UNIVERSIDAD DE ALCALÁ

Departamento de Automática

Grado en Ingeniería Informática

Práctica 2: Herramientas de desarrollo y Servicios POSIX para la gestión de procesos

Sistemas Operativos

Índice

1. Competencias asociadas a la práctica

1. Distinguir las funcionalidades implementadas en una *shell* completa de Unix y las funcionalidades implementadas en el intérprete de órdenes de la práctica y lo que esto implica respecto a los resultados obtenidos al ejecutar diversas órdenes en ambos casos.
2. Identificar los servicios POSIX de gestión de procesos necesarios para la ejecución de órdenes concretas, internas y externas (en primer y segundo plano).
3. Programar el ciclo de ejecución del intérprete de órdenes.
4. Programar la ejecución de órdenes internas del intérprete de órdenes.
5. Programar la ejecución en primer plano de órdenes externas del intérprete de órdenes.
6. Programar la ejecución en segundo plano de órdenes externas del intérprete de órdenes.
7. Programar el uso de redirecciones de entrada y salida estándar ($>$, $<$) en el intérprete de órdenes.
8. Programar el uso de las tuberías sin nombre en el intérprete de órdenes.
9. Programar la secuencia de órdenes (órdenes separadas por ';') en el intérprete de órdenes.
10. Aplicar las herramientas de desarrollo clásicas en Unix: `gcc`, `make`, `gdb` y uso de bibliotecas estáticas.
11. Utilizar un estilo de programación (documentación incluida) correcto y uniforme en la programación del intérprete de órdenes.
12. Desarrollar un proyecto en equipo.

2. Introducción

En prácticas anteriores el alumno se ha iniciado 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 el uso, a alto nivel, de las llamadas al sistema (en concreto, servicios POSIX) para gestionar procesos y su comunicación, y en menor medida, llamadas al sistema, o más bien, servicios POSIX, para gestión de archivos, como preámbulo a su estudio exhaustivo en la asignatura de Sistemas Operativos Avanzados, de segundo curso de los grados de Ingeniería Informática e Ingeniería de Computadores.

Para lograr el objetivo previo, el alumno aplicará los conceptos vistos en las clases teóricas y en el laboratorio mediante la implementación parcial de un intérprete de órdenes en Unix (al que denominaremos *minishell*).

Otro objetivo principal de esta práctica es que el alumno sea capaz de usar las herramientas de desarrollo clásicas utilizadas en entornos Unix, tal y como se han descrito también en las clases teóricas. Actualmente existen entornos de desarrollo (o IDEs) muy avanzados, tales como *Eclipse*, *Netbeans*, etc., que abstraen al desarrollador de casi todos los detalles relacionados con la compilación del proyecto, de manera que su productividad aumenta al permitir que se concentre en la creación de código. Esta perspectiva, a pesar de ser la empleada comúnmente en entornos profesionales, presenta dos problemas: por una parte esa abstracción impide conocer qué sucede en el proceso de compilación a bajo nivel y, por otra parte, ese desconocimiento reduce la capacidad de resolver problemas que pudieran surgir.

Hablar de programación bajo Unix es hablar de C. Este lenguaje de programación 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 (Java, C++, C# o PHP, entre otros) han tomado elementos de C, o directamente son una evolución del mismo. Así pues, no es de sorprender que Unix y C tengan una relación más que estrecha, y por este motivo no se pueda abordar el estudio de Unix sin utilizar C.

Adicionalmente, un subobjetivo importante dentro de la práctica es acostumbrar al alumno a trabajar en equipo, una cualidad fundamental de cara al mundo laboral. Este hecho implica realizar un diseño adecuado de las aplicaciones, su modularización, así como una buena metodología de desarrollo y comunicación dentro del equipo[1].

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 de sobra 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`".
- *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 con un ejemplo 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 `vi`. En el laboratorio se recomienda el uso del editor `vi`, o, si se prefiere, un editor en modo gráfico.
- **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,

[1] En equipos de desarrollo profesionales, se suele utilizar la ayuda de sistemas de control de versiones como CVS o Subversion. Es un software imprescindible que permite que varias personas trabajen simultáneamente sobre un mismo código, manteniendo un registro de cambios.

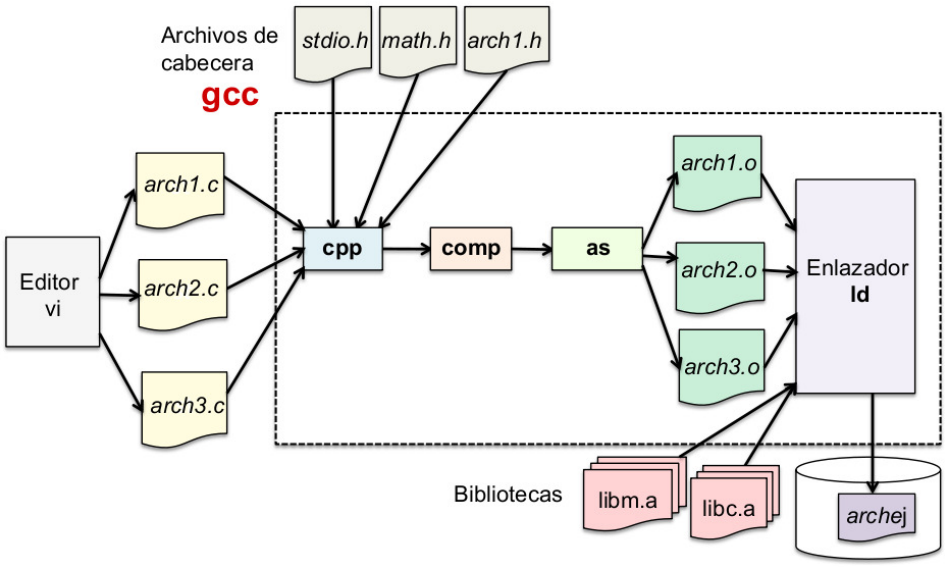


Figura 1: Ejemplo de generación de un archivo ejecutable en lenguaje C.

`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 `vi`, o versiones gráficas como `gvim`.

El editor `vi` 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`[2].

La forma de editar el programa será:

```
user@host:$ vi 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.

[2]El uso de `vi` al principio puede ser un poco frustrante por lo que, si el alumno no conoce el manejo de esta herramienta, es recomendable seguir alguno de los tutoriales existentes como, por ejemplo, el proporcionado en el Aula Virtual de la UAH, en concreto, en el espacio virtual reservado para la asignatura.

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 la siguiente:

```
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 (`.`). 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 una 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 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 *front-ends* 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` en la asignatura de Sistemas Operativos, en el Aula Virtual de la UAH.

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.

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, y es posible que necesitemos usar múltiples opciones en línea de llamada a `cc`. 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.

El programador novato, por lo general, es reacio a la utilización de esta herramienta, bien porque considera innecesario el uso de `make`, bien por pereza a la hora de crear el `Makefile`, bien por no dominar su uso. El empleo de `make` es requisito para todas las prácticas, y el alumno podrá comprobar de forma palpable la enorme ganancia de tiempo que conlleva su uso a cambio de unos minutos en la creación de un `Makefile` y el siempre necesario proceso de aprendizaje.

Puede encontrar numerosa documentación en Internet acerca del uso del `make`[3].

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.

[3] En la asignatura de Sistemas Operativos, en el Aula Virtual de la UAH, en la sección *Material complementario* de la práctica 3, también se han incluido algunos enlaces de interés para el estudio de la herramienta `make`.

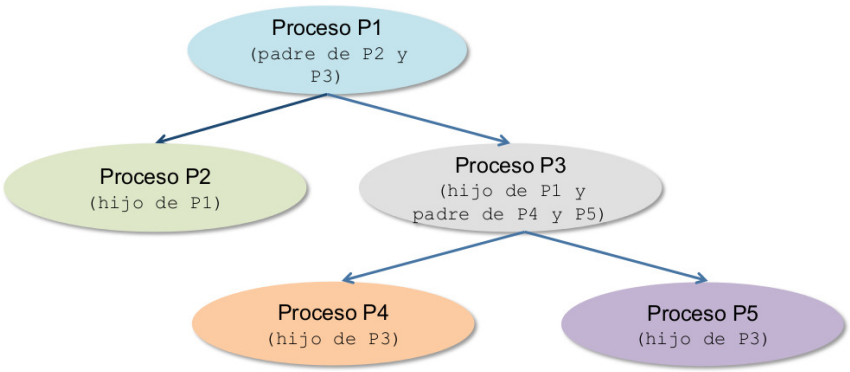


Figura 2: Jerarquía de procesos en UNIX.

