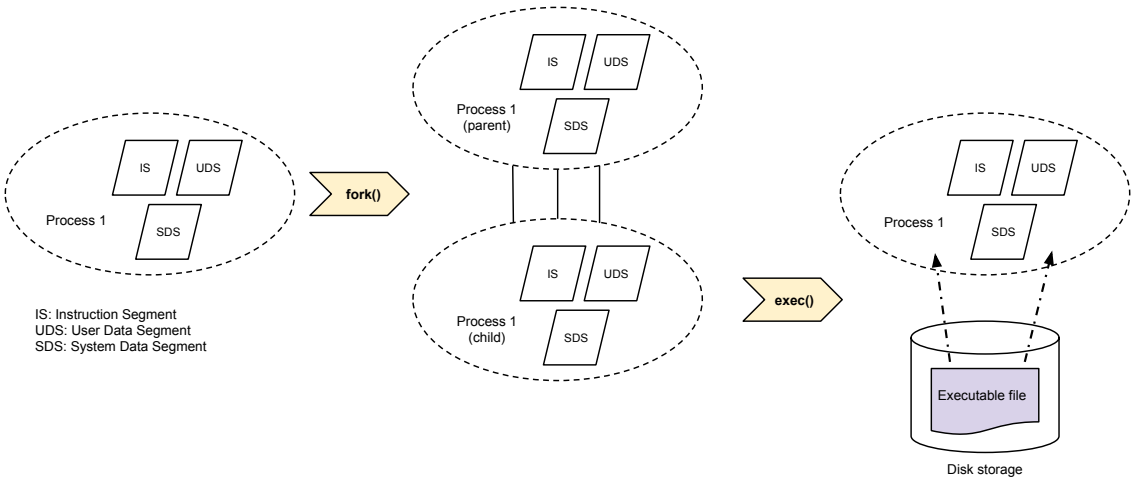

Figura 3: Ejecución de una orden externa dentro de un intérprete de órdenes

3. *Interpretar la orden.* Antes de ejecutar la orden, se realizan un conjunto de funcionalidades como sustituir variables de la shell por sus valores, generar nombres de archivos a partir de meta-caracteres que aparezcan en la orden, o manejar las redirecciones y tuberías. Por ejemplo, si el usuario ha introducido la orden `ls *.pdf`, y en el directorio de trabajo hay dos archivos, `examen.pdf` y `solucion.pdf`, el intérprete transforma `ls *.pdf` en `ls examen.pdf solucion.pdf`. Este es un ejemplo de generación de nombres de archivo a partir del meta-carácter '*' que realiza la *shell*.

4. *Ejecutar la orden.* Existen dos tipos de órdenes, y en función de cuál sea el tipo de orden introducido, el proceso de ejecución de la orden es distinta.

10. Ejercicio propuesto: la minishell

Como aplicación de los servicios POSIX de gestión de procesos descritos, debe completar el desarrollo, en lenguaje C y sobre el sistema operativo Linux, de una aplicación a la que denominaremos *minishell*. Esta aplicación se comportará como una versión reducida del intérprete de órdenes de UNIX.

Como cualquier proceso en UNIX, *minishell* utilizará por defecto la entrada estándar -teclado- para leer las líneas de órdenes a interpretar y ejecutarlas, y la salida estándar -pantalla- para presentar el resultado de las órdenes ejecutadas. Además, para notificar los errores que se puedan producir, usará por defecto la salida de error estándar -pantalla-.

Las tareas a realizar en el desarrollo de la *minishell* se dividen en seis pasos a realizar secuencialmente. Los pasos se evaluarán por separado, y dicha evaluación se realizará de acuerdo con la rúbrica publicada en la plataforma web de la asignatura.

La minishell tendrá las siguientes funcionalidades:

1. **Ejecución de órdenes simples externas en primer plano**. Por ejemplo, `ls -l`, `who`, etc. Para ejecutar una orden externa en primer plano debe crearse una *minishell* hija (`man fork`) que ejecute mediante el servicio `exec()` (`man execvp`) el archivo binario asociado a la orden. Mientras tanto, el proceso padre (*minishell* padre) debe esperar a la finalización del proceso hijo (`man wait`), si y sólo al final de la orden no existe el símbolo '`&`'.

