UNIVERSIDAD DE ALCALÁ

Departamento de Automática

Grado en Ingeniería Informática Grado en Ingeniería de Computadores

Practice 3: Development tools and POSIX services for process management

Operating Systems

Contents

Warning to non(yet)-Spanish speakers

This practice is not yet fully translated into English. The translation process has been prioritized so essential parts to the laboratory practice are available first. Bulk translation has also be prioritized over spelling and grammar, but those aspects are very important so please do not hesitate to contact me with any correction. Thanks for your help and understanding.

Óscar García Población
✉ `oscar.gpoblacion@uah.es`

1 Related competences to achieve in this practice

1. Be able to understand how POSIX services for process management work, as explained in theory class.
2. Understand how process hierarchy if affected in UNIX/Linux when these services is invoked.
3. Apply the required POSIX service for process management.
4. Understand how the shell works.
5. Apply the knowledge acquired about process management POSIX services to the development of an application that need them, such as a shell.
6. Apply the skills explained in theory class on classic development tools: `gcc`, the use of libraries, `make`, and `gdb`.
7. Be capable of write applications made out of modules.
8. Be able to work in teams when developing applications.

2 Introduction

In the last practices we have studied how does the `bash` shell works as an interface with the user in Linux. In this practice we will develop a tiny shell that will help the user to launch the execution of other processes. To this end we will learn how to use POSIX services to create new processes and to manage them. There will be several exercises to test each different service individually, in order to fully understand how it works and how can it be used to provide the required functionality of our `minishell`.

Also we will learn to use a set of development tools commonly used in UNIX/Linux environment, that will help us to deal with the complexity of big applications with many different modules and developers.

When talking about programming in the UNIX/Linux environment often means to talk about programming in C. This language was created with the purpose of recoding the current version of UNIX, that in that time was written in assembler, in a higher level programming language. This decision had a deep impact on the development of Arpanet, and in its successor, Internet, since it was supported by UNIX machines. Therefore, the C language has strongly influenced the development of computing, and many other languages have taken elements from C.

Another important goal in this practice is to acquire good practices when coding along with a team, which is a highly appreciated skill in the industry. To this end we will introduce some concepts about coding style and version control.

3 [🔧] Code development cycle

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

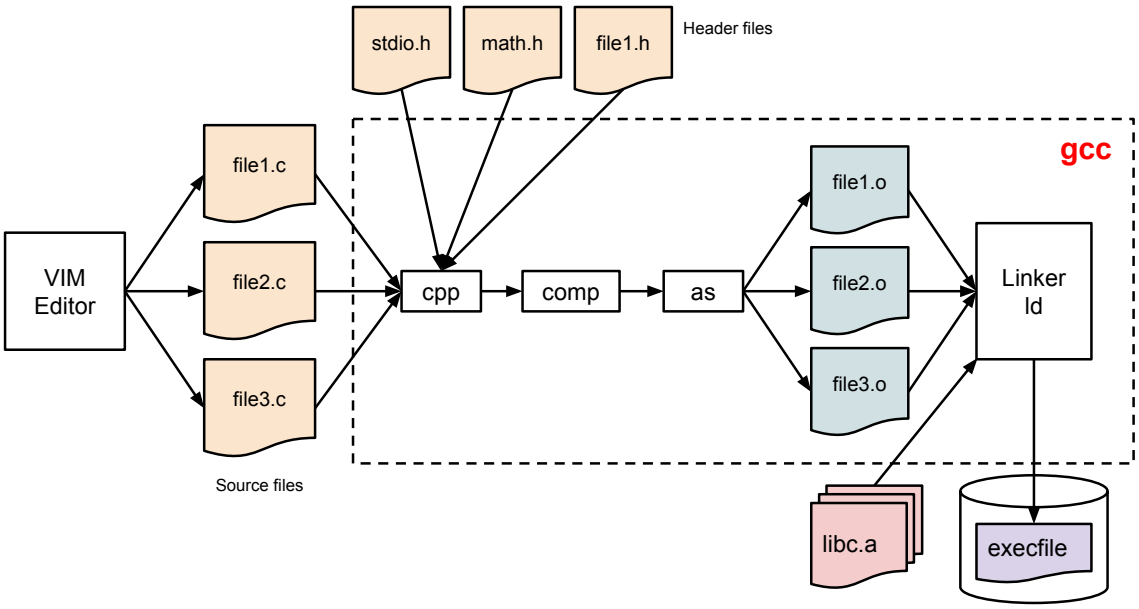


Figure 1: Generation of an executable made up of several modules in C language.

