PRÁCTICA

Objetivo:

Programar un intérprete de órdenes sencillo que permita la ejecución de trabajos con un solo programa, sin cauces, ni redirección de la entrada/salida.

Descripción:

El programa debe soportar algunas de las características propias de los *shells que* detallaremos a continuación. Siempre que no se especifique lo contrario, el comportamiento será análogo al del *shell* por defecto (*bash*).

Sintaxis de línea de orden:

`orden argumentos [& | ;] orden argumentos [& | ;] ...`

Operación:

Las líneas de orden puede contener más de una orden, separadas mediante ";" o "&", en este último caso, la orden se ejecutará en *background*.

Como mínimo debe tener las siguientes *órdenes internas*:

- **`cd dir`** : cambia el directorio de trabajo a *dir*. Si se usa sin argumentos cambia al directorio raíz del usuario (variable de entorno `HOME`).
- **`pwd`** : muestra el directorio de trabajo actual.
- **`jobs`** : muestra la lista de trabajos (jobs), con el siguiente formato:

  `[jobID] estado línea_de_orden`

  dónde
  - `jobID` = struct `job.jobID`
  - `estado` = "Running" o "Stopped"
  - `línea_de_orden` = `struct job.texto`

- **`wait n`** : espera a que termine de ejecutarse el trabajo *n* . Si se usa sin ningún argumento, espera a que finalicen todos.
- **`kill n`** : elimina de ejecución el trabajo número *n.*
- **`stop n`** : detiene la ejecución del trabajo en *background* número *n* (*SIGSTOP*).
- **`fg n`** : pasa a *foreground* el trabajo número *n* (que debería estar detenido).
- **`bg n`** : pasa a *background* el trabajo número *n* (que debería estar detenido).
- **`times`** : (opcional) muestra el tiempo acumulado de usuario y de sistema.
- **`date`** : muestra la fecha actual.
- **`exit`** : finaliza la ejecución del programa de forma segura. Envía una señal *SIGKILL* a los trabajos en background y aguarda a que estos terminen antes salir.

Todo aquello que no sea una orden interna, se interpretará como un *comando externo* que es necesario ejecutar mediante `fork()` y `execvp()`.

Un trabajo en *foreground* puede ser detenido pulsando `CTRL+Z` desde el terminal de control, en cuyo caso el *shell* debe retomar el control. Para ello debe detectar mediante la llamada `waitpid()` que el trabajo por el que esperaba no ha acabado sino que ha sido detenido a causa de la señal *SIGTSTP*.

Para poder manipular el terminal de control sin que se pare nuestro *shell* es necesario ignorar la señal SIGTTOU.

Además, es necesario capturar la señal SIGINT (`CTRL+C` desde terminal) para evitar la finalización del shell como ocurre con bash.

La siguiente figura ilustra los tipos de datos que es necesario manejar, para llevar a cabo la implementación del *shell* .

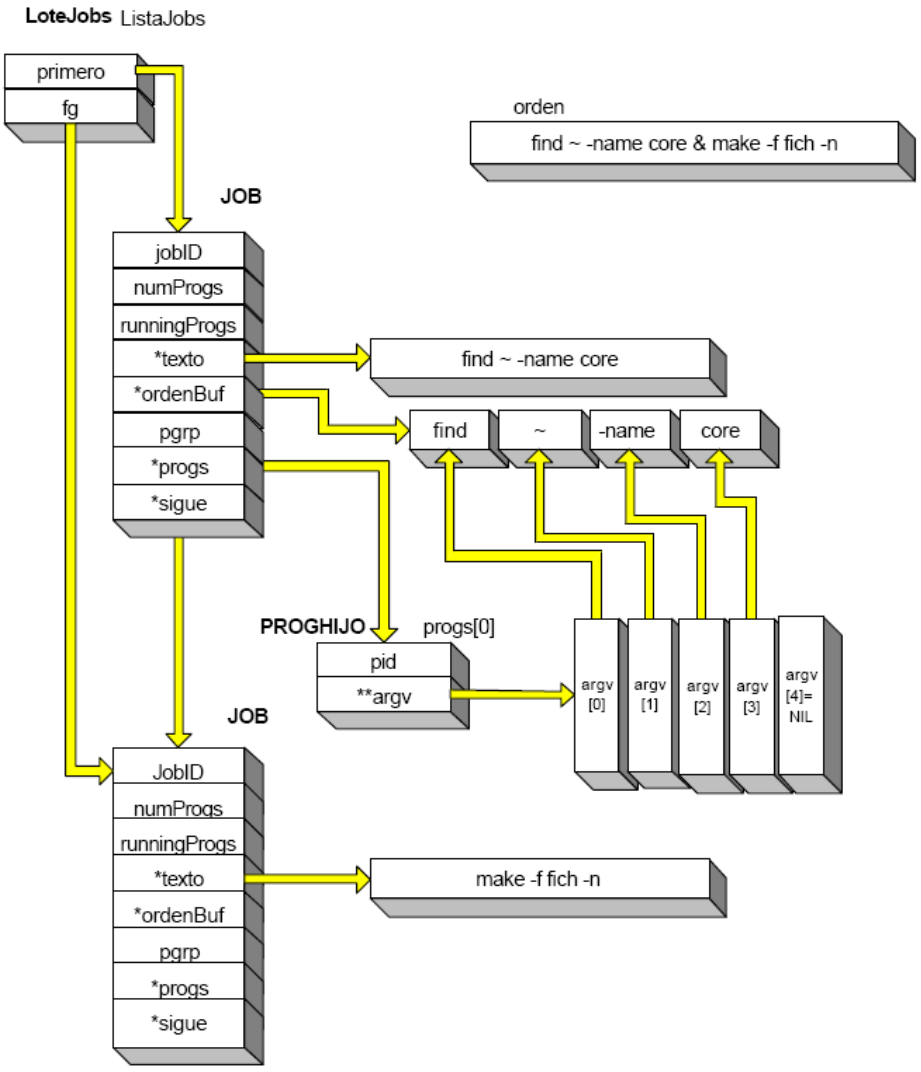

Fig. 2 Estructuras de datos manejadas por el shell.

El campo `jobID` de la estructura `job` se gestiona de una manera análoga al resto de los *shells*, cuando se inserta un nuevo trabajo:

- Si la lista esta vacia --> jobID = 1
- Si la lista no esta vacia --> jobID = jobID_último + 1

Cuando se ejecuta un trabajo en *background*, después de lanzarlo a ejecución es preciso informar del siguiente modo (PID representa el *pid* del proceso):

```
[jobID] PID
```

Para facilitar la implementación de la práctica se proponcionan los siguientes ficheros que podrán utilizarse como plantilla. Incluye un análisis de ordenes rudimentario básico.

`shell.h`	Definiciones de tipos de datos y prototipos de funciones
`shell.c`	Función principal (`main`)
`shell_orden.c`	Funciones de manejo de ordenes
`shell_jobs.c`	Funciones de manejo de jobs

La lectura de ordenes que se proporciona es muy básica y se puede mejorar haciendo uso de la función `readline()`. El fichero `shell-readline.txt` muestra el esquema general de shell usando `readline()` para la lectura de ordenes.