Opciones para ampliar el *shell* básico
Sistemas Operativos

Grupos C - Grados en Ing. Computadores, Informática y Software - 2017/18

- **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 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*.

- **Comando interno para el lanzamiento de trabajos con time-out**

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.
Se denominará **`time-out`** y se invocará de la siguiente manera:

```
SO> time-out 30 xclock –update 1
```

El comportamiento que se pide es: se lanzará el proceso **`xclock`** con los argumentos indicados, y pasados los segundos especificados como primer argumento (30 segundos en el ejemplo) se aniquilará (SIGKILL) el proceso lanzado (`xclock –update 1`, en el ejemplo) si éste aún no hubiera terminado.
Nota: si se lanzara en *background*, el shell no debe quedarse bloqueado esos 30 segundos (por ejemplo, en un wait, sleep o función similar).

Sugerencias: se puede programar una alarma (ver "man 2 alarm") y realizar el lanzamiento del *kill* correspondiente haciendo uso del manejador de la señal SIGALRM. Como puede haber varios trabajos esperando, quizás sea necesaria una lista. Otra alternativa que se podría plantear es disponer de un *thread* auxiliar que se encargue de monitorizar el tiempo para cada uno de los procesos ejecutados con **`time-out`**.

- **Enmascaramiento de señales**

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

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

ejecutará en *foreground* el comando (con sus argumentos) con la señal número s indicada enmascarada. En principio si el comando no modifica su máscara quedará enmascarada.

Por ejemplo:

```
SO> mask 2 -c xeyes -fg red
```

ejecutará en *foreground* el comando `xeyes` (con sus argumentos) con la señal SIGINT (señal número 2) enmascarada, es decir, cuando se pulse ^C (que envía dicha señal) el comando `xeyes` no terminará.

En caso de error de sintaxis se indicará un mensaje indicándolo y no se ejecutará nada. 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.

Como mejora, se pueden poner una secuencia de números de señal separados por espacios (observa que en la sintaxis propuesta el comando va precedido del switch "-c "), por ejemplo:

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

ejecutaría `xeyes` enmascarando las señales 2 (SIGINT) y 14 (SIGALARM).

- **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.

¡Cuidado!: no olvidar hacer el control de errores (los ejemplos no comprueban que exista el programa, ni los permisos de ejecución, 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 que saque estadísticas acerca de los procesos hijos y *threads***

Se trata de crear el comando interno **`children`** 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
- número de procesos hijos
- número de *threads* del proceso

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 coger toda la información de un proceso con un **`pid`** dado a partir de la información situada en `/proc/<pid>` (ver "**man 5 proc**").

Ejemplo:

```
SO> children
PID    COMMAND    #CHILDREN    #THREADS
1      init       142          1
2      kthreadd   84           1
…
28996  firefox    2            12
…
```

- **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 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(ROJO"Demostración %sde %scolor"NEGRO, VERDE, AZUL);
```

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;

  tcgetattr(shell_terminal,&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(shell_terminal,TCSANOW,&conf_new); /* establecer configuracion */

  c = getc(stdin); /* leemos el caracter */

  tcsetattr(shell_terminal,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) */
char 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");
```