UNIVERSIDAD DE ALCALÁ

Departamento de Automática

Grado en Ingeniería de Computadores

Curso 2014/2015

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 modularizadas.
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 subobjetivo 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`".

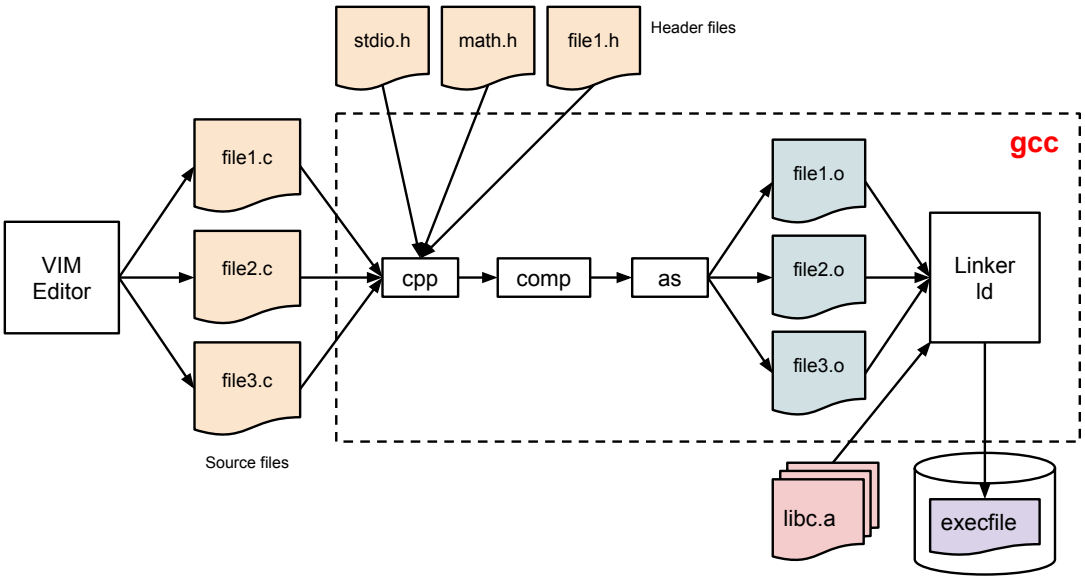


Figura 1: Generación de un archivo ejecutable en lenguaje C.

- *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 `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, `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`. Sin embargo, 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 el tutorial en los apéndices de la práctica.

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.

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.

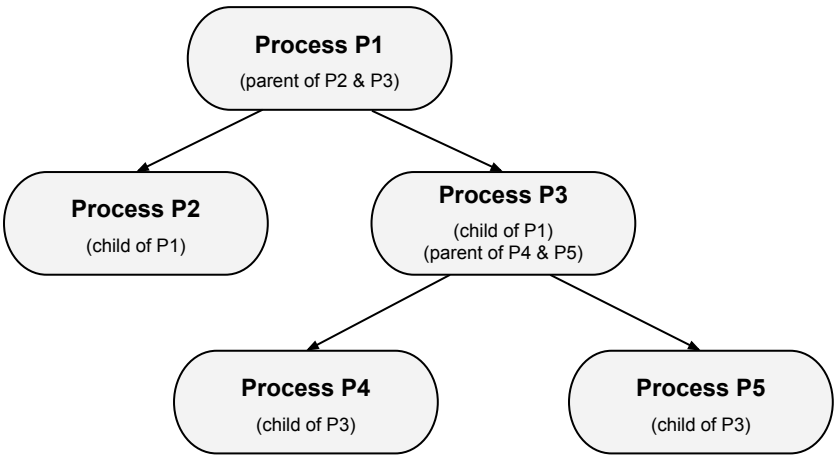


Figura 2: Jerarquía de procesos en UNIX

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 pueder 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 porceso 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[1].

8.1. 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`[2]. 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. 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[3] es la siguiente:

```
struct sigaction
   void(*sa_handler)();
```

[1] Servicios POSIX como `pipe()`, que permite crear una tubería, y que constituye, como ya se vio en la práctica 2, un mecanismo de comunicación (y sincronización) entre procesos, se aplicará en la asignatura de segundo curso: *Sistemas Operativos Avanzados*.

[2] 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.

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

```
    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[4].

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`[5].

```
#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 */
    /* 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.

[4]Se recomienda al alumno que consulte el `man` para `sigaction()`.

[5]La señal SIGINT se genera con la combinación de teclas <CTRL><C>.