(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.

4 Build procedure automation: `make`

Developing a program is a cyclical procedure in which one or more source code files are modified, compiled, linked and executed. If we find a bug in while executing or we need to add new features,

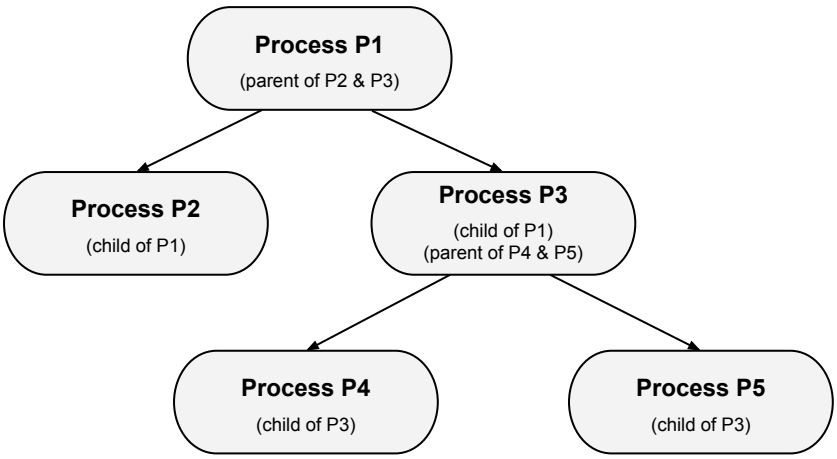


Figure 2: Processes Hierachy in UNIX.

we modify the source code and the cycle starts again. It is not necessary to recompile modules that haven't changed, it could also be very time consuming. Also there will be modules that depends on others to run, creating what is call a dependency tree. Keeping track of the things that has changed and controlling dependency by hand is a tedious and error prone activity.

To solve this problems, most of the UNIX environment introduced a tool called `make`. From the Wikipedia, `make` is a utility that automatically builds executable programs and libraries from source code by reading files called `Makefiles` which specify how to derive the target program.

Even though the use of `Makefiles` might seem to be unnecessary for small projects, it encourages and enforces good development practices and habits, and it is therefore **compulsory to use it in all the practices** in Operating Systems and Advanced Operating Systems.

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].

[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*.

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)();
   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:

[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.

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.

8.2 Servicio POSIX `kill()`

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

[4]Se recomienda al alumno que consulte el `man` para `sigaction()`.
[5]La señal SIGINT se genera con la combinación de teclas <CTRL><C>.

`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 The command interpreter: the shell

The shell is the traditional public face to the UNIX operating system. Understanding how does a shell works internally helps to clarify many concepts about the interaction with the operating system, to differentiate between user space and kernel space and to understand how to manage inter-process communication.

9.1 The shell's execution cycle

Basically, the shell perform the next steps over and over:

1. *Prompt the user.* This is to indicate to the user that the shell is ready and waiting for a command. When the user issues a command and press enter the shell proceeds to the next step.

2. *Parse the command.* To parse means to process what the users has written in the previous step. As we will see later, one of the parser's most important tasks is to split the user entry into independent parts, i.e. the command name, the arguments, the execution mode, etc. For example, if the user typed `cat practice1.c > out.txt`, the result of the parsing would be `command: cat`, `argument: practice1.c`, `stdout redirection: file out.txt`.

3. *Process the command.* Before the actual command gets executed, there is some extra processing to be done, such as variables and metacharacter expansion, redirection and pipes handling, and background execution detection. For example, if the user issues the command `ls *.pdf`, and in the current working directory there are two PDF files named `exam.pdf` and `solution.pdf`, then after this processing the final command to be executed will be `ls exam.pdf solution.pdf`. This is an example of metacharacter expansion.