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 es el proceso *init*, proceso lanzado por el SO durante el arranque del sistema responsable 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 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 para las personas. La forma más cómoda en 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 (nodo-i)*, 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 cubren 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* (interfaz) 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 -que incluye, la entrada del bloque de control de procesos correspondiente al 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. Si la función se ejecuta correctamente, el contenido del contexto de usuario del proceso que invoca a `exec()` deja de ser accesible y este contexto es reemplazado por el del nuevo programa. En estas condiciones, el programa antiguo es sustituido por el nuevo, y nunca se retornará al primero 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. Recuerde que, al abrir un archivo con el servicio POSIX `open()`, si la operación es válida, el sistema operativo devuelve un descriptor de archivo; un número entero correspondiente al índice de la entrada más baja libre de la tabla de descriptores de archivos (TDA) asociada al proceso. Estos descriptores identifican los archivos con los que puede trabajar el proceso. Por defecto, los tres primeros descriptores de archivo (0, 1 y 2) están asignados a la entrada estándar (por defecto, teclado), salida estándar (por defecto, pantalla) y salida estándar de errores (por defecto, pantalla) del proceso, respectivamente. Además, no olvide que los dispositivos son tratados como archivos en Linux y la pantalla está asociada a dos archivos distintos: salida estándar y salida estándar de errores[4].

La sintaxis de `exit()` 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.

La información de `status`, parámetro de `exit()`, podrá ser recuperada por el proceso padre a través del servicio POSIX `wait()`, que se describe a continuación.

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

`wait()` y `waitpid()` son dos servicios POSIX que esperan la finalización de un proceso hijo y permiten obtener información sobre su estado de terminación.

Un ejemplo de uso de estos servicios es cuando un usuario escribe una orden en el intérprete de órdenes de UNIX. El intérprete crea un proceso (*shell* hijo) que ejecuta la orden (el programa) correspondiente. Si la orden se ejecuta en primer plano (*foreground*), el padre esperará a que finalice la ejecución del *shell* hijo. 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 *zombie*; ha devuelto

[4] En la práctica 2 (“Material de apoyo") se explicaron las tablas involucradas en el acceso a los archivos en Linux.

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.

Si `wait()` se ejecuta correctamente, además retorna el PID (identificador) del proceso hijo cuya ejecución ha finalizado así como 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 éstos ya han terminado.

`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 `wait()` si el argumento `pid` es -1 y el argumento `status` es cero.

Parámetros

pid Si es `-1`, espera la finalización de cualquier proceso (como `wait()`). Si es `>0`, espera la finalización del proceso hijo con identificador `pid`. Si es 0, espera la finalización de cualquier proceso hijo cuyo identificador de grupo del proceso es igual que el del proceso que realiza la llamada (proceso padre). Si es `<-1`, espera la finalización de cualquier proceso hijo cuyo identificador de grupo del 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*[5].

8. Servicios POSIX para comunicación entre procesos

Los procesos no son entes aislados sino que, frecuentemente, es necesario que sean capaces de cooperar con otros procesos para lograr un objetivo común. Para facilitar esta tarea de colaboración, el sistema operativo ofrece mecanismos que permiten la transmisión de datos entre procesos (mecanismos de comunicación) y mecanismos que permiten a un proceso

[5]Sugerencia: tenga en mente esta opción para el desarrollo de la práctica.

esperar o continuar su ejecución (despertarse) de acuerdo con ciertos eventos (mecanismos de sincronización)[6]. Algunos de estos mecanismos pueden ser, a la vez, de comunicación y sincronización.

El sistema operativo Unix proporciona diversos mecanismos de comunicación y sincronización. Algunos de ellos son sólo de sincronización tales como señales o semáforos. Otros como tuberías, mensajes, memoria compartida, *sockets*, etc., son mecanismos de comunicación y sincronización. A continuación, se describen los servicios POSIX que implementan las operaciones básicas de dos de los mecanismos de comunicación y sincronización de Unix más utilizados; señales y tuberías.

8.1. Servicios POSIX de señales

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. Este mecanismo tiene gran utilidad de cara a afrontar situaciones en las cuales se producen eventos en instantes de 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 de las señales es muy similar al de las interrupciones pero la notificación del evento, a diferencia de las interrupciones, realizada por hardware (se activa una determinada entrada de la CPU), 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`[7]. 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.

8.1.1. Servicio POSIX `sigaction()`

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);
```

`sigaction()` 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). Este servicio POSIX permite tres opciones cuando llega una señal a un proceso:

1. Ignorar 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.

[6] En realidad la sincronización puede verse como un tipo de comunicación de información entre procesos.

[7] 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.

La estructura `sigaction` para señales estándar[8] es la siguiente:

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

donde:

sa_handler Es un puntero a la función manejadora 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. Sugerencia: consulte estas opciones accediendo a la descripción del servicio `sigaction()` con `man` .

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` (la señal `SIGINT` se genera con la combinación de teclas <CTRL> <C>.).

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

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


int main()
{
   struct sigaction act;

    /*establecer manejador para SIGALRM */
   act.sa_handler = tratar_alarma; /* función manejadora */
```

[8] La estructura `sigaction` incluye información adicional para señales de tiempo real, pero está fuera del ámbito de esta práctica y de la asignatura.

```
    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);
       pause(); /* servicio para detener el proceso hasta que reciba una señal */
    }
    return 0;
}
```

Compruebe el resultado de este programa.

8.1.2. 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 `sigaction()`.

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.

8.2. Servicios POSIX de *pipes*

Las tuberías (*pipes*) son el mecanismo de comunicación y sincronización entre procesos más antiguo de UNIX. Constituye una solución elegante para que el *shell*, tras crear varios procesos, permita que los datos de salida producidos por uno de estos procesos (por ejemplo, por la ejecución de una orden) se utilicen como entrada de datos de otro proceso creado.

El nombre de este mecanismo proviene de que, conceptualmente, se puede ver como una tubería o conducto real con dos extremos. Por uno de ellos, se escriben o se insertan datos y, por el otro, se leen o se extraen datos. El flujo de datos es *unidireccional* y con funcionamiento *First-In-First-Out* (FIFO), es decir, los datos se leen en el mismo orden en el que se escriben en la tubería, tal y como puede verse en la Figura 3.

Figura 3: Flujo de datos en una tubería.

El alumno ya se familiarizó en la práctica 2 con tuberías a nivel de intérprete de órdenes. Un ejemplo sencillo es el siguiente:

```
ls | wc -l
```

Para ejecutar la orden anterior, el *shell* crea dos procesos (con el servicio POSIX `fork()`). Cada uno de ellos ejecuta, mediante un servicio POSIX de la familia `exec()`, las órdenes `ls` y `wc -l`, respectivamente (los servicios POSIX `fork()` y `exec()` ya se han descrito en la sección 7).

Las tuberías o *pipes*, propiamente dichos, son un mecanismo de comunicación sin nombre. Por esta razón, sólo pueden usarse por el proceso que lo cree y sus procesos hijos, que heredan el *pipe*[9].

Desde el punto de vista de la implementación, una tubería es un *byte stream*, es decir, se pueden leer bloques de datos independiente del tamaño de los bloques escritos por el otro extremo de la tubería y el flujo es secuencial (los bytes se leen de la tubería en el mismo orden en que fueron escritos). Las tuberías normalmente se implementan como un buffer (generalmente circular) cuyo tamaño depende del sistema operativo (tIpicamente, de 4 KB). Se dice que es un *pseudoarchivo* mantenido en memoria por el sistema operativo.

La lectura y escritura de y en la tubería, respectivamente, se realiza con los mismos servicios POSIX que para leer/escribir de/en archivos. Estas APIs se verán más adelante en esta sección.

8.2.1. Servicio POSIX `pipe()`

Un *pipe* se identifica mediante dos descriptores de archivo en UNIX; uno para leer de la tubería y otro para escribir en ella. Cada proceso debe cerrar los descriptores que no utilice. El servicio POSIX que permite crear una tubería es `pipe()` cuya sintaxis es la siguiente:

```
int pipe(int filedes[2]);
```

Parámetros

filedes[ ] Si `pipe()` se realiza correctamente, la llamada devuelve en este array los descriptores de dos archivos abiertos que se utilizan como identificadores. `filedes[0]` se utiliza para leer de la tubería y `filedes[1]` para escribir en ella (véase Figura 4).

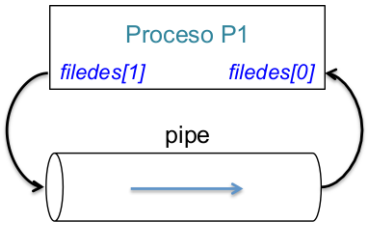


Figura 4: Creación de una tubería con `pipe()`.

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

8.2.2. Servicio POSIX `close()`

El servicio POSIX `close()` permite cerrar el descriptor de archivo asociado a una tubería.

