Opciones para ampliar el *shell* básico
Sistemas Operativos
Grados en Ing. de Computadores, Informática y Software

- **Trabajos en modo de ejecución *respawnable***

Se trata de añadir un nuevo modo de ejecución a los *background* y *foreground* existentes que denominaremos *respawnable*. En este modo, el trabajo es lanzado en *background*, pero si muere, el *shell* se encargará de volverlo a lanzar. Tantas veces como finalice, tantas veces como será relanzado.
Se usará el símbolo "#" al final de la línea de comando para indicar este nuevo modo.

Ejemplo:
```
SO> xclock –update 1 #
```

En este caso, se abrirá una ventana *xclock* (con las opciones *–update 1*), como si se hubiera lanzado en *background*. Sin embargo, cuando este *job* finalice, por algún motivo, el shell lanzará otro proceso empleando el mismo comando y opciones.

Sugerencia: sería necesario añadir un nuevo posible valor al miembro *job_state* de la estructura *job*, informando de este nuevo modo de lanzamiento. El manejador de la señal `SIGCHLD` debe ser el encargado de chequear si un proceso está en este nuevo modo *respawnable* cuando muere para volver a lanzar el trabajo adecuadamente.

Nota: Los comandos internos bg y fg retornarían estos trabajos a modos *background* y *foreground* convencionales, por lo que pasarían a perder su *inmortalidad*.

- **Comando interno para el lanzamiento en segundo plano postergado**

Se trata de añadir un comando interno que realice el lanzamiento de un trabajo pasados N segundos. Se denominará **`after`** y se invocará de la siguiente manera:

```
SO> after 30 xclock –update 1
```

El comportamiento que se pide es: pasados los segundos especificados como argumento (30 segundos en el ejemplo anterior) se creará un trabajo en *background* con el comando (y opciones) que viene a continuación (`xclock –update 1`, en el ejemplo).

Nota: el shell no debe quedarse bloqueado esos 30 segundos (en un wait, sleep o función similar).

Sugerencias: se puede programar una alarma (ver "man 2 alarm") y realizar el lanzamiento del comando en segundo plano en el manejador de la señal SIGALRM. Como puede haber varios trabajos esperando, quizás sea necesaria una lista.

- **Redirección de la entrada/salida estándar (stdin/stdout) del programa a un fichero**

La redirección se expresará en línea de comandos con la sintaxis habitual (con '<' o '>' respectivamente). Ejemplo:

```
SO> ls > fichero.out
SO> wc < fichero.in
```

Como punto de partida se proporcionan en el campus virtual ficheros fuente con ejemplos de cómo se ejecuta un comando redireccionando su entrada/salida estándar a un fichero.

Ojo: no olvidar hacer el control de errores (los ejemplos no comprueban que exista el programa, ni los permisos deejecución, etc…) aunque no vaya incluido en los ejemplos facilitados.

- **Comunicación de dos procesos mediante un *pipe* simple**

El *pipe* se expresará en línea de comandos con la sintaxis habitual (carácter '|').

Ejemplo:

SO> ls | sort

En el campus virtual se proporcionarán ejemplos de cómo la salida estándar de un proceso se utiliza como entrada estándar de otro proceso, estableciéndose la comunicación a través de un *pipe*.

- **Comunicación de múltiples procesos mediante *pipes***

Se trata de generalizar la opción anterior a un número indeterminado de procesos comunicándose a través de pipes.

Ejemplo:

SO> ls | sort | wc | more

- **Comando interno para listar ancestros de procesos**

Se trata de crear el comando interno **`ancestors`** que dé un listado de procesos activos en el sistema (todos, no sólo los lanzados desde nuestro *shell*) mostrando para cada proceso una línea con la siguiente información:

- PID del proceso
- nombre de comando
- lista de sus ancestros, es decir, el PID de su proceso padre, abuelo, bisabuelo, … y así remontándose en sus ancestros hasta el proceso init (pid=1) .

Sugerencia: Los procesos activos pueden consultarse en `/proc`. Éste es un sistema de ficheros virtual que mantiene información para todos los procesos en ejecución y es posible

averiguar cual es el pid del padre (ppid) de cada proceso usando la información de `/proc` (ver "**man 5 proc**").

