**SISTEMAS OPERATIVOS**

Grado en Informática. Curso 2014-2015

**Práctica 1**

Realizar un intérprete de comandos sencillo (shell) en UNIX. El intérprete debe comprender los siguientes comandos que se muestran a continuación.

**fin** Termina la ejecución del intérprete de comandos.

**exit** Termina la ejecución del intérprete de comandos.

**quit** Termina la ejecución del intérprete de comandos.

**autores** Muestra los nombres y los logins de los autores del shell.

**pid [-p]** Muestra el pid del proceso que ejecuta el shell (con -p muestra también el del proceso padre del shell)

**cd [dir]** Cambia el directorio actual del shell a *dir*. Si no se especifica *dir*, muestra el directorio actual del shell.

**list [-s] [-a] [dir]** Lista los ficheros del directorio *dir*, excluyendo los archivos ocultos (aquellos cuyo nombre comienza por .). Para cada fichero nos dará una línea con la información en el formato análogo a como lo hace `ls -li`. Si no se especificase *dir* se listará el directorio actual. La opción *-s* indica listado corto, es decir, para cada fichero se listará solo su nombre. La opción *-a* indica que se listen también los archivos cuyo nombre comienza por **.**

**delete fildir** Elimina el archivo o directorio fildir (si es un directorio se supone vacio).

**deltree dir** Elimina el directorio *dir* junto con todos los ficheros y directorios que contenga.

**priority [pid [valor]]**. Si se especifican dos argumentos (*pid y valor*) se cambiará la prioridad del proceso *pid* a *valor* (utilizando *setpriority*). Si solo se especifica un argumento se entenderá que es el *pid* y nos mostrará la prioridad de dicho proceso. Si no se suministran argumentos, nos mostrará la prioridad del shell

**fork** El shell crea un hijo y se queda en espera a que ese hijo termine. (El hijo continúa ejecutando el código del shell)

**exec prog arg1 . . .** Ejecuta, sin crear proceso (**es decir REEMPLAZANDO el código del shell**) el programa *prog* con sus argumentos. *prog* representa un ejecutable externo y puede llevar argumentos.

**splano prog arg1 . . .** El shell crea un proceso que ejecuta en **segundo plano** el programa *prog* con sus argumentos. *prog* representa un ejecutable

externo. **Además añadirá el proceso a la lista de procesos en segundo plano del shell**

**pplano prog arg1 ...** El shell crea un proceso que ejecuta en **primer plano** el programa *prog* con sus argumentos. *prog* representa un ejecutable externo.

**jobs** [**all|term|sig|stop|act**] Muestra la lista de procesos en segundo plano del shell. Para cada proceso debe mostrar (en una sola línea) su pid, su prioridad, la línea de comando con que se lanzó (ejecutable más argumentos), el instante de inicio y su estado (activo, terminado normalmente, parado o terminado por señal) indicando, en su caso, el valor devuelto o la señal causante de su terminación o parada. Si se indica *jobs* sin argumentos se listarán todos. **ESTE COMANDO ACCEDE A LA LISTA DE PROCESOS EN SEGUNDO PLANO QUE LLEVA EL SHELL (no tiene que recorrer el directorio /proc ni nada por el estilo)**

- **jobs** Muestra todos los procesos
- **jobs all** Muestra todos los procesos
- **jobs term** Muestra los procesos terminados normalmente
- **jobs sig** Muestra los procesos terminados por señal
- **jobs stop** Muestra los procesos parados
- **jobs act** Muestra los procesos activos

**jobs pid** Muestra la información del proceso de pid *pid*.(nótese que es otro uso del comando *jobs*)

**clearjobs** Elimina de la lista de procesos de segundo plano del shell aquellos que han terminado (normalmente o debido a una señal)

**??????** cualquier otro nombre se entenderá que es un programa con sus argumentos y se ejecutará, exactamente igual que con `pplano`

Nótese que los comandos aquí descritos deben interpretarse de la siguiente manera:

- Los argumentos entre corchetes [] son opcionales.
- Los argumentos separados por `|` indican que debe ir uno u otro, pero no ambos simultaneamente.
- El intérprete de comandos debe aceptar y entender la sintaxis aquí propuesta, pero no tiene que forzarla; por ejemplo, si hay varios argumentos deben aceptarse en el orden especificado, pero puede resultar más cómodo de programar asumiendo que pueden ir en cualquier orden.
- La implementación de la lista de procesos en segundo plano del shell

es libre

Además deben tenerse en cuenta las siguientes indicaciones:

- **En ningún caso debe producir un error de ejecución (segmentation, bus error ...). La práctica que produzca un error en tiempo de ejecución no será puntuada.**
- No debe dilapidar memoria (ejemplo: variable que se asigna cada vez que se llama a una función y no se libera). **NO SE REFIERE A DECLARAR LOS ARRAYS DE TAMAÑO PEQUEÑO** (puede utilizarse *valgrind* para detectar errores de memoria)
- Cuando el shell no pueda ejecutar una acción por algún motivo, debe indicarlo con un mensaje como el que se obtiene con `sys_errlist[errno]` o con *perror()* (por ejemplo, si no puede cambiar de directorio debe indicar por qué).

**AYUDAS**

Un shell es basicamente un bucle que realiza lo siguiente

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

*imprimirPrompt()* y *leerEntrada()* pueden ser algo tan sencillo como sendas llamadas a `printf` y `gets`

El primer paso para procesar la entrada es trocear la cadena de entrada. Para trocear la cadena de entrada es muy cómodo usar `strtok`. Téngase en cuenta que `strtok` ni asigna memoria ni copia cadenas, solo pone caracteres de fin de cadena en determinados sitios de la cadena de entrada. La siguiente función trocea la cadena de entrada (se supone que no recibe un puntero NULL) en un array de punteros terminado a NULL (el último puntero del array es NULL). Nos devuelve el número de trozos

```
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;
}
```

Para convertir los permisos de un fichero a la cadena que los representa, puede utilizarse cualquiera de estas funciones, téngase en cuenta que *Convierte-Modo2* es la versión con asignación estática de memoria y *ConvierteModo3* es la versión con asignación dinámica de memoria

```
char TipoFichero (mode_t m)
{
  switch (m&S_IFMT) { /*and bit a bit con los bits de formato,0170000 */
    case S_IFSOCK:  return 's';  /*socket */
    case S_IFLNK:   return 'l';    /*symbolic link*/
    case S_IFREG:   return '-';   /* fichero normal*/
    case S_IFBLK:   return 'b';   /*block device*/
    case S_IFDIR:   return 'd';   /*directorio */
    case S_IFCHR:   return 'c';   /*char  device*/
    case S_IFIFO:   return 'p';   /*pipe*/
    default: return '?';   /*desconocido, no deberia aparecer*/
  }
}

char * ConvierteModo (mode_t m, char *permisos)
{
  strcpy (permisos,"---------- ");

  permisos[0]=TipoFichero(m);
  if (m&S_IRUSR) permisos[1]='r';  /*propietario*/
  if (m&S_IWUSR) permisos[2]='w';
  if (m&S_IXUSR) permisos[3]='x';
  if (m&S_IRGRP) permisos[4]='r';   /*grupo*/
  if (m&S_IWGRP) permisos[5]='w';
  if (m&S_IXGRP) permisos[6]='x';
  if (m&S_IROTH) permisos[7]='r';   /*resto*/
  if (m&S_IWOTH) permisos[8]='w';
  if (m&S_IXOTH) permisos[9]='x';
  if (m&S_ISUID) permisos[3]='s';  /*setuid, setgid y stickybit*/
  if (m&S_ISGID) permisos[6]='s';
  if (m&S_ISVTX) permisos[9]='t';
```

```
  return permisos;
}

char * ConvierteModo2 (mode_t m)
{
  static char permisos[12];
  strcpy (permisos,"---------- ");

  permisos[0]=TipoFichero(m);
  if (m&S_IRUSR) permisos[1]='r';  /*propietario*/
  if (m&S_IWUSR) permisos[2]='w';
  if (m&S_IXUSR) permisos[3]='x';
  if (m&S_IRGRP) permisos[4]='r';   /*grupo*/
  if (m&S_IWGRP) permisos[5]='w';
  if (m&S_IXGRP) permisos[6]='x';
  if (m&S_IROTH) permisos[7]='r';   /*resto*/
  if (m&S_IWOTH) permisos[8]='w';
  if (m&S_IXOTH) permisos[9]='x';
  if (m&S_ISUID) permisos[3]='s';  /*setuid, setgid y stickybit*/
  if (m&S_ISGID) permisos[6]='s';
  if (m&S_ISVTX) permisos[9]='t';
  return (permisos);
}

char * ConvierteModo3 (mode_t m)
{
  char * permisos;
  permisos=(char *) malloc (12);
  strcpy (permisos,"---------- ");

  permisos[0]=TipoFichero(m);
  if (m&S_IRUSR) permisos[1]='r';  /*propietario*/
  if (m&S_IWUSR) permisos[2]='w';
  if (m&S_IXUSR) permisos[3]='x';
  if (m&S_IRGRP) permisos[4]='r';   /*grupo*/
  if (m&S_IWGRP) permisos[5]='w';
  if (m&S_IXGRP) permisos[6]='x';
  if (m&S_IROTH) permisos[7]='r';   /*resto*/
  if (m&S_IWOTH) permisos[8]='w';
  if (m&S_IXOTH) permisos[9]='x';
  if (m&S_ISUID) permisos[3]='s';  /*setuid, setgid y stickybit*/
  if (m&S_ISGID) permisos[6]='s';
  if (m&S_ISVTX) permisos[9]='t';
```

```
  return (permisos);
}
```