Cerrar los descriptores de archivo de una tubería no utilizados por los procesos que manejen el *pipe* es fundamental para su correcto uso, principalmente por las siguientes razones:

[9]Existe otra variación de tuberías, las tuberías FIFO o tuberías con nombre, utilizadas para comunicar y sincronizar varios procesos independientes. Este tipo de tuberías no será objeto de estudio en la asignatura de Sistemas Operativos.

- El conjunto de descriptores de archivo abiertos es limitado. Si un descriptor no se usa, debe cerrarse para poder ser utilizado, si fuera necesario, por otro proceso.
- Si un proceso lee de la tubería, debe cerrar el descriptor de archivo para la escritura de tal modo que, cuando el otro proceso completa la escritura y cierra el descriptor asociado, el proceso lector detectará fin de archivo, tras haber leído los datos restantes en la tubería. En caso contrario, el proceso lector, al no cerrar el descriptor de escritura, no detectaría fin de archivo después de leer todos los datos de la tubería y una lectura posterior bloquearía la espera de datos dado que para el núcleo hay aún un descriptor de archivo de escritura abierto para la tubería.
- Si un proceso intenta escribir en una tubería para la que no hay ningún descriptor de archivo de lectura abierto, el núcleo del sistema operativo envía una señal `SIGPIPE` al proceso de escritura. Por defecto, esta señal mata al proceso que la recibe pero éste puede también establecer la captura o ignorar esta señal, en cuyo caso la escritura en la tubería falla con el error EPIPE (tubería rota). Recibir esa señal u obtener ese error es una indicación útil sobre el estado de la tubería por lo que se deben cerrar los descriptores de lectura no utilizados para la tubería. Además, si el proceso de escritura no cierra el extremo de lectura de la tubería, entonces, incluso después de que el otro proceso cierre el extremo de lectura de la tubería, el proceso de escritura seguirá siendo capaz de escribir en la tubería. Eventualmente, el proceso de escritura llenaría la tubería y un intento adicional de escritura bloquearía indefinidamente al proceso.
- Sólo después de que todos los descriptores de archivo en todos los procesos que utilicen la tubería se cierren, la tubería se destruye y sus recursos se liberan para su posible reutilización por otros procesos. En este punto, todos los datos no leídos de la tubería se pierden.

```
int close(int fd);
```

Parámetros

fd Indica el descriptor de archivo que se pretende cerrar.

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

En la Figura 5 se muestra el uso de una tubería creada con `pipe()` para comunicar dos procesos. El proceso hijo, creado mediante el servicio POSIX `fork()`, hereda una copia de los descriptores de archivo de su proceso padre; entre ellos, los descriptores de archivo de la tubería creada por él (ver Figura 5(a)).

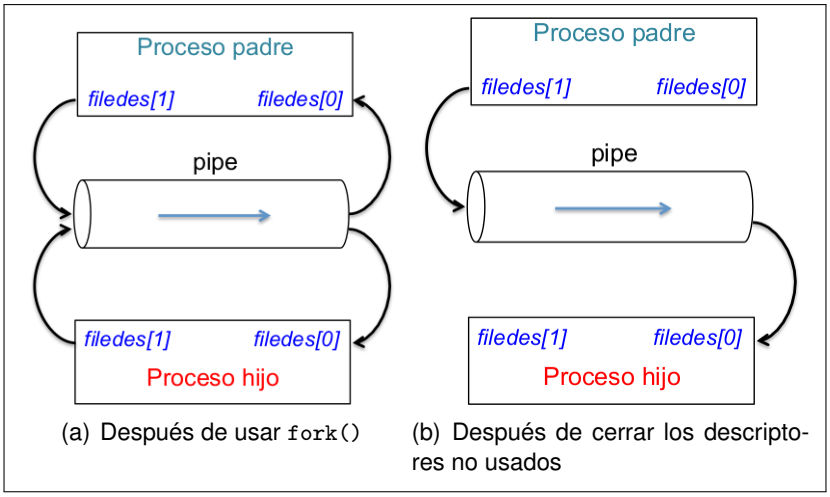
Como puede observarse en la Figura 5, inmediatamente después de usar `fork()`, el proceso padre cierra su descriptor de lectura de la tubería y el hijo cierra su descriptor de escritura en la tubería. En este caso, la comunicación consiste en que el padre envía datos al hijo (ver Figura 5(b)).

8.2.3. Servicio POSIX `read()`

El servicio POSIX `read()` se utiliza para leer datos de una tubería en el orden en el que fueron introducidos[10].

```
int read(int fd, char *buffer, int n);
```

[10] El servicio POSIX `read()` se utiliza también en UNIX para leer datos de un archivo.



(a) Después de usar `fork()` (b) Después de cerrar los descriptores no usados

Figura 5: Creación de un *pipe* para transferir datos de un proceso padre a un proceso hijo.

Parámetros

fd Descriptor de archivo utilizado para leer datos de la tubería.

buffer Buffer de memoria donde se pretende almacenar los datos leídos del *pipe* (la operación actualiza el puntero de lectura en la tubería).

n Número de bytes que se desean leer de la tubería.

El servicio POSIX `read()` devuelve el número de bytes leídos en caso de éxito y `-1` si hubo algún error.

El proceso que lee de una tubería puede leer bloques de cualquier tamaño, independientemente del tamaño de los bloques escritos por el proceso de escritura en la tubería.

A continuación, brevemente, se detalla la semántica de la operación de lectura de una tubería:

- Si el tamaño de la tubería es T y se desea leer de ella un número de bytes mayor, la operación devolverá T bytes. En caso contrario, la operación devuelve n bytes. En ambos escenarios, se eliminan los datos leídos de la tubería.
- Si se cierra el extremo final de escritura de la tubería y no hay procesos escritores, la operación de lectura de la tubería devolverá 0 bytes (final de archivo) una vez que haya leído los datos restantes de la tubería, pero no bloquea al proceso lector.
- La operación se lleva a cabo de forma **atómica**, es decir, si varios procesos intentan leer simultáneamente en una tubería, sólo uno de ellos podrá hacerlo y el resto de procesos se bloqueará hasta que finalice esa lectura. La atomicidad de esta operación se garantiza cuando el número de datos que se intentan leer es menor que el tamaño de la tubería.
- Si la tubería está vacía, la invocación a `read()` bloqueará al proceso que realiza la lectura hasta que otro proceso escriba datos en la tubería[11].

8.2.4. Servicio POSIX `write()`

El servicio POSIX `write()` se utiliza para escribir datos de forma ordenada en una tubería[12].

[11] Se puede evitar el bloqueo usando modo de lectura no bloqueante con el servicio POSIX `fcntl()`.

[12] El servicio POSIX `write()` también se utiliza en UNIX para escribir datos en un archivo.

```
int write(int fd, char *buffer, int n);
```

Parámetros

fd Descriptor de archivo que se utiliza para escribir datos en la tubería.

buffer Buffer de memoria donde se localizan los datos que se van a escribir en la tubería (la operación actualiza el puntero de escritura de la tubería).

n Número de bytes a escribir en la tubería.

El servicio POSIX `write()` devuelve el número de bytes escritos en caso de éxito y `-1` si hubo algún error.

A continuación, brevemente, se detalla la semántica de la operación de escritura en una tubería:

- Si no hay ningún proceso con la tubería abierta para lectura, el sistema operativo envía la señal `SIGPIPE` al proceso que pretende escribir. El proceso recibe la señal `SIGPIPE` y la operación de escritura (`write()`) devolverá un error.
- La operación se lleva a cabo de forma **atómica**, es decir, si varios procesos intentan escribir simultáneamente en una tubería, sólo uno de ellos podrá hacerlo y el resto de procesos se bloqueará hasta que finalice esa escritura. La atomicidad en la escritura se garantiza, al igual que para la lectura, cuando el número de datos que se intentan escribir es menor que el tamaño de la tubería.
- Si la tubería está llena o se llena durante la escritura actual, esta operación bloqueará al proceso escritor hasta que se pueda completar la operación.

8.2.5. Servicios POSIX para redirecciones: `dup()` y `dup2()`

Las redirecciones ya se vieron en la práctica 2 a nivel del intérprete de órdenes y qué operaciones y tablas están involucradas en espacio de usuario y espacio de núcleo. Del mismo modo, este concepto es clave en el caso de uso de órdenes comunicadas con tuberías como el ejemplo `ls | wc -l`. En este caso, se trata de dos redirecciones; en primer lugar, la salida estándar de la orden `ls` debe ser redirigida a la entrada de la tubería (extremo de escritura) y la entrada de la orden `wc -l` debe ser redirigida a la salida de la tubería (extremo de lectura). Para realizar estas dos redirecciones deben usarse los servicios POSIX `dup()` o `dup2()`. Ambos, se utilizan para duplicar un descriptor de archivo. Los prototipos de estas dos funciones son los siguientes:

```
int dup(int fd);
int dup2(int fd, int new_fd);
```

Parámetros

fd Descriptor de archivo cuya entrada en la tabla de descriptores de archivo (TDA) va a ser duplicada en otra (en concreto, en la entrada más baja libre).

new_fd Descriptor de archivo cerrado inicialmente por `dup2()` para, posteriormente, copiar en su entrada en la TDA la del primer parámetro, `fd`.