4. *Command execution.* There are two kind of command: internal and external commands. Internal commands are those that the shell can handle be itself. This may change from one implementation to another, but for example the command `cd` used to change the current working directory is an internal command, while `myplayer`, which launches the execution of a new MP3 music player is an external command.

10 The `minishell`

In this practice we will apply process management POSIX services to develop a very simplified version of the *BASH shell* that we will call `minishell`.

Like any other UNIX process, the `minishell` will use the standard input (typically attached to the keyboard) to receive instructions from the user, and the standard output (typically attached to the screen) to provide responses back. Also it will use the standard error to output any error message.

The `microshell` will have the following features:

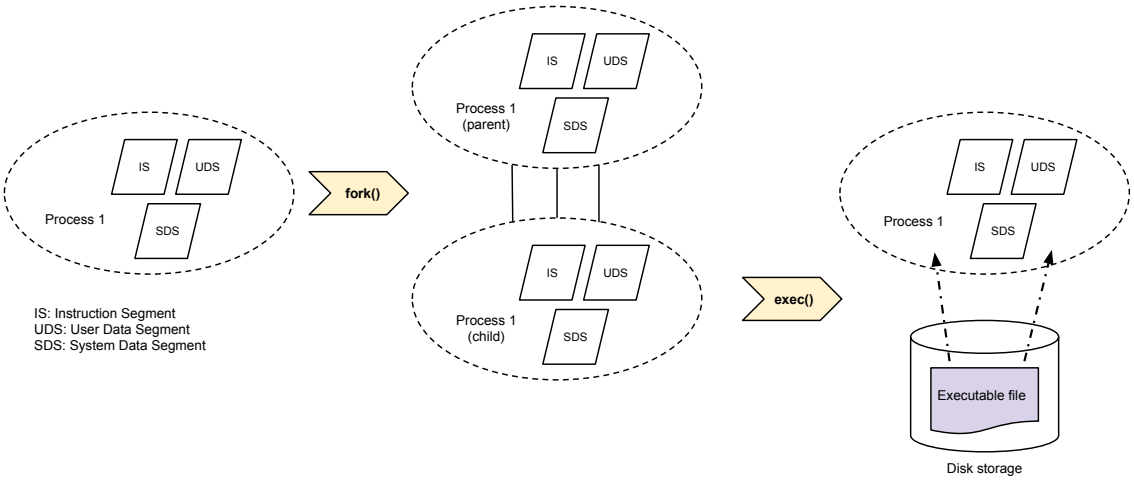


Figure 3: External command execution by the shell.

1. **External commands foreground execution**. For example `ls -l` , `who`, etc. To execute an external command, the `minishell` will `fork()` a child process. This child will be a copy of the father, so it needs to execute `exec` to change it code to the external command to be executed, and then proceed to execute it. Meanwhile the father shell will `wait()` until its child finishes. When this hapend, the father shell will resume its own execution and prompt the user for the next command.

2. **Internal command `exit()`**. When the `minishell` receives this command, it will finish its execution.

3. **External commands background execution**. When a command issued to the `minishell` ends with the ampersand symbol, `&`, the execution of the command by the child process will be done in the background. This means that the father will not wait until de child finished, and instead it will continue its own execution prompting the user for a new command while the child continues its execution. Although up to this point it is possible to execute commands in the background, it does not deal with *zombie* state that the processes are left into. An extra feature will deal with this.

4. **Job listing**. The `minishell` features an internal command `jobs` to list all the processes that this shell has launched.

5. **Execution of command sequences**. The commands will be separated by the semicolon symbol `;` and will be executed sequentially, one first and the following.

The use of a `Makefile` is compulsory in all laboratory practices

Every practice in the laboratory must include a `Makefile` that automates the build procedure. This build procedure may not issue warning messages and must finish correctly. The `Makefile` has to include also a `clean` rule that deletes all the files that are not strictly necessary to build the final executable.

To complete the exercise you will need to use the provided source files, header files, and a pre-built library `libshell.a`. They can be downloaded from the web page. Remember that the library is already compiled, so you need to choose the proper one to use depending on the system's architecture of the computer you will be using to work on this practice.

10.1 STEP 1: The main function

Edit a new source code file and call it `minishell.c`. This file will contain the `main` function, and therefore it will be the entry point from where the execution will start.