Para crear un proceso se usa *fork()*. *fork()* crea un proceso que es una copia del proceso que llama a *fork()*, es decir, después de dicha llamada hay dos procesos iguales, la única diferencia es el valor devuelto por *fork()* (0 al hijo y el pid del hijo al padre). La llamada *waitpid* permite a un proceso esperar a que un proceso hijo termine. Por ejemplo, el código que se muestra a continuación crea un proceso hijo que ejecuta *funcion2()* mientras el padre ejecuta *funcion1*. Cuando el hijo ha terminado de ejecutar *funcion2()* termina y SOLO el padre ejecuta *funcion3()*

```
.......
if ((pid=fork())==0) {
    funcion2();
    exit(0);
    }
else {
   funcion1();
   waitpid(pid,NULL,0);
   funcion3();
   }
```

dado que la llamada *exit()* hace que un proceso termine, el código anterior podría escibirse así

```
.......
if ((pid=fork())==0) {
    funcion2();
    exit(0);
    }
funcion1();
waitpid(pid,NULL,0);
funcion3();
```

En el siguiente ejemplo, tanto el padre como el hijo ejecutarían *funcion3()*

```
.......
if ((pid=fork())==0)
    funcion2();
else
    funcion1();
```

```
funcion3();
```

Para que un proceso ejecute un programa se utilizará la llamada *execvp()* que busca los ejecutables en el PATH. Nótese que *execvp()* solo devuelve valor en caso de error; si no se produce error, execvp() reemplaza el código del proceso y no se ejcutará la siguiente línea.

```
......
execl("/bin/ls","ls","-l","/usr",NULL)
funcion(); /*no se ejecuta a no ser que execl falle*/
```

Para ver el estado de un proceso en segundo plano, usaremos la llamada *waitpid()* con los flags adecuados.

`waitpid(pid, &estado, WNOHANG |WUNTRACED |WCONTINUED)` nos dará información del estado del proceso en el entero *estado* **SOLAMENTE** si el valor devuelto por *waitpid* es *pid*. Dicha información puede ser interpretada por las macros descritas en `man waitpid` (WIFEXITED, WIFSIGNALED ...)

**Información detallada de las llamadas al sistema y las funciones de la librería puede (y debe) obternerse con man (chdir, getcwd, opendir, unlink, rmdir, readdir, execvp, waitpid, fork, exit ...).**

**FORMA DE ENTREGA**

Las práctica se realizará en **GRUPOS DE DOS ALUMNOS** y se entregará mediante el repositorio de *subversion* bajo el directorio P1 antes de proceder a su defensa. Esta carpeta deberá incluir tanto el código fuente como el fichero Makefile que permite su compilación, si lo hubiere. Las cabeceras de los ficheros fuente deben incluir un comentario con los nombres de los integrantes del grupo de prácticas y el horario en el que están apuntados.

La práctica será defendida ante el profesor en el aula y en horario de prácticas. Todos los miembros del grupo deberán estar presentes para la entrega, de forma que el profesor pueda revisar su funcionamiento así como realizar comentarios/cuestiones a los integrantes del grupo o pedir cambios en el código que se puedan considerar pertinentes.

Entrega de la práctica Viernes 17 Octubre.
DEFENSA DE LA PRACTICA semana 20-24/Octubre/2014