Los servicios POSIX `dup` y `dup2` devuelven el nuevo descriptor de archivo en caso de éxito y `-1` si hubo algún error.

El siguiente ejemplo muestra cómo puede usarse `dup()` o `dup2()` para redigir la salida estándar de un proceso.

```
/* uso de dup() */
int fd[2];
pipe(fd);

/* código adicional */

close(STDOUT_FILENO);
dup(fd[1])
```

Sin embargo, piense qué puede suceder si, con el fragmento de código anterior, utilizando `dup()` en vez de `dup2()`, la entrada estándar (`STDIN_FILENO`), con descriptor de archivo 0, ha sido cerrada antes de invocar a `dup()`. Para evitar errores, es conveniente usar `dup2()` como se muestra a continuación:

```
/* uso de dup2() */
int fd[2];
pipe(fd);

/* código adicional */

dup2(fd[1], STDOUT_FILENO)
```

A continuación, como ejemplo de manejo de los servicios POSIX de pipes, se muestra el código que implementa la ejecución de la tubería `ls | wc`[13]. Además, en la Figura 6 se muestran las principales operaciones realizadas en la implementación del uso de la tubería por las dos órdenes implicadas en este ejemplo.

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


int main (int argc, char *argv[])
{

    int pipefd[2];

    pipe(pipefd[2]); /* Creacion de la tuberia por el padre */

    switch(fork()) {
    case -1:
        perror("Error␣al␣crear␣el␣primer␣hijo");
        exit(1);

    case 0: /* Primer hijo ejecuta ls y escribe en tuberia */
        close(pipefd[0]);   /* Descriptor de lectura no usado */

        /* Duplicar entrada de descriptor de escritura de tuberia en la */
        /* de stdout y cierre de descriptor                              */

        if (pipefd[1] != STDOUT_FILENO) {
            dup2(pipefd[1], STDOUT_FILENO);
```

[13]En él, por brevedad, se han obviado las comprobaciones de error de la mayoría de servicios POSIX utilizados (`pipe()`, `close()`, `wait()`, etc.).

```
            close(pipefd[1]);
        }

        execlp("ls", "ls", (char *) NULL); /* Escribe en tuberia */
        perror("El primer hijo fallo en exec");
        exit(1);

    default: /* El padre pasa a crear otro hijo */
        break;
    }

    switch(fork()) {
    case -1:
        perror("Error al crear el segundo hijo");

    case 0: /* Segundo hijo ejecuta el filtro wc leyendo de la tuberia */
        close(pipefd[1]);   /* Descriptor de escritura no usado */

       /* Duplicar entrada de descriptor de lectura de tuberia en la */
       /* de stdin y cierre de descriptor                            */

        if (pipefd[0] != STDIN_FILENO) {
            dup2(pipefd[0], STDIN_FILENO);
            close(pipefd[0]);
        }

        execlp("wc", "wc", (char *) NULL); /* Lee de tuberia */
        perror("El segundo hijo fallo en exec");
        exit(1);

     default:
        break;
    }

    /* El padre cierra los descriptores no usados de la tuberia */
    close(pipefd[0]);
    close(pipefd[1]);

    /* Y espera a los procesos hijos */
    for (i=0; i < 2; i++)
        wait(NULL);

    exit(EXIT_SUCCESS);
}
```

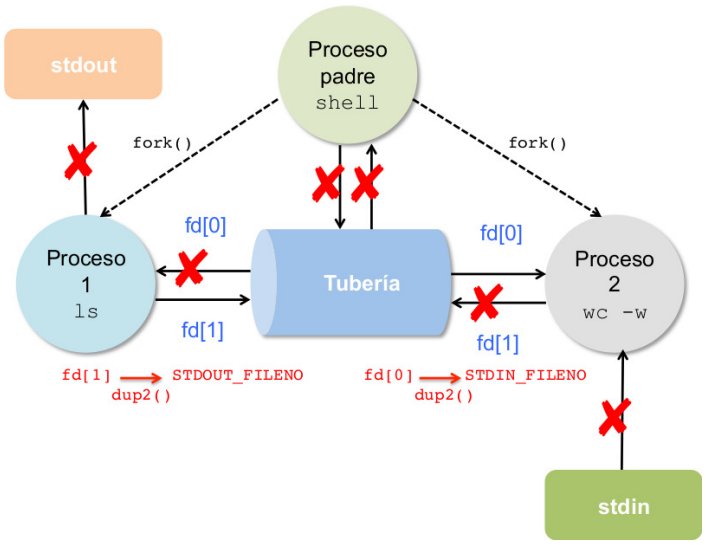


Figura 6: Esquema sobre la implementación de la orden `ls | wc -l`.

Compruebe el resultado de este programa.

9. Intérprete de órdenes

El intérprete de órdenes es la puerta de entrada tradicional a UNIX. Comprender su funcionamiento interno 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 `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`").

3. *Interpretar la orden*. Antes de ejecutar la orden, se realizan un conjunto de funcionalidades como sustituir variables del *shell* por sus valores, generar nombres de archivos a partir de metacaracteres 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 metacarácter '*' que realiza el *shell*.

4. *Ejecutar la orden*. En función de cuál sea el tipo de orden introducido, el proceso de ejecución de la orden es bien distinto. El *shell* verifica si es una *orden interna*. Si lo es, ejecuta su código, perteneciente (interno) al propio código del intérprete de órdenes. En caso contrario, se trata de un programa ejecutable de UNIX (independiente del *shell*); si es una *orden externa*, el intérprete busca su imagen binaria (ejecutable) para solicitar su ejecución al Sistema Operativo[14].

10. Ejercicio

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

Como cualquier proceso en UNIX, *minishell* utilizará por defecto la entrada estándar -teclado- (cuyo descriptor de archivo es 0) para leer las líneas de órdenes a interpretar y ejecutarlas, y la salida estándar -pantalla- (cuyo descriptor de archivo es 1) para presentar el resultado de las órdenes ejecutadas. Y, para notificar los errores que se puedan producir, usará por defecto la salida estándar de errores -pantalla- (cuyo descriptor de archivo es 2).

Las tareas a realizar en el desarrollo de la *minishell* son las siguientes:

1. Implementar las funcionalidades siguientes en *minishell*:

[14] La petición de ejecución se realiza mediante el uso conjunto de los servicios POSIX `fork()` y `exec()`, respectivamente.

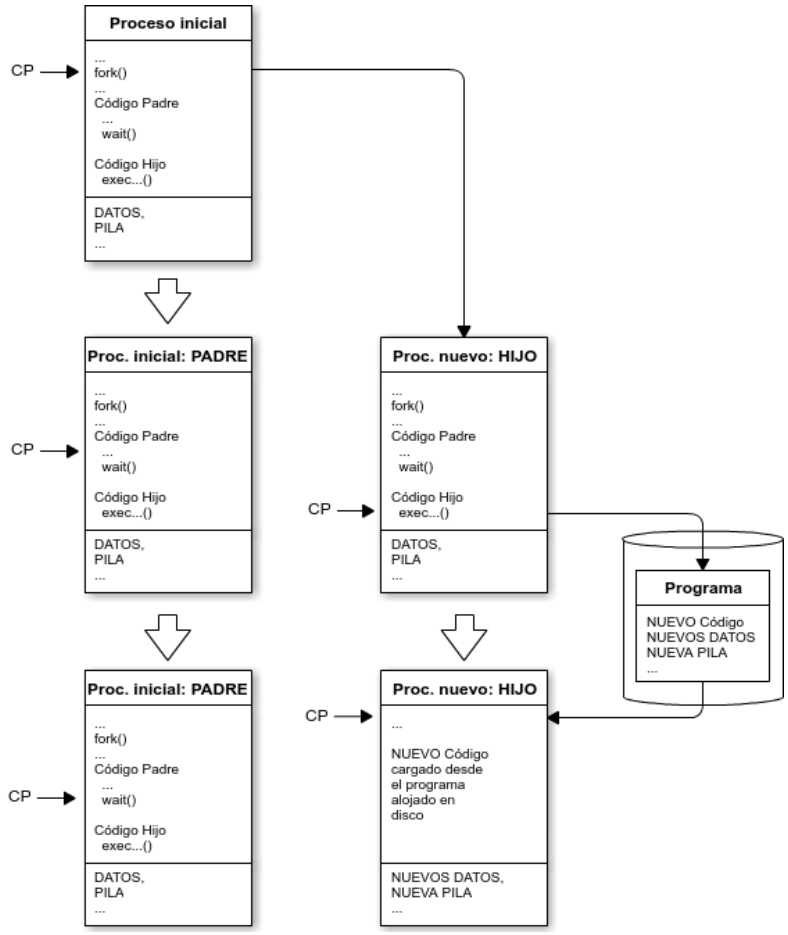


Figura 7: Ejecución de una orden externa dentro de un intérprete de órdenes.

- Ejecución de órdenes simples externas en primer plano (por ejemplo, `ls -l`, `who`, etc.). Para ejecutar una orden externa en primer plano debe crearse un proceso *minishell* hijo (veáse información con: `man 2 fork`) que ejecute mediante el servicio `exec()` (véase información con: `man 2 execvp`) el archivo binario asociado a la orden. Mientras tanto, el proceso padre (*minishell* padre) debe esperar a la finalización del proceso hijo (véase información con: `man 2 wait`). Al final de la orden **no** existe el símbolo `'&'`.
- Ejecución de la orden interna `exit`. Cuando la aplicación *minishell* reciba esta orden, finalizará su ejecución.
- Ejecución de órdenes internas. Sólo se podrán ejecutar como órdenes internas, además de `exit`, `umask`, `cd`, `pwd` y `declare`[15]
- Ejecución de órdenes en *background* (usando el carácter `'&'` al final de las órdenes). Cuando la *minishell* detecte este símbolo al final de una orden, no debe esperar a la terminación del proceso hijo creado para ejecutarla sino que, inmediatamente, imprimirá el *prompt* para aceptar una nueva orden.
- Ejecución de secuencias de órdenes separadas por el símbolo `';'` (**como mínimo**, secuencias de órdenes externas).
- Ejecución de órdenes con redirecciones de entrada (<) y salida (>).
- Ejecución de órdenes externas compuestas (separadas mediante tuberías).

[15]La implementación de estas órdenes, como se reiterará más adelante, se proporciona al alumno en el archivo objeto `internas.o`.

2. Crear un archivo *Makefile* que permita la compilación automatizada de la aplicación.

3. Realizar las pruebas adecuadas de *minishell*.

Para el desarrollo de este ejercicio utilice los archivos fuente y archivos de cabecera que se proporcionan como material de apoyo con la práctica. Realice con ellos las fases que a continuación se detallan, de forma incremental y modular, siguiendo los pasos que se indican en cada una de ellas y probando poco a poco las funcionalidades que se vayan codificando.

Diagrama de la estructura de *minishell*

En el Anexo A se adjunta un diagrama de la estructura detallada de la aplicación *minishell*. El esquema muestra los diferentes archivos, la relación existente entre ellos y las funciones que incluyen distinguiendo si están ya implementadas o no. Se recomienda al alumno tenerlo a mano en todo momento para tener una visión global de toda la aplicación y poder ubicar cada función en el conjunto.

A continuación, realice las fases que se van describiendo en las siguientes secciones para el desarrollo de *minishell*.

10.1. FASE 1: Ciclo de ejecución de órdenes

Edite el archivo fuente denominado **minishell.c**. Complete este código definiendo la función principal (`main()`) encargada de realizar, de forma reducida, acorde exclusivamente a las funciones que se van a incluir en esta práctica, el ciclo de ejecución de órdenes del *minishell* (véase sección 9.1). Para ello, siga el diagrama de la Figura 8.

Código 1: `minishell.c` incompleto

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/wait.h>
4  #include <fcntl.h>
5  #include <signal.h>
6  #include <unistd.h>
7  #include <string.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include <errno.h>
11
12 #include "internas.h"
13 #include "entrada_minishell.h"
14 #include "ejecutar.h"
15
16
17 int main (int argc, char *argv[])
18 {
19
20    char buf[BUFSIZ];
21
22
23    while (1)
24    {
25
26
27
28
29
30
31
32
33
```

```
34
35
36
37
38      }
39
40      return 0;
41  }
```

