Modulo-II-Sesion-6.pdf

KIKONASO

Sistemas Operativos

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Módulo II. Uso de los Servicios del SO mediante la API

Sesión 6. Control de archivos y archivos proyectados a memoria

Ejercicio 1. Implementa un programa que admita t 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 entrada estándar de sort desde un archivo temporal, ejecutaríamos:

$> ./mi_programa sort "<" temporal

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

int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "Uso: %s <comando> <redirección> <archivo>\n", argv[0]);
        return 1;
    }

    char *comando = argv[1];
    char *direccion = argv[2];
    char *archivo = argv[3];

    int fd;
    if (strcmp(direccion, "<") == 0) {
        fd = open(archivo, O_RDONLY);
        if (fd < 0) {
            perror("Error al abrir el archivo");
            return 1;
        }
        printf("Archivo abierto para lectura, fd: %d\n", fd);

        // Redirigir la entrada estándar
        close(0);
        if (fcntl(fd, F_DUPFD, STDIN_FILENO) < 0) {
            perror("Error al redirigir la entrada estándar con fcntl");
            close(fd);
            return 1;
        }
        printf("Entrada estándar redirigida correctamente\n");

    } else if (strcmp(direccion, ">") == 0) {
        fd = open(archivo, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (fd < 0) {
            perror("Error al abrir el archivo");
            return 1;
        }
        printf("Archivo abierto para escritura, fd: %d\n", fd);

        // Redirigir la salida estándar
        close(1);
        if (fcntl(fd, F_DUPFD, STDOUT_FILENO) < 0) {
            perror("Error al redirigir la salida estándar con fcntl");
            close(fd);
            return 1;
        }
        printf("Salida estándar redirigida correctamente\n");
    } else {
        fprintf(stderr, "Redirección no válida: %s\n", direccion);
        return 1;
    }

    // No es necesario mantener abierto fd, ya que la entrada o salida ya está redirigida
    close(fd);

    execlp(comando, comando, NULL);

    return 0;
}
```

Ejercicio 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. Por ejemplo, para simular el encauzamiento ls|sort, ejecutaríamos nuestro programa como:

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

int main(int argc, char *argv[]) {
    if (argc != 4 || strcmp(argv[2], "|") != 0) {
        fprintf(stderr, "Uso: %s <comando1> '|' <comando2>\n", argv[0]);
        return 1;
    }

    char *comando1 = argv[1];
    char *comando2 = argv[3];

    // Crear el cauce
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        perror("Error al crear el cauce");
        return 1;
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("Error al bifurcar el proceso");
        return 1;
    }

    if (pid == 0) { // Proceso hijo: ejecuta el primer comando
        close(pipefd[0]); // Cerrar extremo de lectura

        // Redirigir salida estándar al extremo de escritura del cauce
        close(1);
        if (fcntl(pipefd[1], F_DUPFD, STDOUT_FILENO) == -1) {
            perror("Error al redirigir la salida estándar con fcntl");
            close(pipefd[1]);
            return 1;
        }
        close(pipefd[1]); // Cerrar el descriptor original

        // Ejecutar el primer comando
        execlp(comando1,comando1,NULL);
    } else { // Proceso padre: ejecuta el segundo comando
        close(pipefd[1]); // Cerrar extremo de escritura

        // Redirigir entrada estándar al extremo de lectura del cauce
        close(0);
        if (fcntl(pipefd[0], F_DUPFD, STDIN_FILENO) == -1) {
            perror("Error al redirigir la entrada estándar con fcntl");
            close(pipefd[0]);
            return 1;
        }
        close(pipefd[0]); // Cerrar el descriptor original

        // Ejecutar el segundo comando
        execlp(comando2,comando2,NULL);

        // Esperar a que el proceso hijo termine
        waitpid(pid, NULL, 0);
    }

    return 0;
}
```

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

void bloquear_archivo(int fd, struct flock *cerrojo, short tipo_cerrojo){

    cerrojo->l_type=tipo_cerrojo;
    cerrojo->l_start= 0;
    cerrojo->l_whence= SEEK_SET;
    cerrojo->l_len=0;

    if (fcntl(fd, F_SETLKW, cerrojo) == -1){
        perror("Error  al bloquear el archivo");
        exit(-1);
    }
    printf("Proceso %d: %s bloqueado \n", getpid(), tipo_cerrojo== F_WRLCK ? "Escritura" : "Lectura");
}

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

    int fd1= open("archivo1.txt", O_CREAT | O_RDWR, 0666);
    int fd2= open("archivo2.txt", O_CREAT | O_RDWR, 0666);

    if (fd1 == -1 || fd2==-1){
        perror("Error al abrir los archivos");
        exit(-1);
    }

    struct flock cerrojo1, cerrojo2;

    pid_t pid=fork();

    if (pid == 0) /* HIJO */ {
        bloquear_archivo(fd1, &cerrojo1, F_WRLCK);
        sleep(2); //Se da  tiempo a que el padre bloquee el archivo 2
        //Ahora intentamos bloquear el archivo 2
        printf("\n Proceso hijo intentando bloquear el archivo 2 \n");
        bloquear_archivo(fd2, &cerrojo2, F_WRLCK);

