Ejercicios Prácticas SO

Grado Ingeniería Informática

E.T.S. Ing. Informática y de Telecomunicación (ETSIIT)

Granada

Alejandro Alcalde
elbauldelprogramador.com

Ejercicios sesiones Prácticas

Alejandro Alcalde

1.1. Sesión 6

Exercise 1. Implementa un programa que admita tres argumentos. El primer argumento será una orden de Linux; el segundo, uno de los siguientes caracteres "<" o ">", y el tercero el nombre de un archivo (que puede existir o no). El programa ejecutará la orden que se especifica como argumento primero e implementará la redirección especificada por el segundo argumento hacia el archivo indicado en el tercer argumento. Por ejemplo, si deseamos redireccionar la salida estándar de sort a un archivo temporal, ejecutaríamos (el carácter de redirección > lo ponemos entrecomillado para que no lo interprete el shell y se coja como argumento del programa):

Ejercicio 1.c

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char *argv[]) {

    if (argc < 4){
        printf("Uso: %s <programa> <<|>> <fichero>.\n", argv[0]);
        exit(-1);
    }

    char redireccion = argv[2][0];
    int fd;

    switch(redireccion){
        case '<':
            /* Proporcionamos los datos al programa desde el fichero indicado */
            close(STDIN_FILENO);
```

[1]ETSIIT, Granada.
E-mail address: algui91@gmail.com.

January 14, 2014

```
            if ((fd = open(argv[3], O_RDONLY)) == -1){
                perror("open");
                exit(-1);
            }
            if (fcntl(fd, F_DUPFD, STDIN_FILENO) == -1 ){
                perror ("fcntl falló");
                exit(-1);
            }
            if( (execlp(argv[1],argv[1], NULL)<0)) {
                    perror("Error en el execl\n");
                    exit(-1);
            }

            break;
        case '>':
            /* Escribiremos la salida del programa al fichero indicado */

            close(STDOUT_FILENO);
            if ((fd = open(argv[3], O_WRONLY | O_TRUNC | O_CREAT)) == -1){
                perror("open");
                exit(-1);
            }
            if (fcntl(fd, F_DUPFD, STDOUT_FILENO) == -1 ){
                perror ("fcntl falló");
                exit(-1);
            }
            if( (execlp(argv[1],argv[1], NULL)<0)) {
                    perror("Error en el execl\n");
                    exit(-1);
            }
            break;
    }

    exit(0);
}
```

Exercise 2. Reescribir el programa que implemente un encauzamiento de dos órdenes pero utilizando fcntl. Este programa admitirá tres argumentos. El primer argumento y el tercero serán dos órdenes de Linux. El segundo argumento será el carácter "|". El programa deberá ahora hacer la redirección de la salida de la orden indicada por el primer argumento hacia el cauce, y redireccionar la entrada estándar de la segunda orden desde el cauce.

Ejercicio 2.c

```
#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[])
{
    int fd[2];
    pid_t PID;
```

```
12
13      pipe(fd); // Llamada al sistema para crear un pipe
14
15      if (argc != 4){
16          printf("Uso: %s <programa1> | <programa2>\n", argv[0]);
17          exit(-1);
18      }
19
20      char *programa_1 = argv[1];
21      char *programa_2 = argv[3];
22
23      if ( (PID= fork())<0) {
24          perror("\Error en fork");
25          exit(-1);
26      }
27      if (PID == 0) { // ls
28          //Cerrar el descriptor de lectura de cauce en el proceso hijo
29          close(fd[0]);
30
31          //Duplicar el descriptor de escritura en cauce en el descriptor
32          //correspondiente a la salida estda r (stdout), cerrado previamente en
33          //la misma operación
34          if (fcntl(fd[1], F_DUPFD, STDOUT_FILENO) == -1 ){
35              perror ("fcntl falló");
36              exit(-1);
37          }
38          execlp(programa_1,programa_1,NULL);
39      }
40      else { // sort. Proceso padre porque PID != 0.
41          //Cerrar el descriptor de escritura en cauce situado en el proceso padre
42          close(fd[1]);
43
44          //Duplicar el descriptor de lectura de cauce en el descriptor
45          //correspondiente a la entrada estándar (stdin), cerrado previamente en
46          //la misma operación
47          if (fcntl(fd[0], F_DUPFD, STDIN_FILENO) == -1 ){
48              perror ("fcntl falló");
49              exit(-1);
50          }
51          execlp(programa_2,programa_2,NULL);
52      }
53
54      return(0);
55  }
```

Exercise 3. Construir un programa que verifique que, efectivamente, el kernel comprueba que puede darse una situación de interbloqueo en el bloqueo de archivos.

Ejercicio 3.c

```
1  #include<sys/types.h>
2  #include<fcntl.h>
3  #include<unistd.h>
4  #include<stdio.h>
```