8.2. 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 `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`”).

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 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 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.

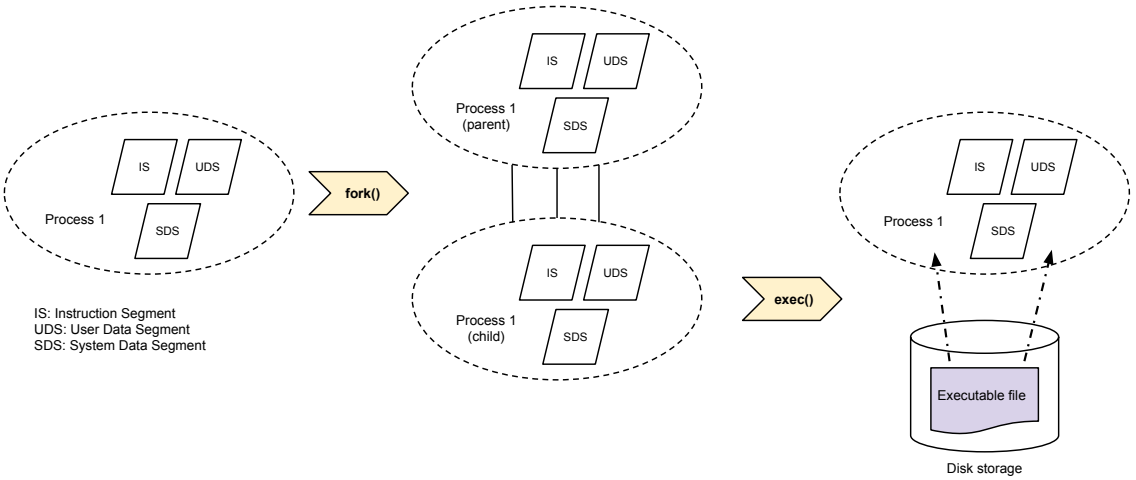


Figura 3: Ejecución de una orden externa dentro de un intérprete de órdenes

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

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

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

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()`

Edite un nuevo archivo fuente llamado `minishell.c`. Este archivo contendrá función principal `main` que será el punto de entrada del programa. Esta función debe realizar el ciclo de ejecución de órdenes de la *minishell* (véase sección 9.1), de acuerdo a las restricciones de la práctica y siguiendo el diagrama que se muestra en la Figura 4.

