Opciones para ampliar el *shell* básico
Sistemas Operativos

A continuación, se describen una serie de mejoras y ampliaciones que pueden ser añadidas a las especificaciones de la práctica.

SOLO se valorarán las mejoras que hay descritas a continuación. Cualquier otra ampliación que el estudiante desee añadir para que le sea valorada en la calificación debe ser consultada con el/la profesor/a con antelación.

- **Comando interno para 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 (y separado por un espacio en blanco del último argumento o comando) para indicar este nuevo modo.

Ejemplo:

```
S0> xclock –update 1 +
```

En este caso, se abrirá una ventana *xclock* (con el argumento –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*.

- **Limitación del tiempo de vida de un trabajo basado en *threads***

Se trata de añadir un comando interno que realice el lanzamiento de un trabajo, de manera que pasados N segundos, el trabajo en cuestión sea aniquilado (*killed*) si no terminó aún. Para ello el *shell* creará un *thread* auxiliar que dormirá esos N segundos, enviando a continuación la señal SIGKILL al proceso en cuestión.

El nuevo comando se denominará **alarm-thread** y se invocará de la siguiente manera:

```
SO> alarm-thread 30 xclock –update 1
```

El comportamiento que se pide es:

- se lanzará el proceso **xclock** con los argumentos indicados;
- el *shell* creará un *thread* que dormirá (`sleep()`) los segundos especificados como primer argumento (30 segundos en el ejemplo), tras lo cual aniquilará (SIGKILL) el proceso lanzado (`xclock –update 1`, en el ejemplo) si éste aún no hubiera terminado;
- dicho *thread* terminará ahí (lo recomendable es que el *thread* esté declarado como *detached* para que libere sus recursos sin más espera).

El comando interno `alarm-thread` funcionará tanto si el comando a lanzar es en primer plano como en segundo (línea acabada en &).

Observa que debe de crearse un *thread* de alarma por cada *job* lanzado de esta manera, es decir, que si se lanzan varios `alarm-thread` en *background* debe haber uno por cada *job*.

- **Ejecución postergada de un comando**

Añadir comando interno **delay-thread** que recibe un número de segundos (s) y un comando con argumentos (cmd):

```
delay-thread <s> <cmd [with arguments]>
```

ejecutará en *background* el comando (con sus argumentos) dentro de <s> segundos, o sea, el shell esperará <s> segundos antes de ejecutar el comando en *background* y durante ese tiempo mostrará el *prompt* y atenderá normalmente las peticiones que lleguen por la línea de comandos.

Por ejemplo:

```
SO> delay-thread 30 xeyes
SO> delay-thread 0 xclock
```

Si tecleamos los comandos anteriores suficientemente rápido (sin dejar pasar 30 segundos entre ellos), `xclock` se lanzará antes que `xeyes`.
`Indicar cero segundos ejecutará el comando en *background* sin esperar. Si no se indican segundos, o sea, el argumento tras delay no se puede interpretar como un número entero, o bien es un número negativo, no se ejecutará el comando y se mostrará un mensaje de error.`
`Como en la opción anterior (alarm-thread) se implementará esta nueva funcionalidad haciendo uso de ***threads***.`

- **Enmascaramiento de señales**

Se pide implementar un comando interno **`mask`** que recibe una secuencia de números de señal (s) y un comando con argumentos (cmd):

```
mask <s> -c <cmd [with arguments]>
```

ejecutará en *foreground* el comando (con sus argumentos) con las señales indicadas en la secuencia de números <s> enmascaradas. En principio, si el proceso no modifica su máscara, estas quedarán enmascaradas.

Por ejemplo:

```
S0> mask 2 14 -c xeyes -fg red
```

ejecutará en *foreground* el comando xeyes (con sus argumentos) con la señal SIGINT (señal número 2) y la señal SIGALRM (señal número 14) enmascaradas, es decir, cuando se pulse ^C (que envía dicha señal) el comando xeyes no ejecutará los manejadores de dichas señales, o sea, no terminará.

En caso de error de sintaxis se indicará un mensaje indicándolo y no se ejecutará el comando. Ejemplos de errores de sintaxis son: no preceder el nombre del comando con **`-c`**, no indicar ningún número de señal, que el argumento que representa el número de señal no sea un entero positivo, etc.

- **Ejecución condicional de un proceso tras la finalización de otro**

Se dispondrá de dos operadores, && (se lee *and*) y || (se lee *or*) para la separación de dos comandos. Se ejecutará el primer comando en *foreground* y trás la finalización del mismo se comprobará si ha terminado con éxito (terminación normal y valor de retorno cero).
Con el operando && (*and*), sólo se ejecutará el segundo comando en *foreground* si el primero terminó con éxito, mientras que con el operando || (or) sólo se ejecutará el segundo operando si el primero no acaba con éxito (retorna en el *status* un valor diferente de cero, muere por una señal, ...)
Obsérvese en los siguientes ejemplos que cada uno de los dos comandos pueden tener sus propios argumentos:

```
S0> xeyes -fg red || xclock -update 1
```

Si cerramos xeyes pulsado la (x) en la ventana (terminación normal) no se ejecutará `xclock`, pero si lo terminamos enviándole una señal, p.ej.: `killall xeyes` desde otra ventana, entonces se ejecutará `xclock`.

```
S0> mkdir testdir && touch testdir/testfile
```

En este ejemplo, si falla `mkdir` al intentar crear el directorio no se ejecutará el comando `touch`.
Se pide implementar **los dos operandos** para tener en cuenta esta ampliación.

