Modulo-II-Sesion-4.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 4. Comunicación entre procesos utilizando cauces

Ejercicio 1. Consulte en el manual las llamadas al sistema para la creación de archivos especiales en general (mknod) y la específica para archivos FIFO (mkfifo). Pruebe a ejecutar el siguiente código correspondiente a dos programas que modelan el problema del productor/consumidor, los cuales utilizan como mecanismo de comunicación un cauce FIFO. Determine en qué orden y manera se han de ejecutar los dos programas para su correcto funcionamiento y cómo queda reflejado en el sistema que estamos utilizando un cauce FIFO. Justifique la respuesta.

Funciones del sistema consultadas

1. **mknod:**
    - Se utiliza para crear archivos especiales en sistemas Unix/Linux, incluyendo archivos FIFO, dispositivos de bloque o caracteres.
    - Sintaxis básica:

      int mknod(const char *pathname, mode_t mode, dev_t dev);

    - Para un FIFO, el argumento mode incluye el bit S_IFIFO.

2. **mkfifo:**

- Específica para la creación de archivos FIFO (cauces con nombre).
- Sintaxis básica:

  int mkfifo(const char *pathname, mode_t mode);

- Más simple y recomendada sobre mknod para este propósito.

Ambas funciones crean un archivo especial en el sistema de archivos, visible con comandos como ls o stat.

Ejecución

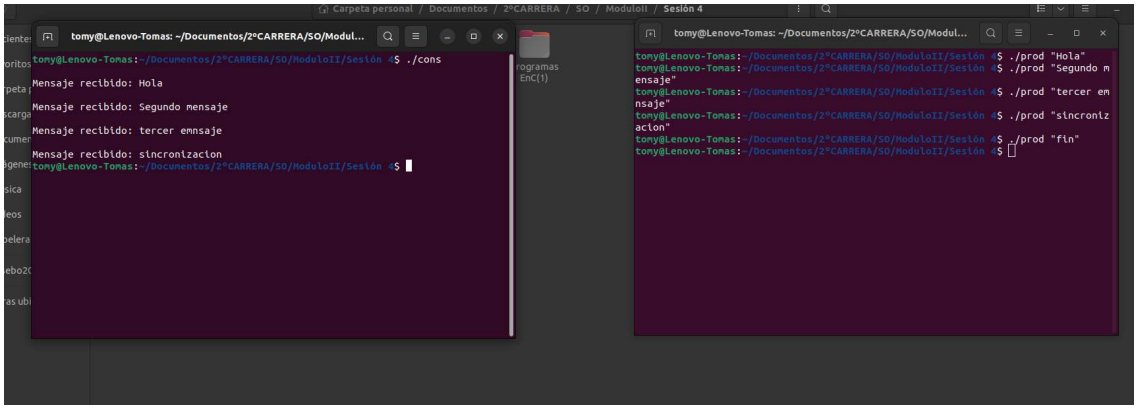
Para la correcta ejecución y realización del ejercicio, primero hay que compilar los dos archivos, luego abrimos una terminal y ejecutamos primero el consumidor:

- Debe ejecutarse primero para abrir el FIFO en modo lectura (`O_RDWR`) y estar preparado para recibir mensajes.
- Si no hay ningún lector activo cuando el productor intenta escribir, el productor se bloqueará (esperará).

Ahora abrimos otra terminal y ejecutamos el productor con los mensajes que queremos:

- Se ejecuta después para abrir el FIFO en modo escritura (`O_WRONLY`) y enviar mensajes al consumidor.
- Si no hay un lector activo, el productor esperará a que alguien abra el FIFO para lectura.

De esta manera podemos ir viendo los mensajes que han sido enviados y recibidos:

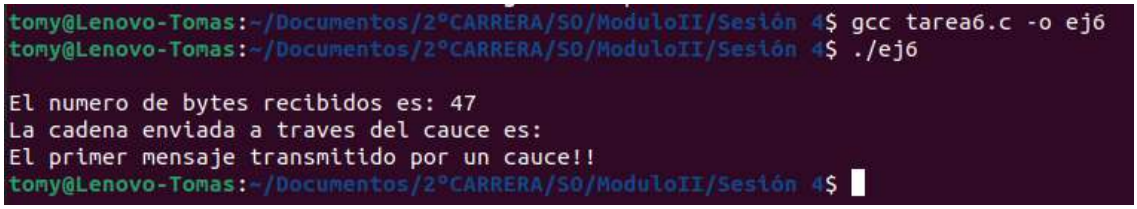


Ejercicio 2. Consulte en el manual en línea la llamada al sistema pipe para la creación de cauces sin nombre. Pruebe a ejecutar el siguiente programa que utiliza un cauce sin nombre y describa la función que realiza. Justifique la respuesta.