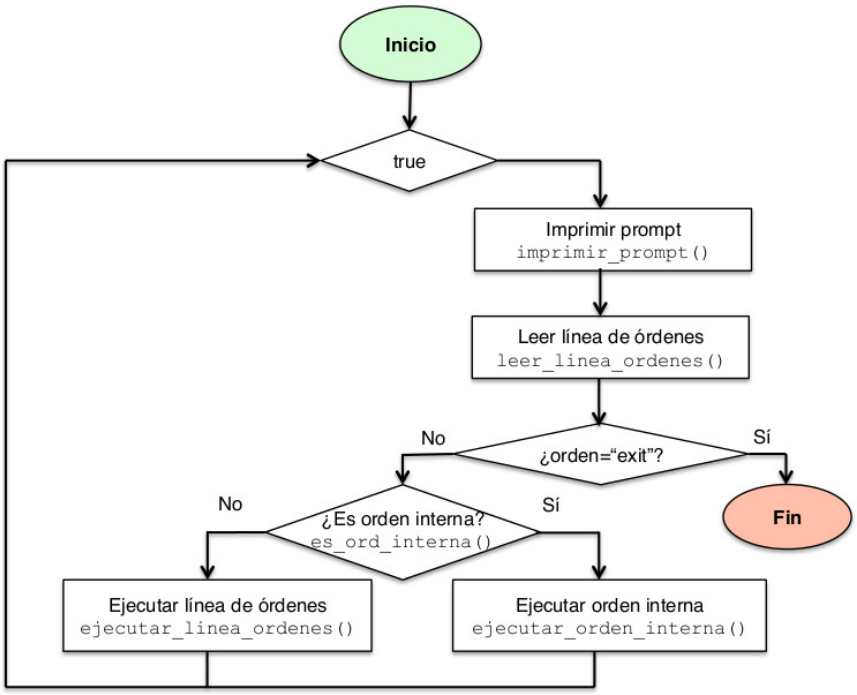


Figura 8: Diagrama de la función `main()` de *minishell*.

Posponga el tratamiento de una orden externa (invocación a `ejecutar_linea_ordenes()` de la Figura 8) hasta la realización de la **FASE 2** de la práctica (se trata de ir resolviendo y probando poco a poco las tareas que se solicitan).

Como puede observar en la Figura 8, para implementar la función `main()` se utilizan sendas funciones: `imprimir_prompt()`, `leer_linea_ordenes()`, `es_ord_interna()` y `ejecutar_ord_interna()`. El código de estas funciones se proporciona como material de apoyo. Sólo debe invocarlas adecuadamente teniendo en cuenta en qué archivos se encuentran tanto la definición como el prototipo de cada una de ellas. A continuación, se proporciona también una descripción de estas funciones[16]:

- `void imprimir_prompt()`. Imprime el prompt de la *minishell*.
- `void leer_linea_ordenes(char *buf)`. Lee la línea de órdenes de la entrada estándar y la almacena como una cadena de caracteres en `buf`.
- `int es_ord_interna(const char *buf)`. Devuelve si la orden que se pasa como argumento es interna (valor 1) o externa (valor 0). Las **únicas** órdenes internas implementadas son: `cd`, `pwd`, `declare` y `umask`.

[16] Estas funciones se han codificado de forma sencilla para que el alumno las pueda interpretar fácilmente y vaya familiarizándose con el lenguaje C. Puede modificar y simplificar las funciones relacionadas con entrada/salida de C. La correcta adaptación de su codificación será valorada en la nota final de la práctica.

- `ejecutar_ord_interna(const char *buf)`. Si la orden que se pasa como argumento es interna, la función ejecuta la orden.
  Las dos primeras funciones se encuentran en el archivo `entrada_minishell.c` proporcionado como material de apoyo. Las dos últimas, relacionadas con órdenes internas, se proporcionan en el archivo en código objeto `internas.o`.

Consulte el diagrama de la aplicación *minishell* (Apéndice A) para comprobar los archivos que se proporcionan como material de apoyo. De momento, se recomienda no crear la biblioteca estática `libshell.a`; ya lo hará más tarde. Para probar esta fase sólo precisa usar los archivos fuente y de cabecera requeridos así como el archivo `internas.o` en la compilación.

Desarrollo incremental y pruebas

Se recomienda probar de forma incremental las diferentes funcionalidades programadas en todas las fases. Antes de entregar la práctica, realice la validación con las pruebas adecuadas.