    }
    else{
        bloquear_archivo(fd2, &cerrojo2, F_WRLCK);
        sleep(2); //Se da  tiempo a que el hijo bloquee el archivo 1
        //Ahora intentamos bloquear el archivo 1
        printf("\n Proceso padre intentando bloquear el archivo 1 \n");
        bloquear_archivo(fd1, &cerrojo1, F_WRLCK);

        wait(NULL); //Esperamos a que termine el hijo
    }

    //Cerramos descriptores de archivo

    close(fd1);
    close(fd2);

    return 0;

}
```

```
tomy@Lenovo-Tomas:~/Documentos/2ºCARRERA/SO/ModuloII/Sesión 6$ gcc ejercicio3.c -o ej3
tomy@Lenovo-Tomas:~/Documentos/2ºCARRERA/SO/ModuloII/Sesión 6$ ./ej3
Proceso 7711: Escritura bloqueado
Proceso 7712: Escritura bloqueado

 Proceso padre intentando bloquear el archivo 1

 Proceso hijo intentando bloquear el archivo 2
Error  al bloquear el archivo: Resource deadlock avoided
Proceso 7711: Escritura bloqueado
tomy@Lenovo-Tomas:~/Documentos/2ºCARRERA/SO/ModuloII/Sesión 6$
```

El error Resource deadlock avoided indica que el kernel detectó un intento de bloqueo que podría resultar en un interbloqueo y lo previno. Esto ocurre porque el kernel no permite ciertos tipos de cerrojos si reconoce una dependencia circular, lo cual es una característica interesante del subsistema de bloqueo de archivos.

El comportamiento depende del sistema operativo y la implementación del sistema de archivos. Linux usa un mecanismo simple de prevención para evitar interbloqueos cuando se detecta una posible situación peligrosa.
Por qué ocurre esto

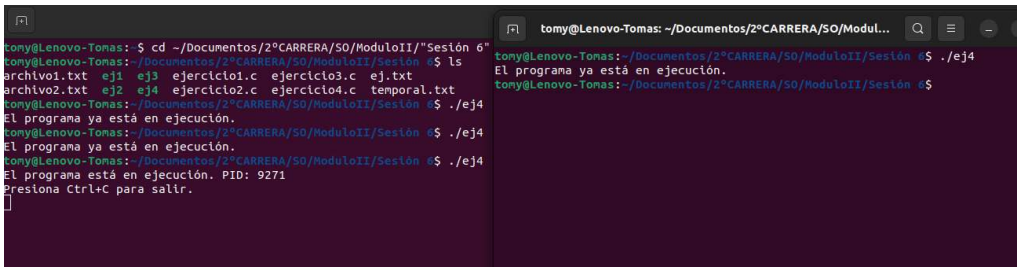
El kernel evita la espera indefinida si detecta que el cerrojo solicitado depende de otro cerrojo ya existente en el mismo proceso o en otro proceso, y se generaría una espera circular.

En este caso:

- El padre bloquea archivo2.
- El hijo bloquea archivo1.
- El hijo intenta bloquear archivo2, pero ya está bloqueado por el padre.
- El padre intenta bloquear archivo1, pero ya está bloqueado por el hijo.

Cuando el segundo intento ocurre, el kernel detecta esta espera circular y evita el bloqueo, devolviendo un error.

Ejercicio 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.



Queremos que la ejecución muestre algo como esto, para ello implementamos el siguiente código:

```c
//Voy a definir la ruta para el archivo de bloqueo

#define AR_BLOQUEO "/tmp/bloqueo.pid"

void liberar_cerrojo(int fd){
    if (close(fd) == -1){
        perror("Error al cerrar el archivo de bloqueo");
    }
    else{
        printf("\n Cerrojo liberado correctamente. Saliendo... \n");
    }

    unlink(AR_BLOQUEO); //Eliminamos el archivo de bloqueo
    exit(EXIT_SUCCESS);
}

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

    int fd;

    fd=open(AR_BLOQUEO, O_CREAT | O_RDWR | O_TRUNC, 0666);

    if (fd == -1){
        perror("Error abriendo el archivo");
        exit(-1);
    }

    struct flock cerrojo;

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

    // Intentar aplicar el cerrojo
    if (fcntl(fd, F_SETLK, &cerrojo) == -1) {
        if (errno == EACCES || errno == EAGAIN) {
            // Si el cerrojo no se pudo establecer, otro proceso ya lo tiene
            fprintf(stderr, "El programa ya está en ejecución.\n");
        } else {
            perror("Error al establecer el cerrojo");
        }
        close(fd);
        exit(EXIT_FAILURE);
    }

    //Escribir el PID del proceso en el archivo de bloqueo
    char pid_str[32];
    snprintf(pid_str, sizeof(pid_str), "Proceso: %d \n", getpid());
    if (write(fd, pid_str, strlen(pid_str)) == -1){
        perror("Error al escribir el PID en el archivo");
        close(fd);
        unlink(AR_BLOQUEO);
    }

    // Configurar señales para liberar el cerrojo al finalizar
    signal(SIGINT, liberar_cerrojo); // Ctrl+C
    signal(SIGTERM, liberar_cerrojo); // kill

    printf("El programa está en ejecución. PID: %d\n", getpid());
    printf("Presiona Ctrl+C para salir.\n");

    //Bucle infinito

    while(1){

    }

    return 0;

}
```