Análisis del programa

1. **Creación del cauce**:
   La llamada a pipe(fd) crea un cauce sin nombre, proporcionando un par de descriptores:
   - fd[0]: Descriptor de lectura.
   - fd[1]: Descriptor de escritura.
2. **Creación de procesos**:
   Se utiliza fork() para crear un proceso hijo:
   - El proceso hijo se encarga de **escribir** datos en el cauce.
   - El proceso padre se encarga de **leer** datos del cauce.
3. **Proceso hijo**:
   - Cierra el descriptor de lectura fd[0] ya que no lo necesita.
   - Escribe el mensaje en el cauce a través de fd[1].
   - Termina usando exit(0).
4. **Proceso padre**:
   - Cierra el descriptor de escritura fd[1] porque no lo usará.
   - Lee los datos desde el cauce a través de fd[0].
   - Muestra en la terminal el número de bytes leídos y el mensaje recibido.
5. **Transmisión de datos**:
   El cauce actúa como un **canal unidireccional** para transmitir datos desde el proceso hijo hacia el proceso padre.

La ejecución del programa debe mostrar algo como esto:



Justificación del funcionamiento

1. **Unidireccionalidad**:
   Los cauces sin nombre permiten transmitir datos de un proceso a otro en una única dirección: escritura en un extremo (fd[1]) y lectura en el otro (fd[0]).
2. **Sin persistencia**:
   A diferencia de los cauces con nombre (FIFO), los cauces sin nombre existen únicamente mientras los procesos que los utilizan están en ejecución. Una vez que ambos procesos cierran los descriptores asociados, el cauce desaparece.
3. **Coordinación entre procesos**:
   Este programa es un ejemplo básico de comunicación interproceso (IPC), donde los procesos padre e hijo colaboran a través de un cauce.
4. **Bloqueo**:
   - Si el proceso padre intenta leer del cauce y no hay datos disponibles, se bloquea hasta que el hijo escriba.
   - Si el hijo escribe y el padre no está leyendo, los datos quedan en el buffer hasta ser leídos.

Ejercicio 3. Redirigiendo las entradas y salidas estándares de los procesos a los cauces podemos escribir un programa en lenguaje C que permita comunicar órdenes existentes sin necesidad de reprogramarlas, tal como hace el shell (por ejemplo ls | sort). En particular, ejecute el siguiente programa que ilustra la comunicación entre proceso padre e hijo a través de un cauce sin nombre redirigiendo la entrada estándar y la salida estándar del padre y el hijo respectivamente.

Análisis del programa

1. **Creación del cauce**:
   - Se utiliza pipe(fd) para crear un cauce sin nombre.
   - fd[0]: Descriptor para lectura.
   - fd[1]: Descriptor para escritura.
2. **Creación de procesos**:
   - Se usa fork() para generar un proceso hijo.
   - **Proceso hijo (ls)**:
     - Cierra el descriptor de lectura fd[0], ya que no lo necesita.

- Redirige su salida estándar (stdout) al descriptor de escritura del cauce (fd[1]) utilizando dup().
- Ejecuta el comando ls con execlp("ls", "ls", NULL), que escribe su salida al cauce.

- **Proceso padre (sort)**:
  - Cierra el descriptor de escritura fd[1], ya que no lo necesita.
  - Redirige su entrada estándar (stdin) al descriptor de lectura del cauce (fd[0]) utilizando dup().
  - Ejecuta el comando sort con execlp("sort", "sort", NULL), que toma la entrada del cauce.

3. **Interacción entre los procesos**:
   - La salida del proceso hijo (ls) es redirigida al cauce.
   - El proceso padre lee del cauce y utiliza esos datos como entrada para ejecutar sort.