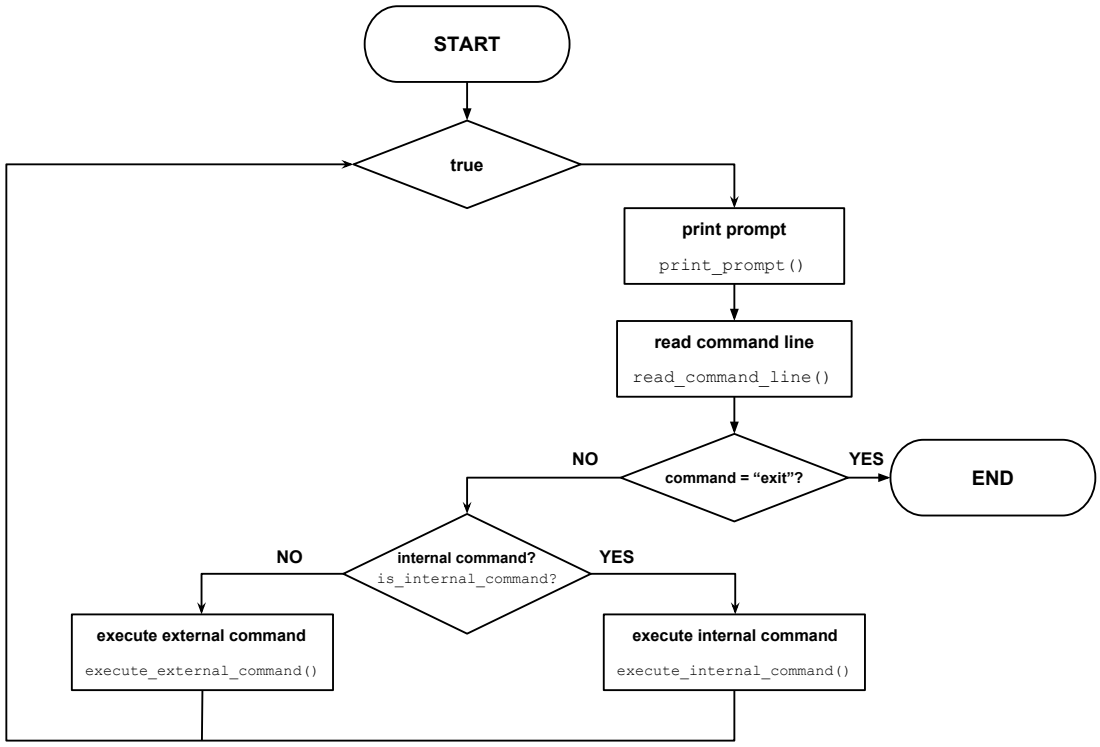


Figura 4: Diagrama de la función `main()` de la *minishell*.

A continuación, se muestra parte del código de `minishell.c`. Complete su implementación teniendo en cuenta el diagrama anterior.

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

```
4  #include <sys/stat.h>
5  #include <sys/wait.h>
6  #include <fcntl.h>
7  #include <signal.h>
8  #include <errno.h>
9  #include <unistd.h>
10 #include <string.h>
11
12 #include "internals.h"
13 #include "execute.h"
14 #include "jobs.h"
15 #include "minishell_input.h"
16
17
18 int main (int argc, char *argv[])
19 {
20     char buf[BUFSIZ];
21
22
23     // Your code goes here
24 }
```

Como puede observar en la Figura 4, para implementar la función `main()` se utilizan las funciones `print_prompt()`, `read_command_line()`, `is_internal_command()`, `execute_internal_command()` y `execute_external_command()`. Las cuatro primeras funciones se proporcionan como material de apoyo. La última función deberá ser implementada por el alumno en un bloque posterior. Puesto que el desarrollo de la práctica es incremental, para este primer bloque se sustituirá la llamada a la función `execute_external_command()` por una llamada a la función `printf` que mostrará un mensaje por la pantalla.

A continuación se detallan las funciones que debe esta información:

- **`void print_prompt()`**. Imprime el prompt de la *minishell*. Esta función se encuentra en el archivo `minishell_input.c`, proporcionado junto con el enunciado de la práctica.
- **`void read_command_line(char *buf)`**. Lee la línea de órdenes de la entrada estándar y la almacena como una cadena de caracteres en `buf`. Esta función se encuentra, al igual que la anterior, en el archivo `minishell_input.c`.
- **`int is_internal_command(const char *buf)`**. Indica si la orden que se pasa como argumento es interna (devuelve 1) o externa (devuelve 0).
- **`int execute_internal_command(const char * buf)`**. Si la orden es interna, la función `ejecutar_ord_interna()` ejecuta la orden. Las órdenes internas implementadas son: `cd`, `pwd`, `declare` y `umask`.

  Esta función se encuentra en el archivo `internals.c`. Este archivo forma parte de la biblioteca `libshell.a` proporcionada como material de apoyo al alumno. Esta biblioteca se utilizará al invocar `gcc` para que los archivos objeto junto con la biblioteca se enlacen en la última fase de compilación para generar el archivo ejecutable *minishell*, tal y como se ha visto en las clases teóricas.

Adicionalmente, se proporciona como material de apoyo los archivos: `minishell_input.h`, archivo de cabecera de `minishell_input.c`; y los archivos de cabecera `internals.h`, `parser.h` y `jobs.h`, correspondientes a las funciones y archivos incluidos en la biblioteca `libshell.a`.

10.2. PASO 2: orden interna `exit`

El alumno debe de implementar la orden interna `exit`. Para ello, antes de comenzar con el proceso de decisión descrito en el apartado anterior, es necesario que la *minishell* compruebe si la orden introducida es `exit`. En este caso, el proceso debe de terminar, ya sea retornando de la función principal `main()` o a través de la llamada al sistema `exit()`. Antes de finalizar, es **obligatorio** que el programa haga una llamada a la función `jobs_free_mem()`. Esta función tiene el siguiente prototipo:

```
void jobs_free_mem()
```

Está definida en la biblioteca proporcionada junto con el resto de archivos de la práctica. Esta función se encarga de liberar la memoria dinámica que haya podido ser reservada para el registro de procesos lanzados del bloque accesorio 1. Aunque no se implemente el bloque accesorio, es necesario que el programa llame a esta función antes de terminar.

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. El alumno **sólo tiene que implementar la ejecución de órdenes externas simples**.