2. **Ejecución de la orden interna `exit`**. Cuando la *minishell* reciba esta orden, finalizará su ejecución.

3. **Ejecución de órdenes en segundo plano o *background***. Cuando la *minishell* detecte el carácter `'&'` al final de una orden, no debe esperar a la finalización del proceso hijo creado para la ejecución de esa orden, sino que inmediatamente escribirá el *prompt* para aceptar una nueva orden. Si no se toman medidas adicionales, los procesos hijo, según vayan finalizando, no podrán comunicar a su padre su código de retorno, ya que el padre no estará haciendo `wait` sobre ellos. La consecuencia de esto es que lo procesos hijos quedarán en estado *zombi*.

4. **Gestión de procesos hijos *zombi***. Cuando el proceso hijo finaliza, el proceso padre (la *minishell*) debe de reaccionar correctamente para evitar que el hijo quede en estado *zombi*.

5. **Listado de procesos lanzados**. Cada proceso lanzado por la *minishell* debe quedar registrado. De esta forma, y mediante la orden interna `jobs`, se podrán mostrar todos los procesos que han sido lanzados y cuáles de estos han finalizado.

6. **Ejecución de secuencias de órdenes**. La *minishell* permite ejecutar secuencias de órdenes separadas por el símbolo ';'.

7. **Ejecución de dos órdenes comunicadas mediante tuberías**. La *minishell* será capaz de comunicar la salida estándar de una orden a la entrada estándar de otra, utilizando para ello una tubería. Por ejemplo `cat /etc/passwd | sort` .

8. **Ejecución de un número arbitrario de órdenes comunicadas mediante tuberías**. La *minishell* será capaz de ejecutar varias órdenes, de forma que cada una de ellas tomará como datos de entrada los datos de salida proporcionados por la orden anterior. Esto es por lo tanto la generalización a n procesos del apartado anterior.

> **El uso de `Makefiles` es obligatorio en todas las prácticas del laboratorio**
>
> Cada práctica de laboratorio debe incluir un `Makefile` que automatice el proceso de compilación. Este proceso no debe provocar mensajes de advertencia (*warnings*) y debe finalizar correctamente. El archivo `Makefile` debe así mismo incluir una regla `clean` que elimine todo los archivos que no sean estrictamente necesarios para la construcción del ejecutable final.

Para el desarrollo de este ejercicio utilice los archivos (archivos fuente, archivos de cabecera, y la biblioteca `libshell.a`) que se proporcionan como material de apoyo de esta práctica. Asimismo, realice los bloques de tareas que a continuación se detallan, de forma incremental y modular, siguiendo los pasos que se indican y probando poco a poco las funcionalidades que se vayan codificando.

10.1. PASO 1: la función `main()`

Entre los distintos archivos que componen el código fuente original de la práctica se encuentra `minishell.c`. Este archivo contiene la función principal `main()` que será el punto de entrada del programa. A lo largo del desarrollo de la práctica, el alumno irá completando, entre otras, esta función hasta alcanzar las funcionalidades requeridas por el enunciado. La versión inicial de `minishell.c` sólo contiene las instrucciones para obtener y analizar las órdenes introducidas por el operador. Es importante entender cómo se realizan estas dos primeras operaciones para poder abordar el resto de funcionalidades.

A continuación, se muestra el código inicial de `minishell.c`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
```

```c
#include <sys/wait.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>

#include "execute.h"
#include "parser.h"

void show_command(command * p_cmd);

int main (int argc, char *argv[])
{
    command* cmds;
    char* raw_command;
    int n_cmds=0;
    int n, i;

    while (1)
    {
        // We print the prompt
        print_prompt();

        raw_command = NULL;
        n = read_line (&raw_command, stdin);
        if (n==-1)
        {
            break;
        }

        n_cmds = parse_commands(&cmds, raw_command);

        for (i = 0; i < n_cmds; i++)
        {
            show_command(&cmds[i]);
        }

        free_commands(cmds, n_cmds);
        free(raw_command);
    }

    exit(0);
}

void show_command(command * p_cmd)
{
    int i;

    printf ("\tRaw command: \"%s\"\n", p_cmd->raw_command);
    printf ("\tNumber of arguments: %d\n", p_cmd->argc);

