

INTRODUCCION A LAS PRACTICAS DE SISTEMAS OPERATIVOS

Comenzaremos a programar un shell; la programación de este shell continuará en las próximas prácticas de laboratorio, es decir, en sucesivas prácticas. COMPLETAREMOS este shell: cada práctica se hará SOBRE el código de la práctica anterior, de manera que al final el shell tendrá las funcionalidades de todas las prácticas.

Un shell es básicamente es un bucle que:

- imprime un indicador.
- lee de la entrada estándar una línea de texto que incluye un comando (con sus argumentos).
- separa el comando de sus argumentos.
- procesa el comando con sus argumentos.
- si el comando es uno de los comandos internos de shell lo ejecuta, si no, asume que es un programa externo y crea un proceso que lo ejecuta.

```
while (!terminado){  
    imprimirPrompt();  
    leerEntrada();  
    procesarEntrada();  
}
```

Las funciones ``imprimirPrompt()`` y ``leerEntrada()`` pueden ser tan simples como llamadas a ``printf`` y ``fgets`` (hay una razón por la que se debe usar ``fgets()`` en lugar de ``gets()``).

El primer paso al procesar la cadena de entrada es dividirla en palabras. Para esto, la función de la biblioteca ``strtok`` resulta muy útil. Hay que tener en cuenta que ``strtok`` no asigna memoria ni copia cadenas; simplemente "divide" la cadena de entrada insertando en ella caracteres de fin de cadena (``\0``).

La siguiente función divide la cadena a la que apunta ``cadena`` (supuestamente no nula) en un array de punteros terminado en ``NULL`` (trozos). La función devuelve el número de palabras (trozos) que contenía ``cadena``. El segundo argumento a `strtok` es una cadena con los caracteres considerados separadores. `NULL` como primer argumento a `strtok` indica que se continúe "troceando" la misma cadena que en la última llamada. Nótese que "trozos" apunta a partes dentro de "cadena".

```
int TrocearCadena(char * cadena, char * trozos[])  
{  
    int i=1;  
    if ((trozos[0]=strtok(cadena, " \n\t"))==NULL)  
        return 0;  
    while ((trozos[i]=strtok(NULL, " \n\t"))!=NULL)  
        i++;  
    return i;  
}
```

Cuando el programa (el shell) no pueda realizar lo que se le pide (por cualquier motivo, por ejemplo, al intentar cambiar el directorio de trabajo actual a uno que no existe o no tienen permisos, o existe y no es un directorio), debe informar al usuario, proporcionando una descripción adecuada del error, como la que ofrecen ``strerror()`, `perror()`, `sys_errlist[errno]`, etc.`

Toda la entrada y salida en el intérprete de comandos debe realizarse a través de la entrada y salida estándar. Podemos usar el par ``read(0, ,)`` y ``write(1, ,)`` o el par ``fgets`` y ``printf``. Se desaconseja mezclar ``read`` y ``write`` con ``fgets`` y ``printf`` para la entrada/salida del intérprete de comandos.

Se suministran 4 ejemplos de código de un shell muy simple: dicho shell incluye únicamente los siguientes comandos internos:

autores: muestra los autores del shell

pwd: muestra el directorio actual del shell

chdir: cambia el directorio actual del shell

pid: muestra el pid del shell. **pid -p** muestra el de su proceso padre

pplano args: ejecuta en primer plano el programa con los argumentos que se le pasan en args

splano args: ejecuta en segundo plano el programa con los argumentos que se le pasan en args

Además, también se puede ejecutar en primer plano un programa externo simplemente tecleando su nombre junto con sus argumentos (ej: **ls -l /home**) y si queremos que sea en segundo plano, podemos añadir un **&** al final (ej: **xterm -fg yellow &**)

Version 1:

En este caso la comparación se hace mediante unas simples sentencias tipo **if**.

Para compilar el programa basta con hacer **gcc Shellv1-ifs.c**

Esto produce un ejecutable **a.out**, que podemos ejecutar mediante **./a.out**

Version 2:

Exactamente igual que la versión 1, salvo que para elegir qué hace el shell usamos la sentencia **switch**, donde previamente hemos asignado a cada comando un número (que de nuevo obtenemos con una sentencia de tipo **if**). Igual que en el caso anterior para compilarlo llega con hacer **gcc Shellv2-switch.c**, aunque siempre debemos pedir que se muestren todos los avisos (warnings) compilar, **gcc -Wall Shellv2-switch.c**

Version 3:

Hemos hecho una "mejora":

- Creamos una **struct** con dos miembros: un puntero a carácter y un puntero a función. Cada **struct** contiene dos punteros: al nombre del comando del shell y a la función que realiza dicho comando.
- Creamos un **array** de dichas **structs**, de manera que cuando el shell recibe un comando, lo va comparando con los nombres almacenados en el **array de structs**, y

cuando lo encuentra, llama a la función correspondiente (a través de su puntero).
Recuérdese que en C el nombre de una función es la dirección de dicha función.