Ejecución del programa

```
tomy@Lenovo-Tomas:~/Documentos/2ºCARRERA/SO/ModuloII/Sesión 4$ gcc tarea7.c -o ej7
tomy@Lenovo-Tomas:~/Documentos/2ºCARRERA/SO/ModuloII/Sesión 4$ ./ej7
ComunicacionFIFO
cons
consumidorFIFO.c
ej6
ej7
prod
productorFIFO.c
ProgramasEnC(1)
tarea6.c
tarea7.c
tomy@Lenovo-Tomas:~/Documentos/2ºCARRERA/SO/ModuloII/Sesión 4$
```

Cuando ejecutes este programa, debería producir un resultado equivalente a ejecutar ls | sort en la línea de comandos. Esto significa que:

1. El programa ls generará una lista de archivos y directorios del directorio actual.
2. Esta lista será enviada al cauce.
3. El programa sort recibirá la lista desde el cauce y la ordenará alfabéticamente antes de mostrarla.

Ejercicio 4. Compare el siguiente programa con el anterior y ejecútalo. Describa la principal diferencia, si existe, tanto en su código como en el resultado de la ejecución.

Uso de dup() vs dup2():

- En el programa **anterior**, la redirección se logra cerrando manualmente la entrada/salida estándar (STDIN_FILENO, STDOUT_FILENO) y luego duplicando el descriptor del cauce con dup(). Esto requiere dos llamadas: primero cerrar el descriptor estándar y luego duplicar.
- En este programa, se utiliza dup2(fd[1], STDOUT_FILENO) y dup2(fd[0], STDIN_FILENO) para realizar la redirección de forma directa y más concisa. dup2() cierra automáticamente el descriptor estándar antes de duplicar el descriptor del cauce.

Ventaja de dup2(): Simplifica el código al combinar ambas operaciones en una sola llamada.

Claridad y legibilidad:

- El uso de dup2() en este programa hace que el código sea más limpio y fácil de entender. No es necesario realizar las operaciones de cierre manual de descriptores estándar.

Manejo de errores:

- Ambos programas son similares en el manejo de errores básicos (fork y pipe). Sin embargo, dup2() es más robusto en caso de que ya exista un descriptor abierto en STDOUT_FILENO o STDIN_FILENO.

En cuanto a la ejecución del programa, no varía en nada en lo que vemos, pero internamente el segundo programa es más eficiente en la redirección estándar debido a la simplicidad de llamada al sistema (se hace una llamada en lugar de dos)

Ejercicio 5. Este ejercicio se basa en la idea de utilizar varios procesos para realizar partes de una computación en paralelo. Para ello, deberá construir un programa que siga el esquema de computación maestro-esclavo, en el cual existen varios procesos trabajadores (esclavos) idénticos y un único proceso que reparte trabajo y reúne resultados (maestro). Cada esclavo es capaz de realizar una computación que le asigne el maestro y enviar a este último los resultados para que sean mostrados en pantalla por el maestro. El ejercicio concreto a programar consistirá en el cálculo de los números primos que hay en un intervalo. Será necesario construir dos programas, maestro y esclavo. Ten en cuenta la siguiente especificación:

1. **El intervalo de números naturales donde calcular los número primos se pasará como argumento al programa maestro. El maestro creará dos procesos esclavos y dividirá el intervalo en dos subintervalos de igual tamaño pasando cada subintervalo como argumento a cada programa esclavo. Por ejemplo, si al maestro le proporcionamos el intervalo entre 1000 y 2000, entonces un esclavo debe calcular y devolver los números primos comprendidos en el subintervalo entre 1000 y 1500, y el otro esclavo entre 1501 y 2000. El maestro creará dos cauces sin nombre y se encargará de su redirección para comunicarse con los procesos esclavos. El maestro irá recibiendo y mostrando en pantalla (también uno a uno) los números primos calculados por los esclavos en orden creciente.**

2. **El programa esclavo tiene como argumentos el extremo inferior y superior del intervalo sobre el que buscará números primos. Para identificar un número primo utiliza el siguiente método concreto: un número n es primo si no es divisible por ningún k tal que 2 < k <= sqrt(n), donde sqrt corresponde a la función de cálculo de la raíz cuadrada (consulte dicha función en el manual). El esclavo envía al maestro cada primo encontrado como un dato entero (4 bytes) que escribe en la salida estándar, la cuál se tiene que encontrar redireccionada a un cauce sin nombre. Los dos cauces sin nombre necesarios, cada uno para comunicar cada esclavo con el maestro, los creará el maestro inicialmente. Una vez que un esclavo haya calculado y enviado (uno a uno) al maestro todos los primos en su correspondiente intervalo terminará.**

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

Que te den **10 € para gastar** es una fantasía. ING lo hace realidad.