10.2. FASE 2: Ejecución de órdenes externas simples en primer plano

Una vez realizadas las pruebas de la ejecución de órdenes internas, realice en esta fase la implementación de órdenes externas simples y en primer plano como las que a continuación se muestran en sendos ejemplos:

```
minishell> cut -f5 -d: /etc/passwd
minishell> sleep 30
```

Sin embargo, una línea de órdenes como la siguiente es una orden compuesta, es decir, formada por varias órdenes comunicadas mediante tuberías, y ejecutada en *background*. De la implementación de órdenes compuestas no debe preocuparse ahora:

```
minishell> cut -f5 -d: /etc/passwd | sort > usuarios.txt &
```

Para realizar la tarea de esta fase, edite el archivo **ejecutar.c** proporcionado como material de apoyo. Este archivo contiene, incompletas, las funciones relacionadas directamente con la ejecución de una línea de órdenes externa. Finalice la implementación del código de `ejecutar.c` que se muestra a continuación.

Código 2: `ejecutar.c` incompleto

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

#include "parser.h"
#include "ejecutar.h"
#include "libmemoria.h"


pid_t ejecutar_orden (const char *orden, int *pbackgr)
{
   char **args;
   pid_t pid;
   int indice_ent = -1, indice_sal = -1; /* por defecto, no hay < ni > */

   if ((args = parser_orden(orden, &indice_ent, &indice_sal, pbackgr)) == NULL)
   {
      return(-1);
   }
```

```
    /* si la linea de ordenes posee tuberias o redirecciones, podra incluir */
    /* aqui, en otras fases de la practica, el codigo para su tratamiento.  */









}

void ejecutar_linea_ordenes(const char *orden)
{
    pid_t pid;
    int backgr;


    /* Si la orden es compuesta, podra incluir aqui, en otra fase de la */
    /* practica, el codigo para su tratamiento                          */


    pid = ejecutar_orden(orden, &backgr);




}
```

- `void ejecutar_linea_ordenes(const char *orden)`. Esta función se encarga de ejecutar una orden externa introducida por el usuario. Como ya se ha indicado, **sólo debe implementar en esta fase la ejecución de órdenes externas simples**. Para ello, la función invoca a `ejecutar_orden()`, función cuya descripción se proporciona más adelante.

  Tras la llamada a `ejecutar_orden()`, `ejecutar_linea_ordenes()` debe comprobar la existencia o no del símbolo de *background* al final de la orden, información que es devuelta por `ejecutar_orden()`. En función de este chequeo, `ejecutar_linea_ordenes()` proporcionará el siguiente tratamiento:

  - **Si no existe el símbolo '&'**. Se trata de una ejecución en primer plano (*foreground*). El proceso padre (*minishell* padre) **debe esperar** (invocando al servicio POSIX adecuado) a que el proceso hijo (*minishell* hija) finalice su ejecución. Cuando esto ocurra, el proceso *minishell* padre podrá realizar otra iteración del ciclo de la función `main()`. Por lo tanto, imprimirá el *prompt* y volverá a leer y ejecutar otra posible orden introducida por el usuario.
  - **Si la orden termina con '&'**. Se trata de una ejecución en segundo plano (*background*). El proceso *minishell* padre simplemente no debe esperar la finalización del proceso *minishell* hijo y podrá volver a ejecutar **inmediatamente** otra iteración del bucle de la función `main()`. Esto significa que el proceso *minishell* padre y el proceso *minishell* hijo podrán ejecutar concurrentemente órdenes.

- `void ejecutar_orden(const char *orden, int *background)`. Función que crea un proceso *minishell* hijo (invocando al servicio POSIX adecuado) para ejecutar la orden externa pasada como parámetro (`orden`). **La función devuelve en** `background` **el valor 1 si hay un símbolo de *background* ('&') al final de la orden**, o 0 en caso contrario. Como puede observar en el código proporcionado previamente, la función `ejecutar_orden()` primero invoca a la función `parser_orden()` cuya descripción y prototipo se detallan más adelante. Esta última función es responsable de convertir la cadena `orden`, que almacena la orden externa introducida por el usuario, en una estructura que facilite su ejecución con

el servicio POSIX correspondiente. La función `parser_orden()` ya está implementada y se incluye en el archivo objeto `parser.o`.

Antes de finalizar la definición de la función `ejecutar_orden()`, deberá invocar a `free_argumentos()` ya que la función `parser_orden()` crea un array dinámico que debe ser liberado para evitar lagunas de memoria. La función `free_argumentos()`, cuyo prototipo se muestra a continuación, ya está implementada en el archivo `libmemoria.c`.

`void free_argumentos(char **argumentos)`. Esta función libera la estructura `argumentos` de tipo array dinámico.

Uso de la función `ejecutar_linea_ordenes()`

La existencia de `ejecutar_linea_ordenes()`, además de la función de `ejecutar_orden()`, pretende ofrecer una descomposición más modular y flexible de *minishell* para dar así la posibilidad de ampliar más adelante en ella la funcionalidad solicitada en la Fase 7 (tratamiento de órdenes compuestas). El alumno puede obviarla y realizar sus tareas, indicadas previamente, en la propia función `ejecutar_orden()`, u otra.

`parser_orden()`:

Descripción

Función que convierte la cadena que representa la orden introducida al ejecutar *minishell*, en una nueva estructura: un array de punteros a cadenas. La finalidad es dejar preparada la orden para ser ejecutada adecuadamente con el correspondiente servicio POSIX. Más en detalle, `parser_orden` analiza la cadena `orden` y genera la nueva estructura que consta de: a) Una cadena por cada uno de los argumentos de `orden`, eliminando los posibles blancos entre ellos (considerando también el nombre de la orden como argumento) y, b) si encuentra redirecciones, añade dos cadenas: una con el carácter de la redirección y otra, con la ruta del archivo asociado a ella. Asimismo, si en `orden` hay un símbolo `'&'` (*background*), la función lo elimina (no lo incluye en la estructura creada) pero devuelve un indicador de su existencia.

¿Por qué es necesario eliminar el '&'?

Si no se elimina el símbolo '&' en la estructura devuelta por la función `parser_orden()` tendrá problemas para que el proceso *minishell* hijo ejecute correctamente la orden correspondiente ¿Por qué?

La función también devuelve, en sendos parámetros, la posición en la nueva estructura de los caracteres de redirección que puedan existir, o -1 en caso de que no existan.

Prototipo

```
char ** parser_orden(const char *orden, int *indentrada,
                     int *indsalida, int *background)
```

Parámetros

orden Contiene la orden introducida por el usuario al ejecutar *minishell*. Esta orden va a ser analizada y convertida por la función en un array dinámico de punteros a cadenas.

indentrada Puntero a un entero que representa el índice en el array devuelto por la función donde se localiza un posible símbolo de redirección de entrada en la orden. Si no existe este tipo de redirección, devuelve un puntero a -1.

indsalida Puntero a un entero que representa el índice en el array devuelto por la función donde se localiza un posible símbolo de redirección de salida en la orden. Si no existe este tipo de redirección, devuelve un puntero a -1.

background Puntero a un entero que representa la existencia (1), o no (0), del símbolo de *background* (`'&'`).

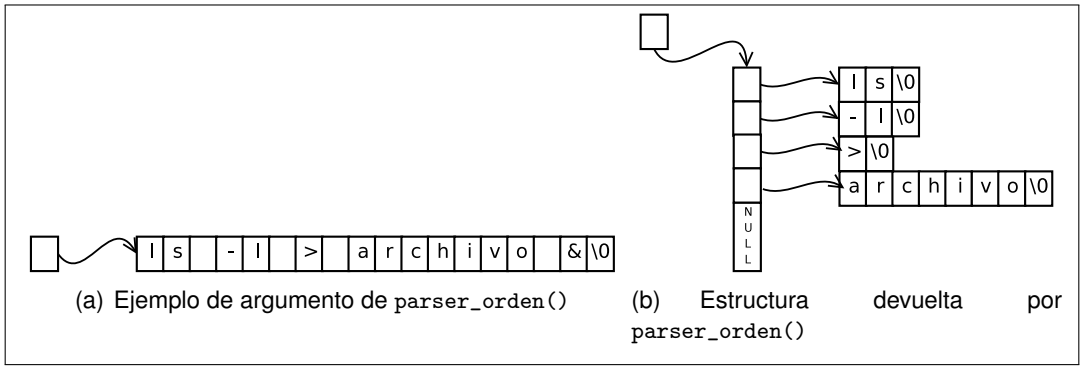
Valores retornados

Array dinámico de punteros a cadenas descrito anteriormente. Además, la función devuelve en `background` la existencia (1) o no (0) del símbolo *background* eliminándolo del array devuelto, así como el índice en el array donde se encuentra una redirección de entrada, en `indentrada`, o una redirección de salida, en `indsalida`. En ambos casos, si no existe redirección, devuelve -1 en el parámetro correspondiente.

Ejemplo: Si la orden, de tipo cadena constante, introducida por el usuario, y argumento de la función `parser_orden()`, es la siguiente: `orden = ‘‘ls -l > archivo &’’` (véase Figura 9(a)), los valores finales de cada parámetro son los que a continuación se detallan:

```
indsalida = 2
indentrada = -1
background = 1
```

Además, la función devuelve un array (de tipo puntero a puntero a carácter) que se representa en la Figura 9(b).



(a) Ejemplo de argumento de `parser_orden()` (b) Estructura devuelta por `parser_orden()`

Figura 9: Estructuras de datos utilizadas por `parser_orden()`.

Versatilidad de la función `parser_orden()`

En esta fase no se tratan redirecciones pero la interfaz así definida de `parser_orden()` le permitirá más adelante incrementar fácilmente la funcionalidad de su *minishell* programando, tras la invocación a `parser_orden()`, y de acuerdo con los valores devueltos a través de los últimos parámetros mencionados (`indentrada` e `indsalida`), el tratamiento de las redirecciones (Fase 6, sección 10.6).

10.3. FASE 3: Ejecución de órdenes externas simples en segundo plano

Compruebe qué sucede cuando se introduce una orden en *background* con el código realizado hasta el momento. Para ello, consulte el estado de los procesos del sistema con la orden apropiada.

Observe que el tratamiento del *background* hasta esta fase puede dejar al proceso *minishell* hijo en estado *zombie* que, como se ha visto en en el aula, es un estado de los procesos en Linux no deseado. Razone por qué sucede esto y solucione este problema utilizando un manejador de la señal adecuada. Piense qué señal se podría utilizar para que, una vez que el proceso *minishell* padre la capture, el manejador se encargue de evitar que el proceso *minishell* hijo quede en estado *zombie* y dónde se debería instalar el manejador.

Una vez que tenga clara la solución al problema planteado, defina el manejador de la señal adecuadamente, de forma similar al ejemplo de manejo de la señal `SIGINT` (véase sección 8.1.1). Utilice para ello, ESTRICTAMENTE, la función `sigaction()`, **no** `signal()`. Asimismo, se recomienda usar la orden `man` para consultar el servicio POSIX `sigaction()`, tanto su descripción como la semántica asociada a sus parámetros.

Finalmente, tras codificar esta fase, 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 y, además, en este escenario, **no debe haber procesos *zombies* en el sistema** (no deje de probar esto último; piense con qué orden puede verificarlo).

10.4. FASE 4: Realización de *Makefile*

Realice el *Makefile* que permita construir una nueva versión de la aplicación.

Cree la biblioteca `libshell.a`, tal y como aparece en el diagrama de la aplicación *minishell* (Anexo A). Para ello, defina una regla específica en el *Makefile* que genere esta biblioteca estática con los archivos `internas.o` y `parser.o`, respectivamente, utilizando la herramienta `ar`[17]. `libshell.a` debe ser utilizada como parámetro en la invocación a `gcc` para que los archivos objeto junto con la biblioteca se enlacen en la última fase de compilación y se genere el archivo ejecutable *minishell*.

Se puntuará con 0 puntos todo archivo *Makefile* que no incluya: a) definición adecuada y explícita de todas las dependencias entre los distintos archivos que componen la aplicación para generar el ejecutable *minishell*, b) regla de creación de la biblioteca estática y/o, c) regla ficticia *clean*.

[17] Consulte qué parámetros debe pasarle a la orden `ar` para crear la librería con el nombre de la biblioteca y el de los archivos objetos. Sugerencia: use `man`, busque en Internet o pregunte al profesor(a).

10.5. FASE 5: Ejecución de secuencia de órdenes

En esta fase se plantea poder ejecutar órdenes separadas por el carácter ';' (**como mínimo, secuencia de órdenes externas**)[18]. Cuando la aplicación *minishell* detecte este símbolo, debe ejecutar de izquierda a derecha cada una de las órdenes. Resuelva este apartado declarando y definiendo su propia función e invocándola donde considere adecuado para que realice correctamente su funcionalidad. Piense que, para poder ejecutar secuencia **sólo** de órdenes externas, se trata de invocar a la función `ejecutar_linea_ordenes()` tantas veces como órdenes haya separadas por el carácter ';'.

Uso de funciones de manejo de cadenas de carateres de C

Se recomienda utilizar para esta fase funciones de manejo de cadenas de caracteres de C que pueden ser muy útiles (`strsep()`, `strtok()`, etc.).

10.6. FASE 6: Tratamiento de redirecciones

En esta fase debe resolver la posibilidad de ejecutar en la aplicación *minishell* órdenes externas con redirecciones de entrada o de salida. A continuación, se presenta un ejemplo con cada tipo de redirección y, finalmente, uno con ambos tipos en la misma orden:

```
minishell> cut -f 5 -d : /etc/passwd > usuarios.tex
minishell> wc -l < /etc/passwd
minishell> sort < /etc/passwd > ordenado_passwd &
```

Para resolver esta parte, edite el archivo **redirecciones.c** proporcionado como material de apoyo y que se muestra a continuación. Complete las funciones relacionadas directamente con el tratamiento de redirecciones.

Código 3: `redirecciones.c` incompleto

```
1  #include <stdio.h>
2  #include <errno.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <stdlib.h>
6  #include <fcntl.h>
7
8  /* funcion que abre el archivo situado en la posicion indice_entrada+1 */
9  /* de la orden args y elimina de ella la redireccion completa          */
10
11 void redirec_entrada(char **args, int indice_entrada, int *entrada)
12 {
13
14
15
16
17
18
19
20 }
21
22 /* funcion que abre el archivo situado en la posicion indice_salida+1 */
23 /* de la orden args y elimina de ella la redireccion completa         */
24
25 void redirec_salida(char **args, int indice_salida, int *salida)
26 {
```

[18] Extender el problema a secuencia de órdenes tanto externas como internas no es complejo; ¡si le queda tiempo, inténtelo!. Se valorará en la nota final de la práctica cualquier tarea adicional realizada.

```
27
28
29
30
31
32
33 }
```

Observe que la función `parser_orden()` que se le ha proporcionado, descrita en la Fase 2 (sección 10.2), devuelve en el parámetro `indice_entrada` la posición en la estructura `args` donde existe un '<' -redirección de entrada-, o -1 en caso contrario. Del mismo modo, en el parámetro `indice_salida` para el símbolo '>' -redirección de salida-. Piense qué debe hacer el código del proceso *minishell* padre y del proceso *minishell* hijo, respectivamente, y retome la codificación de la función `ejecutar_orden()` para incluir el tratamiento de ambos tipos de redirecciones. Para ello, utilice también las funciones previamente implementadas en `redirecciones.c`.

- `void redirec_entrada(char **args, int indice_entrada, int *entrada)`. Esta función abre convenientemente el archivo que acompaña a la redirección de entrada, localizado en `args[indice_entrada+1]` (véase la Figura 9(b))), y devuelve en el parámetro `entrada` su descriptor del archivo.
- `void redirec_salida(char **args, int indice_salida, int *salida)`. Esta función devuelve en el parámetro `salida` el descriptor del archivo que acompaña a la redirección de salida, situado en `args[indice_salida+1]` (véase la Figura 9(b)).

Modificaciones en el *Makefile*

Al realizar esta fase, recuerde que, *quizás*, deba modificar el archivo *Makefile* para que, al invocar a `make`, la compilación de *minishell* se lleve a cabo correctamente teniendo en cuenta la nueva funcionalidad implementada.

10.7. FASE 7: Implementación de tuberías o *pipes*

Una vez realizada la FASE 6, y si ha seguido las pautas indicadas en esta memoria, tiene resuelta parte de la codificación de esta última parte donde se deben tratar órdenes compuestas. El uso de tuberías en realidad supone redireccionar, en este caso, no a un archivo cualquiera, como ocurre en el caso de uso de las redirecciones de entrada o salida, sino al extremo de escritura o lectura correspondiente de la tubería (utilizando el descriptor de archivo correspondiente) para cada orden que compone la orden compuesta introducida por el usuario, si fuera el caso (véase Figura 6).

Por lo tanto, con la FASE 6 implementada, no le queda más que resolver la creación de las posibles tuberías y la preparación de las redirecciones para cada orden separada por tuberías. Esta tarea consiste, *grosso modo*, en detectar cada aparición del símbolo '|' que representa una tubería e invocar adecuadamente a la función `ejecutar_orden()` con cada una de las órdenes separada por tubería(s) y con el descriptor de archivo "que va a hacer las veces" de entrada o salida estándar para cada una de ellas.

Comience por analizar los diferentes casos de órdenes separadas por tubería(s). Observará que la invocación a `ejecutar_orden()` debe ser distinta, en este sentido, si se trata de una

orden situada entre dos tuberías o acompañada sólo por una tubería (órdenes de los extremos). Y, adicionalmente, en este último caso, verá que es necesario discernir si es una orden con una tubería a la izquierda o a la derecha, respectivamente. En definitiva, una orden genérica con tuberías tendrá este formato:

```
minishell> orden_1 | ... | orden_i | ... | orden_n
```

Donde `orden_1`, `orden_i` y `orden_n` deben tratarse de forma diferente antes de invocar a `ejecutar_orden()`, en la que se debe indicar el descriptor de archivo que va a hacer el papel de entrada estándar y/o de salida estándar según el lugar que ocupe la orden en la línea de órdenes compuesta, tal y como se ha explicado anteriormente. La tarea de redireccionar correctamente en la función `ejecutar_orden()`, antes de la propia ejecución de la orden, ya la debe tener resuelta en la FASE 6.

Como apoyo en la implementación, se le propone completar el código que tenga hasta el momento de `ejecutar.c` incluyendo las líneas que se muestran a continuación.

Código 4: `ejecutar.c` incompleto

```
1  #include "minishell.h"
2  #include "parser.h"
3  #include "ejecutar.h"
4  #include "libmemoria.h"
5
6  int **crear_pipes(int nordenes)
7  {
8     int **pipes = NULL;
9     int i;
10
11    pipes = (int **)malloc(sizeof(int *) * (nordenes - 1));
12    for(i = 0; i < (nordenes - 1); i++)
13    {
14       pipes[i] = (int *)malloc(sizeof(int) * 2);
15       /* inserte linea de codigo para crear tuberia pipes[i] */
16    }
17    return(pipes);
18 }
19
20 pid_t ejecutar_orden (const char *orden, int entrada, int salida, int *pbackgr)
21 {
22    char **args;
23    pid_t pid;
24    int indice_entrada = -1, indice_salida = -1; /* por defecto no hay < y > */
25
26    if ((args=parser_orden(orden,&indice_entrada,&indice_salida,pbackgr))== NULL)
27    {
28
29       /* introducir codigo, para cerrar, si es necesario, entrada y salida */
30
31       return(-1);
32    }
33
34
35
36
37    /* codigo ya insertado para FASE 2 y FASE 6 */
38
39
40
41
42
43
44
45
46
47
48
```

```
49
50  }
51
52  void ejecutar_linea_ordenes(const char *orden)
53  {
54      char **ordenes;
55      int nordenes;
56      pid_t *pids = NULL;
57      int **pipes;
58      int backgr;
59
60      ordenes = parser_pipes(orden, &nordenes);
61
62      pipes = crear_pipes(nordenes);
63
64
65      /* para cada orden, invocar adecuadamente a ejecutar_orden */
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83       /* tratamiento de background para ordenes simples y compuestas */
84
85       /* liberar estructuras de datos dinamicas utilizadas */
86       free_ordenes_pipes(ordenes, pipes, nordenes);
87       free(pids);
88  }
```

Como puede observar, se ha modificado sólo la interfaz de `ejecutar_orden()` para tratar ya órdenes simples y compuestas. Simplemente, y sin variar el resto de parámetros y su semántica, ni el código ya implementado de la propia función para FASE 2 y FASE 6, se han introducido dos nuevos argumentos: `entrada` y `salida`, respectivamente, de tal forma que la invocación a `ejecutar_orden()` desde la función `ejecutar_linea_ordenes()` deberá incluir el valor para estos dos nuevos parámetros formales. Estos dos parámetros servirán para indicar los descriptores de archivo a los que deben ser redireccionadas la entrada y/o salida estándar de las órdenes que componen una orden compuesta en el momento de invocar a `ejecutar_orden()`. Y, en el caso más sencillo, si la orden es simple, debe invocar a la función `ejecutar_orden()` con valor 0 para el parámetro `entrada` (descriptor de entrada estándar) y con valor 1 para `salida` (descriptor de salida estándar), respectivamente. Además, en este caso, sin necesidad de realizar redireccionamiento.

Como apoyo para completar el código de `ejecutar.c` para la FASE 7, se proporcionan las siguientes funciones:

- `int ** crear_pipes(int ordenes)`. Crea tantas tuberías como número de órdenes pasado como parámetro menos uno (si hay n órdenes, hay n-1 tuberías intermedias) y las añade a una estructura dinámica de punteros a arrays de dos enteros. Estos arrays corresponden a los dos descriptores de archivo, de entrada y salida, asociados a cada *pipe* creado.

- `char ** parser_pipes(const char *orden, int *total)`. Convierte la cadena asociada a una orden (compuesta o simple), pasada a través del parámetro `orden`, en una

estructura dinámica de cadenas independientes correspondientes a cada una de las órdenes simples de que consta. Devuelve esta estructura y en el parámetro `total` el número de órdenes simples de `orden`. Esta función ya está implementada y se incluye en el archivo fuente `parser.o`.

- `void free_ordenes_pipes(char **ordenes, int **pipes, int nordenes)`. Esta función libera las estructuras `ordenes` y `pipes` de tipo array dinámico para las `nordenes` existentes. Esta función está ya implementada en el archivo fuente `libmemoria.c`.

Interpretar el código de funciones

Se deja intencionadamente al alumno que interprete el código proporcionado para las funciones `crear_pipes()` (en la que debe añadir una línea de código) y `free_ordenes_pipes()`.

A. Diagrama de la aplicación *minishell*

Finalmente, en la Figura 10 se muestran los diferentes archivos de la aplicación *minishell* y la relación entre ellos. Se representan con rectángulos de color naranja los archivos fuente con las funciones que debe codificar. Intencionadamente se han omitido las funciones correspondientes al desarrollo de las fases 3 y 5. El alumno debe decidir, en este caso, cuál es el archivo fuente donde considera más razonable ubicar ambas funciones así como sus prototipos o declaraciones. Asimismo, se muestran en el diagrama los archivos de cabecera y fuente ya codificados. En este último caso, se incluyen también las funciones definidas.

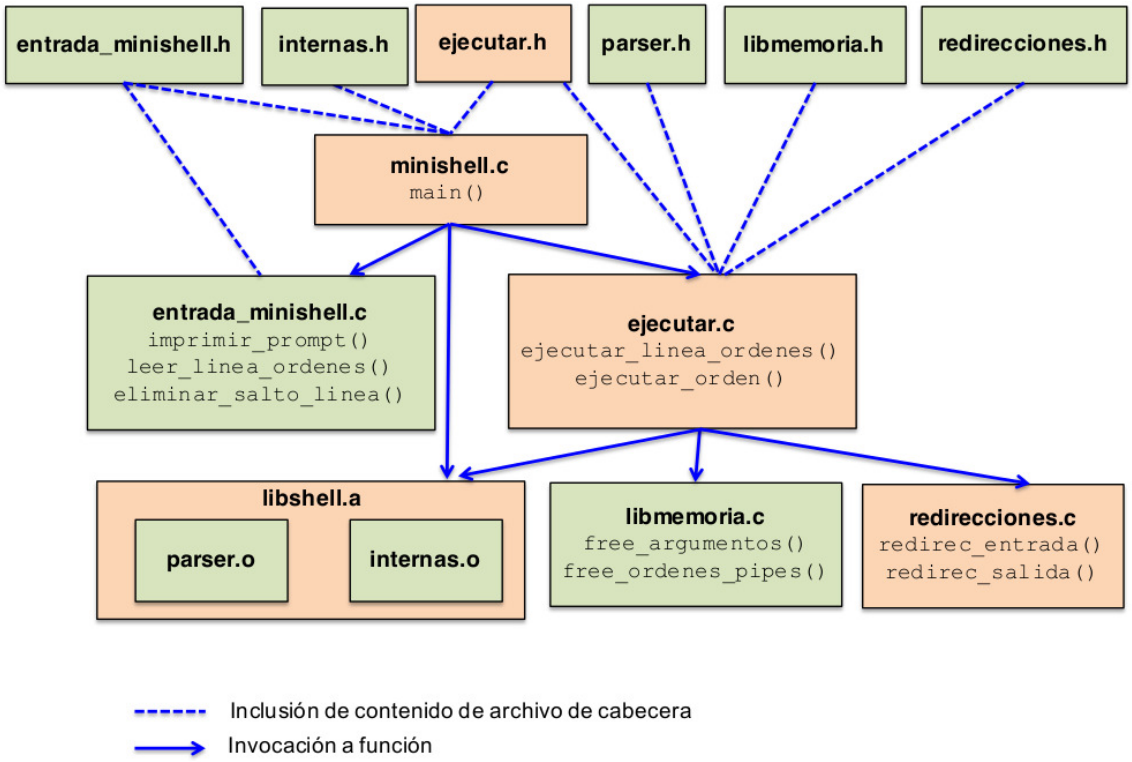


Figura 10: Esquema general de la aplicación *minishell*.