- **Múltiple ejecución condicional**

Es una generalización del caso anterior donde hay una secuencia de comandos separados por cualquiera de los dos operadores && y ||.
Se lanza un comando si el proceso del comando anterior acaba con éxito y está separados por && (and) o no acaba con éxito y están separados por || (or). O lo que es lo mismo, si un proceso acaba con éxito y va seguido de || (or) se ignora el resto. Si un proceso no acaba con éxito y va seguido de && (and) también se ignora el resto.
Nota: es una **simplificación**, no se corresponde con lo que ocurre en sh o bash.

```
S0> stat testdir && tree || ls -l
```

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

El *pipe* se expresará en línea de comandos con la sintaxis habitual utilizada en bash (operador pipe con el carácter |). Observa que cada uno de los dos comandos pueden tener sus propios argumentos.

Los dos procesos conectados con el pipe, se considerarán un único trabajo (*job*) por lo que deberán pertenecer al mismo grupo de procesos y aparecerán en única línea del comando `jobs` que mostrará el nombre de los dos comandos conectados. Obsérvese que será necesario mantener en la estructura *job* la lista de los procesos conectados por el pipe para realizar el *waitpid* apropiadamente y que no quede ningún proceso en estado *zombie*.

He aquí algunos ejemplos de uso del operador |:

```
S0> ls | sort
S0> ls -l -a | wc –l &
S0> cat /etc/passwd | grep –i root
```

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*. El operador | se separará por espacios en blanco en la línea de comandos para simplificar el parseo de la línea.

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

Se trata de generalizar la opción anterior a un número indeterminado de procesos comunicándose a través de pipes. Ten en cuenta que cada comando puede tener sus argumentos y que todos los procesos conectados entre sí forman un único *job*.

Ejemplos de *pipe* múltiple:

```
S0> cat /proc/cpuinfo | grep processor | wc -l
S0> ls -la | sort -k 2 | grep . | less
```

- **Historial de comandos**

Mantener el historial de comandos tecleados para poder ser reutilizados.

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, ...)

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/ Libreria/ leeme.txt

SO>
```

Apéndice: 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 RESET     "\x1b[0m"
#define RED       "\x1b[31m"
#define GREEN     "\x1b[32m"
#define YELLOW    "\x1b[33m"
#define BLUE      "\x1b[34m"
#define MAGENTA   "\x1b[35m"
#define CYAN      "\x1b[36m"
#define WHITE     "\x1b[37m"
#define DEFAULT   "\x1b[39m"
```

Ejemplo de uso:

```
printf(RED"Demostración %sde %scolor"RESET, GREEN, BLUE);
```

Es recomendable poner el `RESET` como último color para evitar dejar el último color usado en el terminal al finalizar el comando interno.
Prueba en línea de comandos: `echo -e " \x1b[31mHOLA"`

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

```
/*============================================================*/
/* lee un caracter sin esperar a '\n' y sin eco */
/* retorna EOF en caso de error o con ^D */
#include <stdio.h>
#include <termios.h>
#include <unistd.h>

int getch()
{
  struct termios conf, conf_new;
  int c;

  tcgetattr(STDIN_FILENO, &conf);        /* leemos la configuracion actual */
  conf_new = conf;

  conf_new.c_lflag &= (~(ICANON|ECHO));  /* configuramos sin buffer ni eco */
  conf_new.c_cc[VTIME] = 0;
  conf_new.c_cc[VMIN]  = 1;

  tcsetattr(STDIN_FILENO, TCSANOW, &conf_new); /* establecer configuracion */

  c = getchar();                                    /* leemos el caracter */

  tcsetattr(STDIN_FILENO, TCSANOW, &conf); /* restauramos la configuracion */
  return c;
}
```

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

```
/* Las teclas de cursor devuelven una secuencia de 3 caracteres,
27 --   91 --   (65, 66, 67 ó 68) */

int sec[3];
sec[0] = getch();
switch (sec[0])
{
      case 27:
             sec[1] = getch();
             if (sec[1] == 91) // 27,91,...
             {
                  sec[2] = getch();
                  switch (sec[2])
                  {
                        case 65: /* ARRIBA */
                                   break;
                        case 66: /* ABAJO */
                                   break;
                        case 67: /* DERECHA */
                                   break;
                        case 68: /* IZQUIERDA */
                                   break;
                        default:
                  }
             }
             break;
      case 127: /* BORRAR */
             break
      default:
        ...
}
```

4. Movimiento del cursor en el terminal para poder elegir línea y columna donde escribir. 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");
```