    for (i=0; i<=p_cmd->argc; i++)
        if (p_cmd->argv[i] != NULL)
            printf ("\t\targv[%d]: \"%s\"\n", i, p_cmd->argv[i]);
        else
            printf ("\t\targv[%d]: NULL\n", i);

    if (p_cmd->input)
        printf ("\tInput: \"%s\"\n", p_cmd->input);

    if (p_cmd->output)
        printf ("\tOutput: \"%s\"\n", p_cmd->output);

    if (p_cmd->output_err)
        printf ("\tError output: \"%s\"\n", p_cmd->output_err);

    printf ("\tExecute in the background: %s\n",
            p_cmd->background ? "Yes" : "No");
}
```

De forma cíclica, la `microshell` presenta un *prompt* al usuario, mediante la llamada a la función `print_prompt()`, invitándole a introducir una orden. La orden se lee desde la entrada estándar haciendo uso de la función `read_line()`. Esta función recibe un puntero por referencia, y sobre él se reserva de forma dinámica tanta memoria como sea necesario para contener la cadena tecleada por el usuario. Es responsabilidad del usuario de la función liberar dicha memoria haciendo uso de la función `free()`.

Una vez leída la orden desde teclado es necesario descomponerla. Por convenio, la primera palabra de la línea introducida se considera el nombre de la orden, y las consecutivas sus argumentos.

También es posible encadenar diferentes órdenes formando una tubería. Las tuberías se describen en detalle más adelante, por ahora sólo es necesario saber que el delimitador de tubería es el carácter | (*pipe*). Por último, también es posible indicar que se desea ejecutar la orden correspondiente en segundo plano (*background*) añadiendo el carácter `&` al final de la línea.

Para identificar qué parte de la línea es el nombre de la orden y cuál sus argumentos, si se han encadenado órdenes mediante tuberías o si se ha solicitado ejecución en segundo planto, es necesario analizar la línea obtenida con `read_line()` haciendo uso de la función `parse_commands()`. Esta función recibe un puntero por referencia sobre el que se reservará memoria para contener el resultado del análisis, y un puntero con la cadena de texto a analizar. Como resultado de la operación, la función devolverá el número total de órdenes, y dejará el puntero pasado como referencia apuntando al resultado del análisis.

El puntero pasado por referencia debe ser del tipo `command *`, que es un puntero a una estructura definida en `parser.h`. Así, por ejemplo, si invocamos la función así: `parse_commands (&cmds, ‘‘cat minishell.c’’)`, `n_cmds` sería 1, `cmd[0].argc` sería 2, `cmd[0].argv[0]` sería el nombre de la orden, es decir, `cat`, y `cmd[0].argv[1]` sus argumentos, en este caso `minishell.c`. Si se invoca esta función con una secuencia de órdenes enlazadas por tuberías, por ejemplo “`cat minishell.c | wc -l`”, entonces `n_cmds` valdría 2, siendo `cmd[0]` la primera orden y `cmd[1]` la segunda. Como se ha realizado reserva dinámica de memoria es necesario liberarla cuando ya no haga falta llamando a la función `free_commands()` tal y como se hace en el código entregado.

Para ilustrar el comportamiento del parser, la función `main()` incluye una función para mostrar el resultado del análisis de la cadena de texto introducida, que se llama `show_command()`. Estudie el código para comprender el resultado del parser y la forma en la que éste descompone la línea introducida por el usuario en cada caso.

10.2. PASO 2: orden interna `exit`

En este paso se dotará a la `microshell` de su primera orden: `exit`, que como su nombre indica, hará que finalice la `microshell`. Para hacer esto bastará con comprobar si la orden introducida ha sido `exit` haciendo uso de la función de comparación de cadenas de texto `strcmp()`. Para romper el bucle infinito en el caso oportuno puede hacer uso de la palabra reservada del lenguaje C `break`. No olvide liberar la memoria dinámica antes de finalizar la `microshell`.

10.3. PASO 3: ejecutar órdenes externas

El alumno deberá editar el archivo **execute.c**. Este archivo contendrá las funciones relacionadas directamente con la ejecución de una línea de órdenes. En este momento, el alumno **sólo tiene que implementar la ejecución de órdenes externas simples**.

> **Órdenes simples vs. órdenes compuestas (*pipes*)**
>
> Una línea de órdenes como la que se muestra a continuación es una orden compuesta:
>
> ```
> minishell> cut -f5 -d: /etc/passwd | sort > usuarios.txt &
> ```
>
> Esta línea contiene dos órdenes que se comunican mediante tuberías. Además, se almacena el resultado de la ejecución de la última orden en un archivo, tal y como se vio en la práctica 2.
> Sin embargo, los siguientes ejemplos muestran dos órdenes simples.
>
> ```
> minishell> cut -f5 -d: /etc/passwd
> minishell> sleep 30 &
> ```

El archivo **execute.c** debe contener como mínimo una función que se encargará de ejecutar la orden introducida por el usuario, y que tendrá el siguiente prototipo:

```
int execute_piped_command_line(command* cmds, int n)
```

Siendo `cmds` un puntero a un array de structuras de tipo `command`, y `n` el número de órdenes introducidas por el usuario. Es fácil ver que `cmds` es el resultado de haber ejecutado `parse_commands()` sobre la línea introducida por el usuario, tal y como se hizo en el paso anterior en `minishell.c`.

Modifique `minishell.c` eliminando el bucle *for* que iteraba por cada una de las órdenes mostrándolas (`show_command()`), y sustituya esa construcción por la llamada a `execute_piped_command_line()`. Como en este paso sólo ejecutaremos órdenes sencillas, `n` siempre valdrá 1 y por lo tanto `cmds` solo tendrá un elemento, `argv[0]`. Es una buena práctica verificar, en la medida de lo posible, que los argumentos con los que se llama a una función son coherentes. En este caso compruebe que `n` no sea ni inferior ni superior a 1. En el primer caso se puede interpretar como que no hay órdenes para ejecutar, por ejemplo porque el usuario introdujo una línea en blanco, así es que la función retornara 0 indicando que no hay error, pero tampoco hay nada que ejecutar. Si `n` es superior a 1 es que se ha introducido una orden que contiene tuberías. Por ahora no vamos a contemplar ese caso así es que la función debería mostrar un mensaje de error del tipo "funcionalidad aún no soportada" y retornar -1 indicando este hecho. Más adelante, cuando implementemos esta funcionalidad eliminaremos esta comprobación. Por último, si `n` vale 1, puede ser una buena idea utilizar de nuevo la función `show_command()` para mostrar la orden y sus detalles, retornando 0 a continuación. Evidentemente la función aun no hace lo que se espera de ella, pero es un buen momento para compilar y verificar que hasta este punto todo funciona correctamente.

El siguiente paso será hacer uso de las llamadas POSIX de la familia `exec` para ejecutar la orden solicitada por el usuario. Para esto es necesario crear un proceso hijo mediante `fork()` que se encargará de ejecutar (`exec`) la orden solicitada por el usuario. Mientras tanto el proceso padre puede optar por esperar a que el hijo termine (`wait()`) o continuar y solicitar una nueva orden al usuario. El primer comportamiento es el comportamiento por defecto. El segundo se denomina ejecución en segundo plano o background, y se solicita poniendo un & al final de la linea. La función `parse_commands()` detecta ese símbolo y pone a uno la variable `background` de la estructura `command` correspondiente.

Siguiendo con la metodología de desarrollo incremental, el siguiente paso aconsejable será crear el proceso hijo con la llamada `fork`. El padre guardara el PID del proceso hijo y le esperará haciendo uso de la función `wait()`. El hijo por su parte sacará un mensaje por la pantalla con `printf()` indicando que se está ejecutando, y acto seguido finalizará. Si se quiere simular el hecho de que el proceso hijo tardaría algo de tiempo en concluir su trabajo se puede poner una llamada a `sleep(3)` para que espere tres segundos antes de terminar, mostrando un mensaje adicional en ese momento. Es importante que el proceso hijo finalice explícitamente su ejecución haciendo una llamada a `exit(0)`, de lo contrario continuaría ejecutando el código que tuviera a continuación, y terminaríamos con una segunda `microshell` en ejecución. Compile y ejecute el código desarrollado hasta este punto.

Una vez que se crean procesos hijo correctamente, y que el padre espera a su finalización, lo siguiente será verificar si el usuario ha solicitado ejecución en segundo plano. Es ese caso, sencillamente, el proceso padre no esperará la finalización del hijo y la función retornará 0 regresando al bucle principal.

El siguiente paso incremental será sustituir el código de prueba que hemos puesto en el proceso hijo y sustituirlo por una llamada a `exec` que haga que el hijo ejecute el código de la orden que el usuario ha introducido. De todas las órdenes de la familia `exec`, para esta práctica aconsejamos utilizar `execvp()`, que es la que mejor se adapta a la estructura de datos apuntada por `cmds[0]`.

Llegados a este punto habríamos dotado a la *microshell* de la funcionalidad más importante, la de ejecutar las orden introducida por el usuario. No obstante, la variante de ejecución en segundo plano tiene un grave problema. Como el proceso padre no hace `wait()` para esperar al hijo, cuando éste termina no puede comunicar el resultado de la ejecución a su padre, y por lo tanto no puede terminar. Cuando los procesos quedan en este estado se dice que están *zombi*; son procesos que han concluido el código que tenían que ejecutar pero no pueden terminar porque antes tienen que comunicar a su padre el resultado de su ejecución. Resolveremos este problema en el siguiente paso.

10.4. PASO 4: gestión de procesos *zombi*

Para resolver es problema de los procesos *zombi* vamos a emplear un manejador de señal. Cuando un proceso hijo muere emite una señal que puede ser capturada por el proceso padre. Como respuesta a esta señal, el padre hará `waitpid()` para recoger el código de finalización del hijo, que podrá finalmente terminar.

Defina el manejador de la señal adecuadamente. Utilice para ello la función `sigaction()`, tal y como se ha descrito previamente. Si fuese necesario, puede usar el `man` para consultar la información relativa a dicho servicio POSIX.

Pruebe a realizar una secuencia de órdenes en *background* tal y como se muestra a continuación:

```
minishell> sleep 12 &
minishell> sleep 10 &
```

El resultado de ejecutar las órdenes previas debe ser el correcto. Una vez introducida cualquier orden en *background*, el *prompt* automáticamente debe volver a salir en pantalla para introducir otra orden. Si esto no sucede, consulte en el `man` el servicio POSIX que haya utilizado en el manejador de señal creado y la semántica asociada a sus parámetros para lograr una solución al problema.

10.5. PASO 5: listado de procesos lanzados

La *minishell* debe de guardar una estructura con todos los procesos lanzados durante su ejecución. Mediante la orden interna `jobs`, el usuario podrá listar dichos procesos. La lista indicará el PID asignado al proceso lanzado, así como el nombre del programa. Además, indicará si el proceso ha finalizado o si, por el contrario, sigue en ejecución. Para poder implementar esta funcionalidad, se proporcionan las siguientes funciones. Estas funciones ya se encuentran implementadas en archivo `jobs.c`, incluido dentro de la biblioteca `libshell.a`. La declaración de las funciones se encuentra en el archivo de cabecera `jobs.h`.

- **`void jobs_new(pid_t pid, const char * nombre)`**. La función registra en la estructura interna el lanzamiento de un nuevo proceso. La función recibe dos parámetros: el PID del proceso hijo y el nombre de la orden externa. Este nombre se corresponderá con el primer campo de la línea de órdenes.
- **`void jobs_finished(pid_t pid)`**. Esta función se emplea para indicar que el proceso identificador por el PID pasado como parámetro ha terminado.

Cuando un nuevo proceso se crea, la *minishell* padre debe registrarlo llamando a la función `jobs_new()`. Asimismo, cuando el proceso hijo finaliza, el padre debe indicar su fallecimiento empleando la función `jobs_finished()`. Esto último ha de realizarse tanto si el proceso hijo se estaba ejecutando en primer plano como si lo hacía en segundo plano. Para poder implementar correctamente este bloque, es necesario haber desarrollado con éxito el bloque principal 3.

Para poder comprobar el uso correcto de las funciones, se puede llamar a la función interna `jobs()`, que listará todos los procesos registrados hasta ese momento.

10.6. PASO 6: ejecución de órdenes secuenciales

Cuando la *minishell* detecte el símbolo ';', debe ejecutar de izquierda a derecha cada una de las órdenes. Resuelva este apartado declarando y definiendo su propia función e invocándola en donde considere adecuado para que realice correctamente su funcionalidad. Tenga en cuenta que es tan sencillo como invocar a la función `execute_piped_command_line()` tantas veces como órdenes haya separadas por el carácter ';'. Se recomienda también utilizar para este bloque funciones de manejo de cadenas de caracteres de C que pueden ser muy útiles (`strtok()`, `strsep()`, etc.).

Realice las pruebas necesarias para comprobar que ha realizado correctamente las tareas asociadas a los bloques.

10.7. PASO 7: ejecución de órdenes enlazadas mediante tuberías.