Abre la **Cuenta NoCuenta** con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Quiero el cash

Consulta condiciones **aquí**

do your thing

Maestro.c

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

#define TAM 30

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

    if (argc !=3){
        perror("Error, uso ./maestro <limite_inferior> <limite_superior>");
        exit(-1);
    }

    int inferior= atoi(argv[1]);
    int superior= atoi(argv[2]);

    if (inferior >= superior){
        perror("El limite inferior debe ser menor que el limite superior");
        exit(-1);
    }

    int centro= inferior + ((superior-inferior)/2);

    int pipe1[2], pipe2[2];

    if (pipe(pipe1) == -1 || pipe(pipe2) == -1 ){
        perror("Error al crear los cauces");
        exit(-1);
    }

    pid_t esclavo1= fork();

    if (esclavo1 == -1) {
        perror("Error en fork para el primer esclavo");
        exit(-1);
    }

    if(esclavo1 == 0){
        //Proceso esclavo 1
        close(pipe1[0]); //Cierra el extremo de lectura
        dup2(pipe1[1], STDOUT_FILENO); //Redirige stdout al cauce
        close(pipe1[1]);

        //Almaceno los limites en cadenas
        char inferior_str[TAM], centro_str[TAM];
        sprintf(inferior_str, "%d", inferior);
        sprintf(centro_str, "%d", centro);

        execlp("./esclavo", "esclavo", inferior_str, centro_str, NULL);
        //Si la ejecución del esclavo falla:
        perror("Error al ejecutar el primer esclavo");
        exit(-1);
    }

    pid_t esclavo2= fork();

    if (esclavo2 == -1) {
        perror("Error en fork para el segundo esclavo");
        exit(-1);
    }

    if(esclavo2 == 0){
        //Proceso esclavo 2
        close(pipe2[0]); //Cierra el extremo de lectura
        dup2(pipe2[1], STDOUT_FILENO); //Redirige stdout al cauce
        close(pipe2[1]);

        //Almaceno los limites en cadenas
        char centro_str[TAM], superior_str[TAM];
        sprintf(centro_str, "%d", centro + 1);
        sprintf(superior_str, "%d", superior);

        execlp("./esclavo", "esclavo", centro_str, superior_str, NULL);
        //Si la ejecución del esclavo falla:
        perror("Error al ejecutar el segundo esclavo");
        exit(-1);
    }

    //Proceso maestro

    close(pipe1[1]); // Cierra los extremos
    close(pipe2[1]); // de escritura

    int mostrar;
    printf("Números primos encontrados: \n");

    //Leemos y mostramos los resultados de los cauces

    printf("\nEscritos por esclavo 1: \n \n");
    while(read(pipe1[0], &mostrar, sizeof(int)) > 0){
        printf("%d \n", mostrar);
    }

    printf("\nEscritos por esclavo 2: \n \n");
    while(read(pipe2[0], &mostrar, sizeof(int)) > 0){
        printf("%d \n", mostrar);
    }


    close(pipe1[0]); // Cierra los extremos
    close(pipe2[0]); // de lectura


    return 0;

}
```

Esclavo.c

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

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

    if (argc !=3){
        perror("Error, uso ./esclavo <limite_inferior> <limite_superior>");
        exit(-1);
    }

    int inferior= atoi(argv[1]);
    int superior= atoi(argv[2]);

    if (inferior >= superior){
        perror("El limite inferior debe ser menor que el limite superior");
        exit(-1);
    }


    for(int i=inferior; i <= superior ; ++i){
        int divisible=0;
        for(int k=2; k <= (int)sqrt(i) && divisible==0 ; ++k){
            if(i % k == 0 ){
                divisible=1;
                break;
            }
        }

        if (divisible == 0){ //Si es primo

            if (write(STDOUT_FILENO, &i, sizeof(int)) != sizeof(int)){
                perror("Error al escribir en el cauce.");
                exit(-1);
            }
        }
    }

    return 0;

}

```

Ejemplo de ejecución