- Con esto conseguimos librarnos de tener una función llena de sentencias **if**, que va creciendo a medida que añadimos funcionalidad al shell. Las sentencias **if**, siguen estando ahí, pero ahora están en un bucle.
- El efecto lateral es que todas las funciones deben tener el mismo prototipo, con lo que hacemos que todas reciban de argumentos el array de trozos prescindiendo el primer trozo (que es el nombre del comando), aunque no necesiten argumentos. Recuérdese que en C el nombre de un array es la dirección de dicho array, y pasar el nombre +1 es pasar el array sin el primer elemento.

Version 4.

Hemos complicado un poco el shell: en lugar de usar la llamada **execvp** que busca un ejecutable en el PATH, usamos **execv**, que recibe la trayectoria completa hasta el ejecutable (es decir, a **execvp** podemos pasarle "ls" para que ejecute el **ls**, a **execv** hemos de pasarle **/bin/ls** o **/usr/bin/ls**, dependiendo de donde esté el ejecutable de ls.

Ahora el path lo maneja el shell, como una lista (implementada como un array de punteros). Al shell le hemos añadido los comandos

path -add dir: añade un directorio al path

path -del dir: elimina un directorio del path

path -show: muestra los directorios en el path

path -clear: vacía el path

path -import: añade al path los directorios de la variable de entorno PATH

importpath: hace lo mismo que **path -import**

where file: nos dice en que directorio del path está el fichero file

Además ahora hemos dividido el programa en una serie de ficheros fuente (.c), con sus correspondientes ficheros header (.h) que pueden ser compilados separadamente.

Los ficheros header tienen la siguiente estructura

```
#ifndef NOMBREFICHERO_H
#define NOMBREFICHERO_H

/*includes necesarios para el .c*/
/*declaraciones de tipos y variables que necesita conocer el que usa las
funciones de este .c*/
/*prototipos de las funciones del .c que queremos usar desde fuera*/

#endif
```

El motivo de las sentencias **#ifndef**, **#define** y **#endif** es que si el fichero se incluye varias veces (**#include** desde distintos sitios) no de error de símbolos duplicados.

En este caso hemos hecho los siguientes ficheros

listasimple.c (junto con listasimple.h): implementa una lista simple de punteros a void (implementada como un array de punteros terminado a NULL)

path.c (junto con path.h): implementa el path como una lista de directorios. Utiliza listasimple

ejemplo.c (junto con ejemplo.h): Implementa los comandos del shell que ponemos como ejemplo. Nótese que en ejemplo.h solo figuran los prototipos de las funciones llamadas desde el shell

Shellv4.c: el shell

Para compilar el shell ahora podemos hacerlo de varias maneras

- * gcc Shellv4.c ejemplo.c path.c listasimple.c (compilamos todos los programas a la vez)

- * gcc -c listasimple.c

 - gcc -c path.c

 - gcc -c ejemplo.c

 - gcc Shellv4.c listasimple.o path.o ejemplo.o

Los .o se generan al hacer **gcc -c**, es decir, compilar sin linkar. Además, esto es muy poco óptimo pues cada vez compilamos TODO, aunque no haga falta.

En estos casos se utiliza **make**, que junto con un **Makefile** sabe qué y cuándo compilar. En el ejemplo anterior, si yo modifico algo en Shellv4.c no debería recompilarlo todo, pero si yo modifico algo en path.c, debería recompilar path.c, ejemplo.c y Shellv4.c. Un makefile tiene, aparte de posibles definiciones de variables, una estructura que se repite

target: dependencias

(tab) reglas para obtener el target a partir de sus dependencias (pueden ser varias líneas)

target, u objetivo, es lo que se quiere obtener

dependencias son los archivos de los que depende

reglas son los comandos necesarios para obtener el *target* a partir de sus dependencias.

Veamos el Makefile del ejemplo

```
CC = gcc
CFLAGS = -g -Wall

shell: Shellv4.c ejemplo.o path.o listasimple.o
    $(CC) -o shell $(CFLAGS) Shellv4.c  ejemplo.o path.o listasimple.o

ejemplo.o: ejemplo.c ejemplo.h path.h
    $(CC) -c $(CFLAGS) ejemplo.c

path.o: path.c path.h listasimple.h
    $(CC) -c $(CFLAGS) path.c

listasimple.o: listasimple.c listasimple.h
    $(CC) -c $(CFLAGS) listasimple.c

clean:
    rm shell *.o
```

En las dos primeras líneas definimos el nombre del compilador y los flags que usamos (**-g**, para debug y **-Wall** para todos los warnings). Así si queremos p.e. quitar el flag de debug, lo quitamos de la variable y no de todos los sitios donde se usa.

La primera regla establece que el ejecutable `shell` depende del archivo sin compilar `Shellv4.c` y de los compilados `practica1.o` `path.o` `listasimple.o` y se obtiene haciendo

```
gcc -o shell -g -Wall Shellv4.c  practica1.o path.o listasimple.o
```

Las otras reglas obtienen los correspondientes `.o` a partir de sus dependencias (hay maneras abreviadas de escribir esto). Finalmente en la última regla el target **clean** no depende de nada y borra el ejecutable y los `.o` generados.

Al ejecutar **make**, **make** busca en el directorio donde estamos un **Makefile** y si no se le especifica nada hace el primer target y resuelve recursivamente sus dependencias (utiliza las fechas de los archivos para saber qué compilar). También se le puede pasar explícitamente un target. P.e. **make clean** para borrar el ejecutable y los `.o` generados