Ejemplo:

```
SO> ancestors
1 (init):0
2 (kthreadd): 0
3 (migration): 1
…
28996 (bash): 28987, 28957, 28955,  15425, 1
…
```

- **Historial de comandos**

Mantener el historial de comandos tecleados para poder ser reutilizados. El comando interno "historial" mostrará la lista de comandos tecleados junto con un índice para cada uno. Si el comando "historial" va acompañado de un número ("historial n"), se volverá a ejecutar el comando que ocupe la posición indicada por dicho número en el historial.

Ejemplo:

```
SO> historial
1     ls
2     ps
3     cd ..
4     ls

SO> historial 1
ls
Descargas/   Escritorio/ Librería/ léeme.txt

SO>
```

Las teclas de cursor arriba y abajo nos permitirán desplazarnos por el historial comando a comando, mostrando el comando correspondiente en la línea de entrada de comandos. Se deberá poder editar cualquier comando recuperado de esta forma (desplazamiento del cursor, inserción y borrado de caracteres, …)

Información sobre el manejo del terminal a bajo nivel:

Esta información es útil para el posicionamiento del cursor, navegación por el historial y cambio de colores sin la utilización de librerías, sólo usando secuencias de escape ANSI.

1. Aquí se presentan las secuencias de escape ANSI para cambiar de color el texto de salida en la consola (funcionará solo con terminales con soporte ANSI):

```
#define ROJO "\x1b[31;1;1m"
#define NEGRO "\x1b[0m"
#define VERDE "\x1b[32;1;1m"
#define AZUL "\x1b[34;1;1m"
#define CIAN "\x1b[36;1;1m"
#define MARRON "\x1b[33;1;1m"
#define PURPURA "\x1b[35;1;1m"
```

Ejemplo de uso:

```
printf("%s Demostración %sde %scolor %s", ROJO, VERDE, AZUL, NEGRO);
```

Es recomendable poner el negro como último color para evitar escribir en un color distinto al negro por teclado.

2. Lectura de pulsaciones de teclado sin esperar a “\n” ni ECHO.

```
/*============================================================*/
/* lee un caracter sin esperar a '\n' y sin eco*/
#include <stdio.h>
#include <termios.h>
#include <unistd.h>

char getch(){
  int shell_terminal = STDIN_FILENO;
  struct termios conf;
  struct termios conf_new;
  char c;
  /* leemos la configuracion actual */
  tcgetattr(shell_terminal,&conf);
  conf_new=conf;
  /* configuramos */
  conf_new.c_lflag&=(~ICANON); // sin buffer
  conf_new.c_lflag&=(~ECHO); // sin eco
  conf_new.c_cc[VTIME]=0;
  conf_new.c_cc[VMIN]=1;
  /* guardamos la configuracion */
  tcsetattr(shell_terminal,TCSANOW,&conf_new);
  /* leemos el caracter */
  c=getc(stdin);
  /* restauramos la configuracion */
  tcsetattr(shell_terminal,TCSANOW,&conf);
  return c;
}
```

3. Interpretación de las teclas de cursor, borrar, etc..

/* Los cursores devuelven una secuencia de 3 caracteres, 27 - 91 - (65, 66, 67 ó 68) */

```
   int text;
  text = getch();
  if(text == 27){
      text = getch();
      if(text == 91){
          text = getch();
          if(text == 65){
              /* ARRIBA */ …
          }else if(text == 66){
              /* ABAJO */ …
          }else if(text == 68){
              /* IZQUIERDA */ …
          }else if(text == 67){
              /* DERECHA */ …
          }else{
              …
          }
      }
  }else if(text == 127){
      /* BORRAR */ …
  }else if …{
…
  }
```

4. Movimiento del cursor para escribir en cualquier parte de la pantalla.

Secuencias de escape ANSI para el movimiento y posicionamiento del cursor

- Posicionar el cursor en la línea L, columna C: `\033[*/L/*;*/C/*H`

- Mover el cursor arriba N líneas: `\033[*/N/*A`

- Mover el cursor abajo N líneas: `\033[*/N/*B`

- Mover el cursor hacia adelante N columnas: `\033[*/N/*C`

- Mover el cursor hacia atrás N columnas: `\033[*/N/*D`

- Guardar la posición del cursor: `\033[s`

- Restaurar la posición del cursor: `\033[u`

Ejemplo para escribir el carácter “A” en la posición 10,10:

```
printf("\033[*/10/*;*/10/*H A");
```