Órdenes simples vs. órdenes compuestas

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 la siguiente función:

```
void execute_external_command (const char *command)
```

Esta función se encarga de ejecutar una orden externa simple, es decir, una orden sin redirecciones y sin tuberías. Esta función simplemente debe invocar a la función `ejecutar_orden()` de acuerdo a su prototipo, que se especifica a continuación. Adicionalmente, `execute_external_command()` debe resolver la posible existencia del símbolo de *background* al final de la orden (véase Ejemplo 2). En caso de no existir el símbolo '&', el proceso padre (*minishell* padre) debe esperar (invocando al servicio POSIX adecuado) a que el proceso hijo (*minishell* hija) finalice su ejecución. Cuando este evento suceda, la *minishell* padre podrá realizar otra iteración del bucle de la función `main()` y, por lo tanto, imprimir el *prompt* y volver a leer y ejecutar otra orden introducida por el usuario. En caso contrario, si la orden termina con '&', la *minishell* padre simplemente no debe esperar la finalización de la *minishell* hija y podrá volver a ejecutar inmediatamente otra iteración del bucle de la función `main()`. Esto significa que la *minishell* padre y la *minishell* hija podrán ejecutar concurrentemente órdenes.

A continuación se muestra un esqueleto de la función `execute_external_command()`:

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <sys/wait.h>
4
5  #include "parser.h"
6  #include "execute.h"
7  #include "free_args.h"
8  #include "jobs.h"
9
10
11 void execute_external_command(const char *command)
12 {
13     char **args;
14     int backgr = 0;
15     pid_t pid;
```

```
16
17      if ((args=parser_command(command,&backgr))== NULL)
18      {
19          return;
20      }
21
22      //Your code goes here
23
24      return;
25  }
```

Como puede observar en el código anterior, la función `execute_external_command()` en primer lugar debe convertir la cadena `orden` en una estructura que facilite su manejo en su ejecución con el servicio POSIX correspondiente. Para ello, se invoca a la función `parser_command()` cuya descripción y prototipo se detallan más adelante. Esta función ya está implementada y se proporciona compilado formando parte de la biblioteca `libshell.a`.

Antes de finalizar la función `execute_external_command()`, el proceso padre debe invocar a `parser_free_args()` ya que la función `parser_command()` crea un array dinámico que debe ser liberado para evitar lagunas de memoria. La función `parser_free_args()`, cuyo prototipo se muestra a continuación, ya está implementada en el archivo `free_args.c` proporcionado también como material de apoyo:

```
void parser_free_args (char **argumentos)
```

La función `parser_command()` tiene el siguiente prototipo:

```
char ** parser_command(char * orden, int * background);
```

Esta función convierte la cadena que representa una orden introducida al ejecutar la *minishell*, en una nueva estructura: un array de punteros a cadenas. Esta función permite un manejo posterior de la orden más adecuado para su ejecución con el correspondiente servicio POSIX. En concreto, `parser_command()` analiza la cadena `orden` y genera una cadena por cada uno de los argumentos de la misma, eliminando los posibles espacios entre ellos (considerando también el nombre de la orden como argumento). Si en la estructura obtenida hay un símbolo `'&'` (*background*), la función lo elimina de la misma pero devuelve un indicador de su existencia en el segundo parámetro, pasado por referencia.

Parámetros

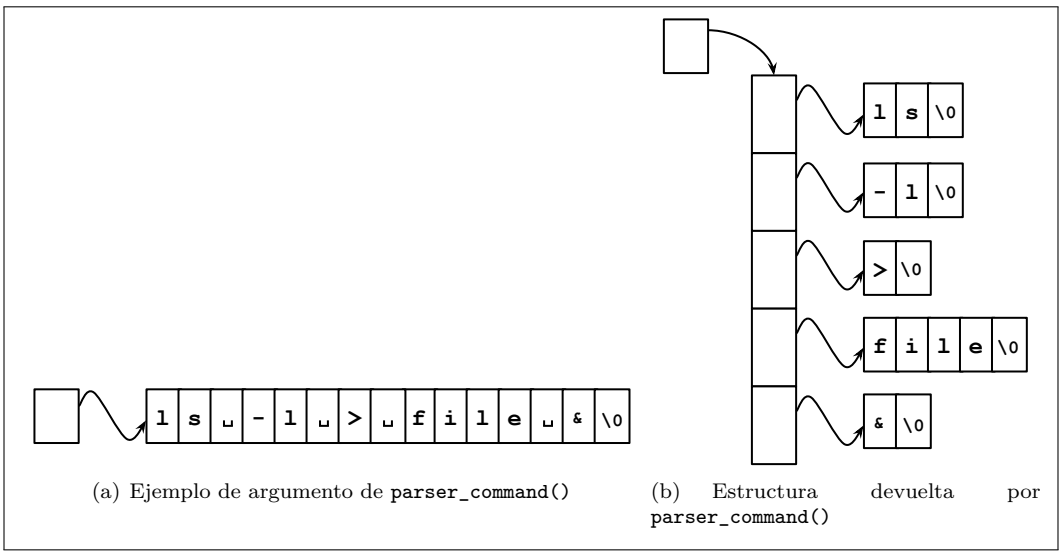
orden contiene la orden introducida en la ejecución de la *minishell*, y que se va a analizar y convertir en un array dinámico de punteros a cadenas.

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. Cada una de estas cadenas representa uno de los argumentos de la orden. Además, la función devuelve en `background` la existencia (1) o no (0) del símbolo *background* eliminándolo del array devuelto.

Por ejemplo, si la orden introducida (de tipo cadena constante), argumento de la función `parser_command()`, es: `orden=''ls -l -a &''` (Figura 5(a)), la función devolvería el array de cadenas de caracteres (de tipo puntero a puntero a carácter) que se representa en la Figura 5(b). Además, en este caso, el valor devuelto en el parámetro `background` sería 1, indicando que ha encontrado el símbolo `&` al final de la línea de órdenes.



(a) Ejemplo de argumento de `parser_command()`

(b) Estructura devuelta por `parser_command()`

Figura 5: Estructuras de datos utilizadas por `parser_command()`

10.4. PASO 4: gestión de procesos *zombi*

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 este paso puede dejar a la *minishell* hija en estado *zombi* que, como se ha visto en las clases teóricas, es un estado de los procesos no deseado. Razone porqué sucede y solucione este problema utilizando un manejador de señal.

Piense qué señal se podría utilizar para que, una vez que la *minishell* padre la capture, el manejador se encargue de evitar que la *minishell* hija quede en estado *zombi* y dónde se debería instalar el manejador.

Finalmente, una vez que tenga clara la solución al problema planteado, 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_external_command()` 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.

Finalmente, se muestra en la figura 6 las relaciones entre los módulos de la *minishell* y las funciones que **obligatoriamente** tiene que desarrollar el alumno para tener apta la parte práctica de la asignatura. Se incluyen en color rojo los archivos que debe implementar el alumno.

11. Requisitos de entrega

Para la evaluación de esta práctica, es necesario que los alumnos suban los archivos de la *minishell* a la plataforma web de la asignatura a través del enlace correspondiente habilitado a tal efecto. Se han establecido una serie de requisitos de entrega **OBLIGATORIOS**, que se deberán cumplir forzosamente para que la práctica sea evaluada. Aquella práctica subida que no se ajuste a estos requisitos será calificada directamente con un 0. Estos requisitos son los siguientes:

- **Archivo a enviar**. La práctica será enviada en un archivo de almacenamiento creado con la utilidad **tar** y comprimido con **gzip**. El nombre del archivo en cuestión a tendrá el siguiente formato:
  - Si la práctica la entrega un único alumno: `ID.tar.gz`, donde `ID` será el identificador utilizado por el alumno (DNI, NIE o número de pasaporte), que podrá estar compuesto únicamente por números o letras.
  - Si la práctica la entregan dos alumnos:`ID1_ID2.tar.gz`, donde `ID1` será el identificador del primer alumno y `ID2` el identificador del segundo. Para obtener este tipo de archivos, se tiene que emplear la siguiente orden:

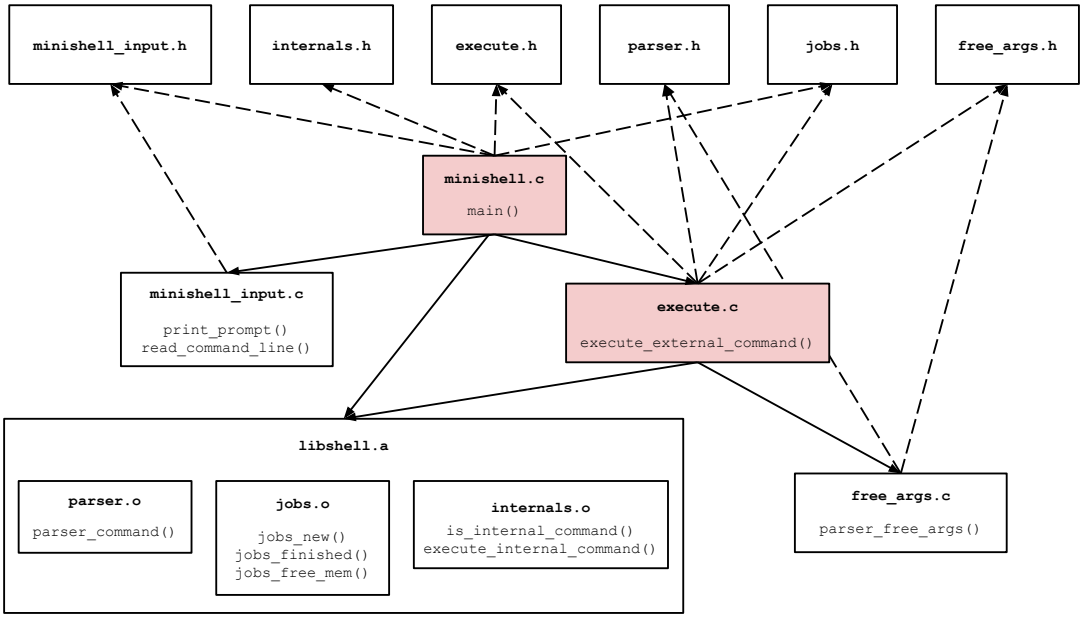


Figura 6: Esquema general de la aplicación *minishell*.

La siguiente orden es un ejemplo de cómo utilizar correctamente `tag` para empaquetar los archivos de la entrega:

```
tar cvfz 06574383H.tar.gz prac3
```

La orden anterior generará un archivo **tar** comprimido con **gzip** que contendrá los archivos del directorio `prac3`. **NO SE ADMITIRÁN ARCHIVOS COMPRIMIDOS CON OTRA UTILIDAD QUE NO SEA TAR + GZIP**. El envío de cualquier otro tipo de archivo supondrá un 0 irrevocable en la evaluación de la práctica.

- **Contenido del archivo comprimido**. El archivo comprimido deberá contener **ÚNICAMENTE** los siguientes archivos:

```
prac3/Makefile
prac3/P3Data
prac3/execute.c
prac3/execute.h
prac3/free_args.c
prac3/free_args.h
prac3/internals.h
prac3/jobs.h
prac3/lib
prac3/minishell.c
prac3/minishell_input.c
prac3/minishell_input.h
prac3/parser.h
```

Los archivos fuente y de cabecera son los mencionados en el enunciado. El contenido del archivo `P3Data` se describe en el siguiente punto. Si el archivo entregado no contiene alguno de los archivos anteriores, o si el directorio que los contiene no se llama `prac3`, la práctica se calificará con un 0.

- **Contenido del archivo `P3Data`**. El archivo `P3Data` tiene que seguir el siguiente formato. Cada una de las entradas debe aparecer en una única línea:

  1. `STUDENT1=Nombre y apellidos del alumno 1`
  2. `STUDENT2=Nombre y apellidos del alumno 2` o `STUDENT2=` si la práctica la entrega únicamente un alumno
  3. `LAST_STEP_TO_BE_GRADED=n`, donde n es el número de paso inclusive hasta el que se solicita la corrección de la práctica.

  Por ejemplo, si la práctica únicamente la entrega un alumno, de nombre *John Doe*, y éste entrega todos los bloques para calificar menos el último, su archivo `P3Data` deberá tener el siguiente contenido:

```
1 STUDENT1=John Doe
2 STUDENT2=
3 LAST_STEP_TO_BE_GRADED=5
```

  Junto con los archivos de la práctica se proporciona una plantilla para rellenar correctamente este archivo. La edición del archivo se podrá realizar empleando el mismo procesador de textos utilizado para editar el resto de los archivos del programa.

En la plataforma web de la asignatura el alumno puede encontrar un programa para comprobar antes de enviar la práctica que el archivo cumple con todos los requisitos. Este programa está implementado en el lenguaje Python. El archivo con el código del programa tiene el nombre `checkp3.py`. Para poder lanzarlo, es necesario tener instalado el intérprete de Python. Este intérprete viene por defecto instalado la mayoría de las distribuciones de Linux, así como en los sistemas basados en Mac OS X. Para poder lanzar el programa, es necesario ejecutar la siguiente orden:

```
python checkp3.py 06574383H.tar.gz
```

Si el programa detecta algún error, lo notificará debidamente con un mensaje por la pantalla. Si el archivo es correcto, mostrará un mensaje de *OK*.