```c
#include<stdlib.h>
#include<string.h>
#include<sys/stat.h>

/**
 * Este programa probara como actua el kernel al detectar interbloqueo
 * cuando intenta realizar E/S sobre un fichero que tiene un cerrojo
 * activo y ha sido creado con bloqueo obligatorio
 */
int main(int argc, char *argv[])
{
    /* Permisos necesarios */
    mode_t permisos = (S_ISGID & ~S_IXGRP) | S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH; //S_IRWXU;

    /* Creacion del fichero */
    int fd = open("bloqueoObligatorio", O_CREAT|O_TRUNC|O_RDWR, permisos);

    if (fd == -1){
        perror("Creando o abriendo el fichero");
        exit(EXIT_FAILURE);
    }
    /* En caso de que el fichero existiera, reseteamos los permisos */
    if ( fchmod(fd, permisos) == -1){
        perror("Estableciendo bloqueo obligatorio");
        exit(EXIT_FAILURE);
    }

    /* Escribir algo en el fichero y fijar el bit de bloqueo obligatorio */
    const char* msj = "linea1\nlinea2\nlinea3\n";

    if ( write(fd, msj, strlen(msj)) == -1){
        perror("Escribiendo en el archivo");
        exit(EXIT_FAILURE);
    }

    /* Intentamos escribir en un area con un cerrojo de lectura */
    /* Establecer el cerrojo */
    struct flock cerrojo;
    cerrojo.l_type = F_RDLCK;
    cerrojo.l_whence = SEEK_SET;
    cerrojo.l_start = 0;    /* Bloquear desde el principio de la segunda linea */
    cerrojo.l_len = 0;      /* hasta el final de la segunda linea */
    cerrojo.l_pid = getpid();

    if ( fcntl(fd, F_SETLK, &cerrojo) == -1) {
        perror("Adquiriendo bloqueo");
        exit(EXIT_FAILURE);
    }

    /* Con el cerrojo de lectura, intentamos escribir en la segunda linea */
    if ( lseek(fd, 8, SEEK_SET) == -1){
        perror("Posicionando el puntero de escritura");
```

```
57          exit(EXIT_FAILURE);
58      }
59      if ( write(fd, "g", 1) == -1){
60          perror("Escribiendo");
61          exit(EXIT_FAILURE);
62      }
63
64      sleep(800);
65  }
```

Exercise 4. Construir un programa que se asegure que solo hay una instancia de él en ejecución en un momento dado. El programa, una vez que ha establecido el mecanismo para asegurar que solo una instancia se ejecuta, entrará en un bucle infinito que nos permitirá comprobar que no podemos lanzar más ejecuciones del mismo. En la construcción del mismo, deberemos asegurarnos de que el archivo a bloquear no contiene inicialmente nada escrito en una ejecución anterior que pudo quedar por una caída del sistema.

Ejercicio 4.c

```c
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/stat.h>

#define LOCK_FILE "lck.ej4"

/**
 * En este programa se pretende mostrar como usan los demonios
 * cerrojos para asegurarse que unicamente hay una instancia de
 * ellos ejecutandose, para ello se establece un cerrojo de escritura
 * en /var/run
 */
int main(int argc, char *argv[])
{
    struct flock cerrojo;
    struct stat sf;
    int fd;

    if ((fd = open(LOCK_FILE, O_WRONLY | O_CREAT, S_IRWXU)) == -1){
       perror("Creando el fichero");
       exit(EXIT_FAILURE);
    }

    cerrojo.l_type = F_WRLCK;
    cerrojo.l_whence = SEEK_SET;
    cerrojo.l_start = 0;
    cerrojo.l_len = 0;
    cerrojo.l_pid = getpid();

    /* Bloquear el fichero completo */
    if (fcntl(fd, F_SETLK, &cerrojo) == -1){
```

```
37          perror("Estableciendo cerrojo");
38          exit(EXIT_FAILURE);
39      }
40
41      /* Comprobamos si existe el fichero */
42      if (stat(LOCK_FILE, &sf) == -1 ){
43          perror("stat");
44          exit(EXIT_FAILURE);
45      }
46
47      if (sf.st_size != 0){
48          printf("Error: otro proceso no ha finalizado correctamente el fichero, borralo\n");
49          exit(EXIT_FAILURE);
50      }
51
52      char *pid;
53      sprintf(pid, "%d", getpid());
54
55      if (write(fd, pid, strlen(pid)) == -1){
56          perror("Escribiendo en el cerrojo");
57          exit(EXIT_FAILURE);
58      }
59
60      printf("Intenta ejecutar de nuevo el programa, no se podra debido a que se ha establecido el cerrojo, %s\n", pid);
61      printf("Para finalizar pulsa enter");
62      fgetc(stdin);
63
64      /* Borrar el fichero */
65      if ( close(fd) == -1 ){
66          perror("close");
67          exit(EXIT_FAILURE);
68      }
69      if (unlink(LOCK_FILE) == -1){
70          perror("unlink");
71          exit(EXIT_FAILURE);
72      }
73      return 0;
74  }
```