En este último paso vamos a considerar el caso en el que el usuario solicita la ejecución de una o más órdenes, comunicadas entre sí mediante tuberías, por ejemplo: `cat /etc/passwd | grep root | wc`. En esta orden `cat` leería el contenido del archivo `/etc/passwd` y lo escribiría en su salida estándar. Dicha salida estándar estaría conectada a la entrada estándar de `grep`, desde la que leería filtrando únicamente las líneas que contengan la secuencia `root`, escribiendo dichas líneas a su vez en su salida estándar. De la misma forma que antes, `wc` tendría conectada ahí su entrada estándar y leería de ella las líneas proporcionadas por `grep`, contándolas y sacando el resultado por su salida estándar. Esta última salida estándar fue la heredada de su padre, la *microshell*, que por lo general suele ser el terminal.

Para abordar esta parte es importante proceder también de forma incremental. A pesar de que el número de órdenes que se pueden enlazar mediante tuberías es potencialmente infinito, inicialmente vamos a considerar únicamente el caso en que haya dos procesos comunicados por una única tubería. Dichas órdenes serían respectivamente `cmds[0]` y `cmds[1]`, junto con sus argumentos.

La secuencia para llevar a cabo la comunicación sería la siguiente:

1. El proceso padre crea una tubería haciendo uso de la llamada `pipe()`.
2. El proceso padre crea el primer hijo, quien por lo tanto hereda la tubería creada por el padre.
3. Este primer hijo no escribirá en la salida estándar asignada por defecto, sino en el extremo de lectura del *pipe* que ha heredado. Para lograr esto, podemos usar cualquiera de las dos funciones `dup`. En caso de usar la función `dup()`, tendremos que cerrar primero su salida estándar haciendo `close(STDOUT_FILENO)`[5] para, acto seguido, hacer uso de la función `dup()` sobre el descriptor de archivo del extremo de escritura del *pipe*. Puesto que la función duplica el descriptor de archivo pasado como argumento usando el de numeración más baja disponible, que en nuestro caso es `STDOUT_FILENO`, ya que lo acabábamos de cerrar, conseguimos que dicho descriptor apunte al extremo de escritura del *pipe*. Si nos decantamos por usar la función `dup2()`, no es necesario cerrar de forma explícita la salida estándar, ya que es la propia función quién lo hace internamente. Por último cerraremos ambos extremos del *pipe*, ya que uno, el de lectura, no lo necesitamos, y el otro, el de escritura, lo acabamos de duplicar. Una vez configuradas la entrada y salida estándar, podemos proceder a la ejecución de `exec()` como en el caso ya implementado.
4. El proceso padre crea el segundo hijo, que hereda así mismo de su padre la tubería de comunicación.
5. Este segundo hijo configura su entrada y salida estándar de la misma forma que el anterior, solo que en este caso es la entrada estándar la que hay que apuntar al extremo de lectura del *pipe*. Una vez hecho esto se prosigue con la correspondiente llamada a `exec`.

En este caso, el comportamiento del padre es similar al del caso anterior. Si se ejecutan las órdenes en primer plano, esperará por el último de los procesos hijos, y en caso contrario continuará con su ejecución. En cualquier caso es muy importante que el padre, una vez haya creado los procesos hijos y que estos hayan heredado la tubería, cierre ambos descriptores de la tubería que él mismo acaba de crear. De lo contrario pueden darse casos de bloqueo.

Una vez conseguida la comunicación entre dos procesos, el siguiente paso será generalizar al caso en que haya un número arbitrario de procesos comunicados entre sí.

[5] Para poder usar las constantes `STDOUT_FILENO` y `STDOUT_FILENO` es necesario incluir el archivo de cabecera `unistd.h`