The function `main()` will implement the execution cycle explained in section 9.1 following the steps shown in the figure 4.

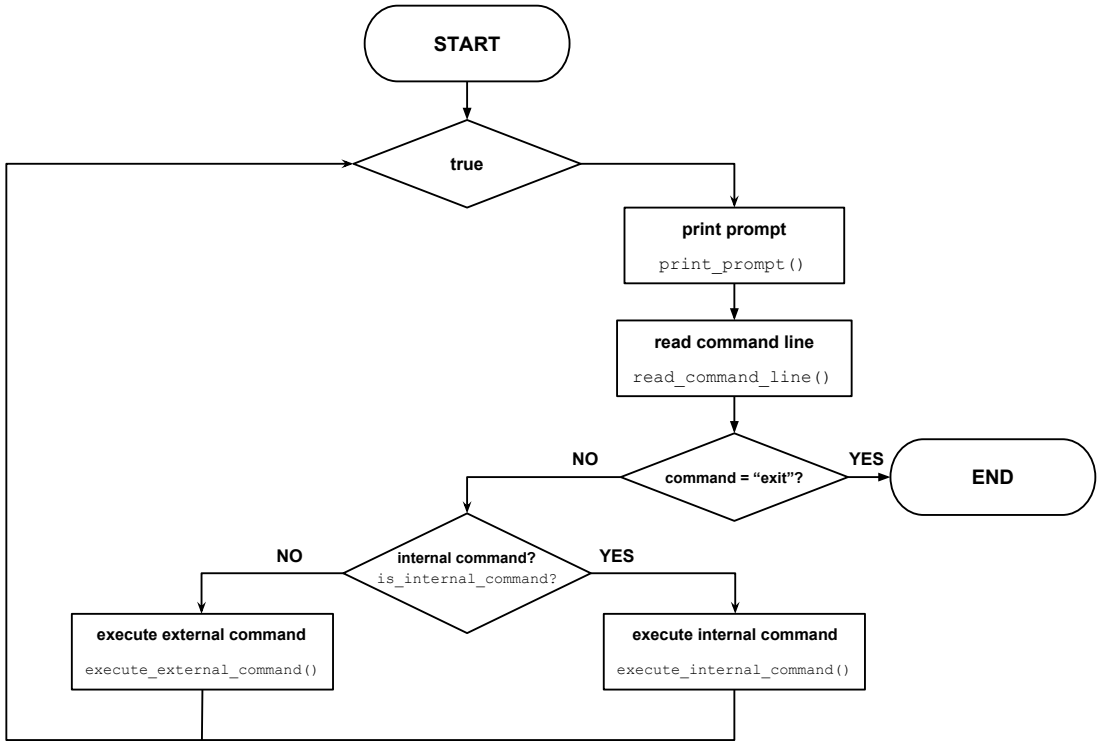


Figure 4: Workflow diagram for the `main()` function.

The following is a template of the `minishell.c` source file. Complete it following the execution flow from the diagram 4.

```
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 }
```

To complete this step you will need to use four functions: `print_prompt()`, `read_command_line()`, `is_internal_command()` `execute_internal_command()` and `execute_external_command()`. The

first four are already implemented and provided as support material for this practice. The fourth does not exists yet and therefore we can not use it. But since we want to do incremental development, instead of using the inexistent function we are going to substitute it by a placeholder. The easier placeholder is just a call to `printf` with a message saying that *this function has not already been implemented*. This way we will be able to compile all we have up to now and check that it works. But first let's review each of the provided functions so we can use them properly:

`void print_prompt()` . Prints the `minishell` prompt in the standar output. This function is in the file `minishell_input.c`.

`void read_command_line(char *buf)` . Reads user input from the standard input and stores it as a string of characters in the position pointed by the `buf` pointer. This function is in `minishell_input.c`.

`int is_internal_command(const char *buf)`. Returns true (1) or false (0) depending on whether `buf` is an internal command or not.

`int execute_internal_command(const char *buf)` . The pointer `buf` points to a null-terminated string with the name of a command. Executes the internal command in `buf`. The internal commands are `cd`, `pwd`, `declare` and `umask`.

