Software de sistemas y de comunicaciones (EI1058)
Grado en Ingeniería Informática

Práctica 1. Repaso de llamadas al sistema de Unix

1 Objetivos

Esta primera práctica, a la que dedicaremos una sesión de laboratorio, tiene los dos siguientes objetivos:

- Recordar lo que has estudiado en cursos anteriores acerca de la utilización de llamadas al sistema operativo UNIX.
- Facilitar la implementación del mini-intérprete de comandos que se propone en la segunda práctica.

Para ello, realizaremos un conjunto de programas que irán incrementando progresivamente su grado de complejidad. En ellos se utilizarán las siguientes funciones de librería en lenguaje C:

- Relacionadas con la gestión de procesos: `fork`, `execlp` y `execvp`.
- Relacionadas con la comunicación entre procesos: `pipe`, `dup`, `wait` y `exit`.
- Relacionadas con la gestión de ficheros: `creat`, `open`, `close`, `read` y `write`.
- Relacionadas con la gestión del entorno de un proceso: `putenv`, `setenv` y `getenv`.

2 Trabajo a desarrollar

Recuerda que puedes aprovechar mejor el tiempo de las sesiones de prácticas si **te la preparas previamente antes de acudir al laboratorio**.

1. Los ejercicios de los **entregables ET1 y EA1**, que están accesibles en el Aula Virtual, son una buena manera de comenzar con el repaso de llamadas al sistema. Su realización te permitirá abordar mejor los ejercicios que aparecen a continuación. Lee también detenidamente la información que aparece en el Aula Virtual a partir del enlace "***Lo que no debes olvidar sobre las llamadas al sistema***". En ella encontrarás conceptos básicos importantes sobre estas que evitarán que comentas errores muy habituales.

2. Realizar un programa que permita la ejecución del comando `ls -l` aplicado sobre el fichero pasado como segundo argumento, redireccionando la salida estándar a un fichero dado como primer argumento. Si se produce algún fallo en la invocación de dicho comando, se mostrará por la salida de error estándar un mensaje de error. Asegúrate de que el programa funciona tanto si existe el fichero sobre el que se vuelca la salida estándar como si no existe.
   Ejemplo:

   ```
   % prg2 f_salida fich
   ```

   ejecuta el comando `ls -l fich > f_salida`.

3. Desarrollar un programa que permita la ejecución de un comando (dado a partir del segundo argumento) redireccionando la salida estándar a un fichero (pasado como primer argumento).
   Ejemplo:
   ```
   % prg3 f_salida ls -l *.c
   ```
   ejecuta el comando `ls -l *.c > f_salida`.

4. Modificar el programa anterior para que permita la ejecución de un comando (dado a partir del tercer argumento) redireccionando (previamente) la salida estándar a un fichero (pasado como primer argumento) y la salida de error a un fichero (pasado como segundo argumento).
   Ejemplo:
   ```
   % prg4 f_salida f_error ls -l
   ```
   ejecuta el comando `ls -l > f_salida 2> f_error`.

5. Modificar el programa anterior para que redireccione la salida estándar, de forma que, si el fichero pasado como primer argumento existe, se vuelque el contenido de ésta al final de dicho fichero o que, en caso contrario, lo cree. La salida del error ha de estar redireccionada teniendo en cuenta que, si existe el fichero pasado como segundo argumento, se sobreescribirá (se "machacará") su contenido.

6. Desarrollar un programa que cree un proceso que devuelva un código de terminación que se pasa como argumento a dicho programa. El proceso padre deberá escribir dicho código de terminación por su salida estándar. En las transparencias del tema 1 aparece un ejercicio similar a este.

7. Modificar el programa anterior para que el código de terminación del proceso hijo sea el código de terminación del programa. Para comprobar el funcionamiento del programa ejecuta (desde el intérprete de comandos) el comando `echo $?` justo al finalizar la ejecución del programa. Dicho comando muestra el código de terminación del programa o comando que se ha ejecutado inmediatamente antes.

8. Realizar un programa que cree un proceso hijo que lleve a cabo las operaciones especificadas en el ejercicio 5. Además, el código de terminación de dicho hijo será el código de terminación del programa.

9. En el apéndice I aparece un programa que ejecuta la siguiente línea de comandos:

   ```
   ps -l | sed -e '1,$p' | sort | uniq
   ```

   siendo todos los procesos creados, procesos hijos del programa principal. Comprueba la jerarquía de procesos creada analizando la salida de ejecución del programa (esto es, los identificativos de proceso del padre y del hijo –PPID y PID, respectivamente– asociados a cada proceso creado). Modificar dicho programa para que siga generando tres tuberías pero utilizando para ello únicamente dos variables del tipo `int[2]`. ¿Es necesario añadir alguna función `wait` para sincronizar la correcta finalización de los procesos? ¿Por qué?

