**Librerias de C:**

<https://www.ibm.com/docs/es/i/7.5?topic=functions-include-files>

**Vamos a tener que usar GLib para nuestras listas**

**SHELL**

es un programa que actúa como una interfaz entre el usuario y el sistema operativo. Su función principal es interpretar y ejecutar los comandos que el usuario introduce, además de proporcionar un entorno en el que los usuarios pueden interactuar con el sistema operativo.

**Programa Robusto**

Un programa se considera robusto ante entradas incompletas y/o inválidas cuando puede manejar situaciones en las que los datos de entrada no son correctos, están mal formados, son incompletos, o no siguen el formato esperado, sin fallar o producir resultados incorrectos.En lugar de colapsar o comportarse de manera impredecible, un programa robusto puede:

1. **Detectar errores:** Identificar cuando una entrada es inválida o incompleta.
2. **Manejar el error de manera apropiada:** Esto podría implicar ignorar la entrada, pedir una nueva, o proporcionar un valor por defecto.
3. **Informar al usuario:** Mostrar mensajes de error claros que expliquen qué salió mal y cómo puede solucionarse.
4. **Continuar funcionando:** Seguir ejecutándose de manera segura y estable, incluso si encuentra entradas problemáticas.

**Foreground y Background**

En un shell, los comandos pueden ejecutarse en **Foreground** (primer plano) o **Background** (segundo plano).

### **Foreground (Primer Plano):**

* Cuando ejecutas un comando en **foreground**, el shell espera a que el comando termine antes de permitirte realizar otra tarea. El comando tiene el control total de la terminal hasta que finaliza.

### **Background (Segundo Plano):**

* Cuando ejecutas un comando en **background**, el comando se ejecuta "detrás de escena", lo que permite que la terminal siga aceptando otros comandos mientras ese proceso sigue corriendo.
* Para ejecutar un comando en **background**, se coloca un **ampersand (&)** al final del comando.

**Foreground Execution:**

* Simplemente ejecutas el comando y esperas a que termine. Esto puede hacerse usando funciones como fork() y execvp() en C.
* El shell espera el resultado del comando con waitpid().

**Background Execution:**

* Para ejecutar un comando en background, primero debes comprobar si el comando termina con &.
* Si es así, creas un nuevo proceso usando fork() y ejecutas el comando en el proceso hijo con execvp().
* No esperas a que el proceso termine, permitiendo que el shell siga aceptando nuevos comandos.

**Parsear/Parsing**

tarea de interpretar y descomponer comandos, datos, o archivos en partes más manejables para su procesamiento posterior.

**Pipeline**

consiste de dos (o más) comandos simples conectados vía operador “|”.permite conectar la salida de un comando directamente a la entrada de otro, creando una cadena de procesamiento de datos.

Cuando usas una **pipeline** en la línea de comandos, estás utilizando el carácter | para conectar dos o más comandos. La salida estándar (stdout) del primer comando se convierte en la entrada estándar (stdin) del segundo comando.

Ej: $ ls -l | grep ".txt" | sort

>ls -l lista los archivos en el directorio actual en formato largo.

>La salida de ls -l se pasa directamente a grep ".txt", que filtra y muestra solo los >archivos con la extensión .txt.

>Luego, la salida filtrada de grep se pasa a sort, que ordena alfabéticamente los resultados.

Al ejecutar una pipeline, el shell realiza los siguientes pasos:

1. **Creación de procesos hijos:**
   * Para cada comando en la pipeline, el shell crea un nuevo proceso hijo usando fork().
2. **Redirección de entrada/salida:**
   * El shell configura las tuberías para redirigir la salida de un comando a la entrada del siguiente.
   * Esto se hace usando la función pipe(), que crea un par de descriptores de archivo; uno para lectura y otro para escritura.
   * Después, el shell usa dup2() para redirigir stdout del primer comando al descriptor de escritura de la tubería y stdin del segundo comando al descriptor de lectura de la tubería.
3. **Ejecución de los comandos:**
   * Cada proceso hijo ejecuta su comando utilizando execvp() o una función similar.
   * Mientras tanto, el shell principal puede esperar a que todos los comandos terminen utilizando waitpid().

*llamadas al sistema (syscalls)*

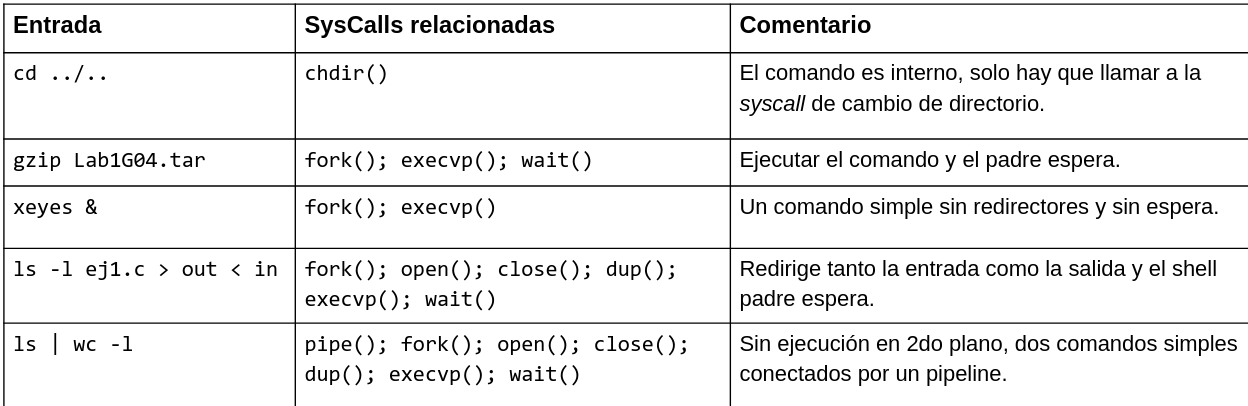
fork() /\* para crear un nuevo proceso (hijo) \*/

pipe() /\* para crear una tuberia \*/

dup() /\* para modificar un descriptor de archivo \*/

wait () /\* para bloquear un proceso \*/

execvp() /\* para ejecutar un programa externo \*/



NO TE ANDAN LOS TESTS?

A NO DESESPERAR PAPILISIMO

CPPFLAGS+= -I.. $(shell pkg-config --cflags glib-2.0)

LDFLAGS+= $(shell pkg-config --libs glib-2.0) -lcheck -lsubunit -lm