This function is defined in the file `internals.c`, along with some other things related with internal commands. In turn, `internals.c` is included in the `libshell.a` library that is available as suport material for this practice. In the next course, Advanced Operating Systems we will implement some of this commands in a laboratory practice. By now, we will use the current implementation by linking the library with our own source code using the flag `-l`.

The file `internals.h` which is the corresponding header file of `internals.c`, and the header files `parser.h` and `jobs.h`, are also provided as support material.

Following the incremental coding style, now that we have implemented a small set of functionality, it is time to compile and check that everything is working so far as expected. However, there are already many files and libraries involved, so we are not going to compile things manually. Instead we will now write a `Makefile` that will help us during the rest of the development. The section 10.7 in the page 20 describes the basic dependency graph. This graph will help you with the task of completing a `Makefile`.

10.2 STEP 2: The internal command `exit`

In this step we will complete the implementation of the internal command `exit`. If this command issued the `minishell` must finish its execution getting out of the `main()` function or calling `exit()`. But before finishing it is mandatory to call `jobs_free_mem()`. This function has the following prototype:

```
void jobs_free_mem()
```

It is library provided with the rest of the files of this practice. This functions takes care of freeing the dynamic memory that might had been allocated in operations in step 4. Even if this step wouldn't have been implemented this call has to be made before finishing.

10.3 STEP 3: external command execution

For this step we will code `execute.c`. This file should contain only the functions directly related with the execution of a single line with a simple commands.

Simple vs. compount commands

Consider the following command line:

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

This command line contains two command communicated with a pipe. These are the lines that we will considered as compound command. Note that there is also an output redirection.
On the other hand, the following examples are simple commands:

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

The file `execute.c` must contain the following function:

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

This function executes a simple external command, that is, a command with no redirections and no pipes. This function must fork a child process that will in turn execute the external command calling the corresponding `exec` function. If the command is to be executed in the foreground, that is, no ampersand `&` symbol has been found, then the father `minishell` has to `wait` until its child finishes its execution.

On the other hand, if background execution is requested then everything is the same, except that the father will not wait for the child, prompting the user for a new command immediately. In other words, the father and the child will execute concurrently.

Beware of the zombies

It seems that background execution is easier than foreground, but since the father doesn't worry of waiting for the child when they finish nobody will be listening for their execution result code. This means that the child has no more code to execute but if can not die because they need to communicate the result to the father; they will become zombies. You must deal with this problem and get rid of the zombies to quality for this part of the practice.

The following is a template to the function `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 }
```

As you can see in the previous code, the function `execute_external_command()` converts in the first place the string `command` in a structure so that it is easier to use. For this, the function `parser_command` is called. This function, that will be analyzed later, is included in the provided library `libshell.a`.

Before the end of the `execute_external_command()` function, it is necessary to call `parser_free_args(char** argumentos)` because `parser_command()` creates a dynamic array that has to be released, otherwise this memory will remain reserved but inaccesible and can't be reused for other purposes. This is call *memory leak*. This function is also included in the provided library.

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

One of the key functions of this practice is `parser_command()`. This function transforms the string with the user input in a new structure: an array of pointers to string. This is a better representation for the POSIX service `exec`, which is it consumer. Its behaviour is described in detail next.

The function `parser_command()` has the following prototype:

```
char ** parser_command(char * command, int * background):
```

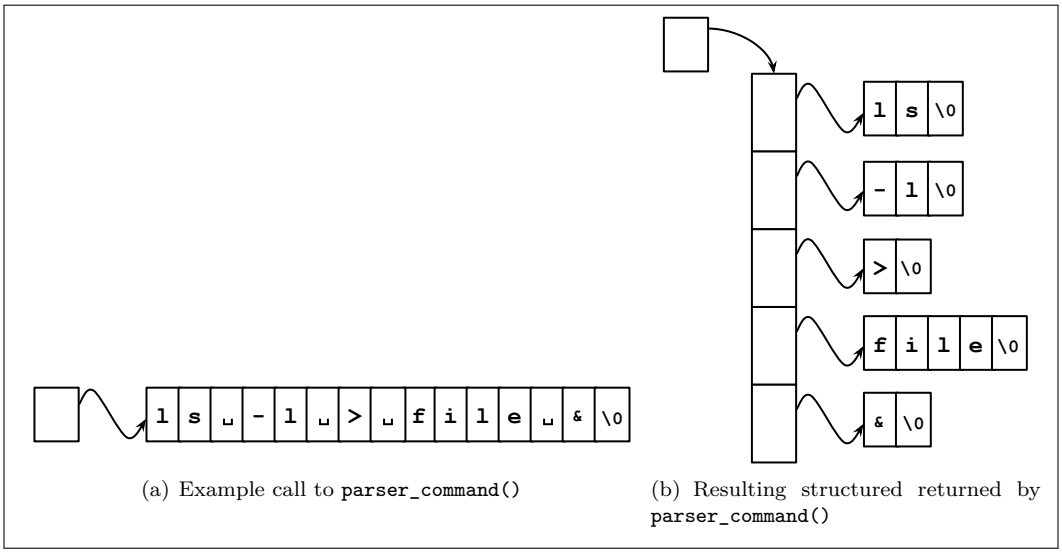
This function scans the input string and, using the whitespace as a delimiter, it splits it in one string for each word. If the input string contains a background symbol `&` it removes it but also acknowledges it by writing a '1' in the position pointed by its argument `background`.

These are the required arguments:

command (input argument) is the input string from the user to be parsed by this function.

background (output argument) is a pointer to an integer where this function will write a '1' if the background symbol `&` was found while scanning and parsing `command`, or '0' otherwise.

For example, if the user issued `orden=''ls -l -a &''` as an argument to the `parser_command()` function, then the result will be the one shown in the figure 5(b).



(a) Example call to `parser_command()`

(b) Resulting structured returned by `parser_command()`

Figure 5: Structures used by `parser_command()`

10.4 STEP 4: dealing with zombie processes

If you check what happens when you try to execute a command in background with the code as it is right now, you will find that after each background execution the child do not finishes, as it should. Instead, if remains in zombie state because the father, the `minishell`, is not waiting (`wait()`) for them to finish and therefore they can not communicate their exit code. You can check this executing something in background, for example `sleep 5 &`, wait five second, open a new terminal and try `ps ax`. You will see that `sleep` is already there but in the `Z` state. Of course, if you exit from the `minishell` all the zombies will become orphans, they will be adopted by `init`, `init` will `wait` for them and finally they will die.

To solve this problem we have to focus on its root, which is that the father is doing something else but waiting for its children to die. We need a mechanism to interrupt the father when a child dies, and we have it: signals. Whenever a child dies it send a signal to his father, specifically `SIGCHLD`.

The goal in this step is to implement the signal handler that deals with the `SIGCHLD` signal. Is up to the student to find out the why to implement this signal handler checking the documentation.

Try the following commands in background to check that no processes are left in the zombie state:

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

10.5 STEP 5: listing jobs

The `minishell` will keep a record of all the processes created during its execution. By using the command `jobs`, the user will be able to list these processes. The list will show the PID assigned to the process and the command name. Also it will show the status of the process, i.e, if it is still running or it has already finished. To implement this functionality the following functions are provided:

`void jobs_new(pid_t pid, const char * name)`. This function registers in the internal structure the new process. The arguments to this functions are the PID and the name of the external command. This name is always the first element in the command line.

`void jobs_finished(pid_t pid)`. This function is used to flag that the process corresponding to the given PID has finished.

When a new process is launched, the father `minishell` must register it calling `jobs_new()`. Also, when the child finished the father must call `jobs_finished()` to register this fact. It is important to remark that this registry must happen whether the command was executed in the foreground or in the background.

10.6 STEP 6: list of commands separated by semicolon

If the `minishell` detects the semicolon symbol it will assume that instead of a single command, it has to execute a list of commands, from left to right, one after the other, using the semicolon as a delimiter between commands.

Making the long story short, it is a matter of calling `execute_external_command` as many times as commands there are in the user input line separated by the semicolon. You can use any of the library functions to work with strings in C, for example `strtok()`, `strsep()`, etc.

10.7 A `Makefile` for my `minishell`

The final executable file of the `minishell` is built from many different components, and these components also depends on another components to compile. The figure 6 shows the dependency relations between the components. The files to be developed by the student are depicted in red.

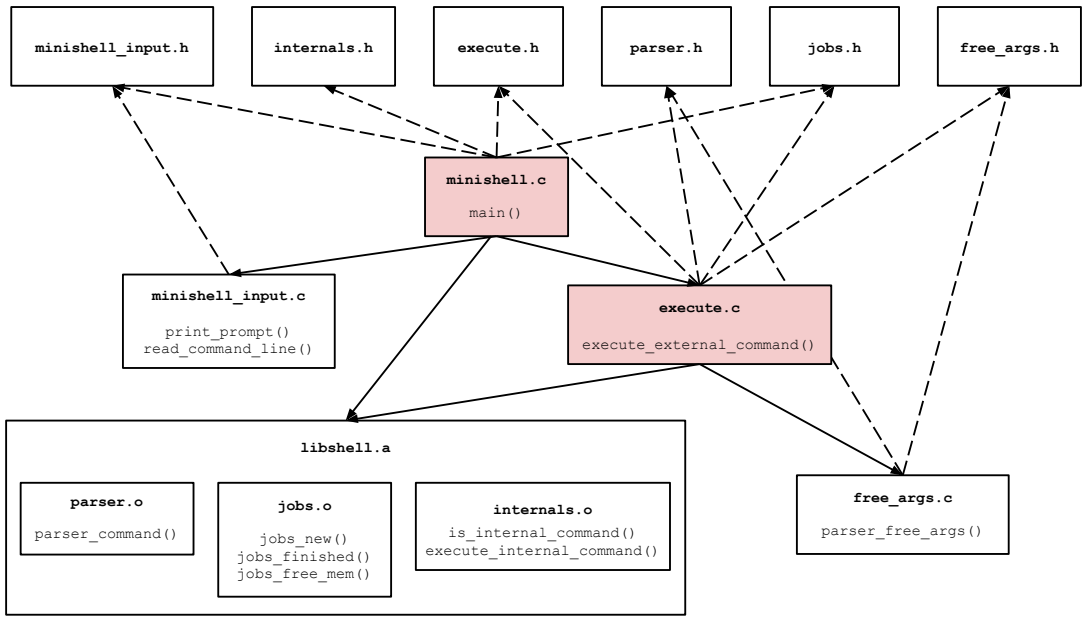


Figure 6: The *minishell* build dependency graph.

The graph does not include the optional parts of this practice. It is up to the student to include them if he implements them.

Remember to include a `clean` rule to wipe out all the things that can be generated from others, such as object files, executables, backup files, etc. You can use the following rule as an example:

```
clean:
    rm -f *.o
    rm -f executable
    rm -f whateverYouDontNeed
```

11 Delivery procedure

In order to be grade, the student have to upload the `minishell` files to the course's web platform. It is compulsory to strictly comply with the existing regulations to submit the practice, otherwise it will not be graded.

The regulations are as follow:

- **Archiver type to submit**. All the files must be archived in a `gzip` compressed `tar` archiver. The name of the file will follow this format: `ID.tar.gz`, where `ID` is the identification number used by the student (DNI, NIE, passport number), than can only be made up of letters and numbers.

  To create this kind of file you can use the following example command:

  `tar 06755244H.tar.gz prac3`

- **Contents of the archiver**. The archiver will contain **only** the following files:

  ```
  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
  ```

  Source and header files are the one explained previously in the practice. The file `P3Data` is described next. Check that you are not submitting more or less than those files or the checker will refuse to process the practice and it will be not graded.

- **`P3Data` contents**. The file `P3Data` will follow the next format, with an entry per line:

  1. `STUDENT1=Name and surname of the student`
  2. `STUDENT2=`
  3. `LAST_STEP_TO_BE_GRADED=n`, where n is the number of the last step to be graded.

  For example, if the practice is to be submitted by *John Doe* and he submits all the steps but the last, the contents of the file will be as follows:

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

  Along with the files downloaded for this practice there is a template file P3Data. You can edit it with any text editor.

In the downloads there is a file called `checker.py` that will help to determine if the file that you are about to submit complies with the rules. To run it type:

```
python checkp3.py 06755244H.tar.gz
```

If it passes the rules then it will reply with an OK message. Otherwise if will issue an error message.