10. Desarrollar un programa que ejecute la siguiente línea de comandos:
    ```
    ps -l | sed -e '1,$p' | sort | uniq
    ```
    siendo cada uno de los procesos hijo del proceso que ejecuta el comando de su derecha y siendo el proceso que ejecuta el comando `uniq` hijo del proceso que ejecuta el programa principal. Comprueba la jerarquía de procesos creada analizando la salida de ejecución del programa. ¿Cuántas funciones `wait` son necesarias en el proceso más alto de la jerarquía para sincronizar su correcta finalización? ¿Y en los restantes procesos?

11. Desarrollar un programa que muestre por pantalla el valor de la variable de entorno PATH, que la modifique para que su valor sea `/usr/bin:/sbin` y que compruebe el cambio. Comprobar desde el intérprete de comandos el valor de dicha variable al acabar la ejecución del programa. ¿Es el esperado? ¿Por qué?

12. Escribe un programa en C que muestre por pantalla las tablas de multiplicar. En primer lugar se mostrará la tabla completa del número 1, a continuación la del 2 y así sucesivamente hasta llegar a la tabla del número que se pasa como (único) argumento al programa. Para ello el programa creará *tantos procesos como tablas menos uno. Y todos los procesos se ejecutarán en paralelo en el sistema.* Cada proceso imprimirá su número de identificación (su PID) y, a continuación, la tabla que le corresponda. *La sincronización entre los procesos se realizará mediante tuberías, siguiendo un esquema similar a los empleados en los ejercicios 3 al 7 del entregable EA1.* **Justifica cuáles de esos esquemas podrían ser válidos para este ejercicio y cuáles no, así como la razón por la que elijes el esquema que propones para tu ejercicio.** Esta misma idea te servirá también para implementar el *minishell* de la próxima práctica.

    No se considerarán válidas aquellas implementaciones que utilicen una única tubería o vectores de tuberías para sincronizar los procesos. Tampoco se aceptarán soluciones que impliquen espera activa, esto es, soluciones en las que un proceso está leyendo continuamente números de una tubería hasta que encuentra el número correspondiente a la tabla que le corresponde imprimir.

    La justificación de los esquemas válidos para resolver este ejercicio y la implementación de este son objetivos imperdonables del tema 1 de la asignatura.

3 Evaluación

Para evaluar la práctica únicamente has de entregar los dos ejercicios que se especifican más abajo. Pero los demás ejercicios también son importantes y es altamente recomendable que los hagas todos. Cada uno de ellos tiene un propósito y te ayudará a implementar una parte del mini-intérprete de comandos que se propone en la segunda práctica.

Debéis entregar el código de los **ejercicios 8 y 12**. Pero antes de ello, revisad si la solución que habéis dado para estos ejercicios se ajusta a los criterios que seguiré para evaluarlos. Estos criterios de evaluación aparecen en unas tablas que encontrarás en el apartado del Aula Virtual destinada a la práctica 1. Estas tablas os permitirán supervisar y mejorar la calidad de vuestro trabajo antes de entregarlo.

A continuación, dejad en el **Aula Virtual** el código de los **ejercicios 8 y 12** y las tablas con los criterios de evaluación rellenadas. Las soluciones a los ejercicios deberán estar en sendos ficheros de texto. No olvidéis poner el nombre tanto en el fichero con las tablas de autoevaluación como en cada uno de los ficheros que contienen vuestras soluciones a los ejercicios (en una línea comentada al inicio de estos). **Es imprescindible dejar en el Aula Virtual el fichero que contiene las tablas con los criterios de evaluación rellenadas para que os corrija los ejercicios que habéis entregado.**

El **plazo máximo de entrega** es el domingo de la semana dedicada a esta práctica.

Apéndice I

A continuación se muestra el código de un programa que ejecuta la siguiente línea de comandos:

```
ps -l | sed -e '1,$p' | sort | uniq
```

```
#include <fcntl.h>
#include <unistd.h>

int main()
{ int estado, t1[2], t2[2], t3[2];

pipe(t1);
if (fork()!=0)
{ pipe(t2);
  if (fork()!=0)
  { pipe(t3);
    if (fork()!=0)
    { /* Código del padre */
        close(0); dup(t3[0]); close(t3[0]); close(t3[1]);
        close(t2[0]); close(t2[1]);
        close(t1[0]); close(t1[1]);
        execlp("uniq","uniq",NULL);
        perror("Error en el uniq");
        exit(-1);
    } else { /* Código del tercer hijo */
        close(0); dup(t2[0]); close(t2[0]); close(t2[1]);
        close(1); dup(t3[1]); close(t3[1]); close(t3[0]);
        close(t1[0]); close(t1[1]);
        execlp("sort","sort",NULL);
        perro("Error en el sort");
        exit(-1);
    }
  } else { /* Código del segundo hijo */
      close(0); dup(t1[0]); close(t1[0]); close(t1[1]);
      close(1); dup(t2[1]); close(t2[1]); close(t2[0]);
      execlp("sed","sed","-e","1,$p",NULL);
      perror("Error en el sed");
      exit(-1);
  }
} else { /* Código del primer hijo */
    close(1); dup(t1[1]); close(t1[1]); close(t1[0]);
    execlp("ps","ps","-l",NULL);
    perror("Error en el ps");
    exit(-1);
}
}
```