Sistemas Operativos

Formulario de auto-evaluación

Modulo 2. Sesión 4. Comunicación entre procesos utilizando cauces

Nombre y apellidos:

Jesús Manuel García Palma

a) Cuestionario de actitud frente al trabajo.

El tiempo que he dedicado a la preparación de la sesión antes de asistir al laboratorio ha sido de ..... minutos.

1. He resuelto todas las dudas que tenía antes de iniciar la sesión de prácticas: ..... (si/no). En caso de haber contestado “no”, indica los motivos por los que no las has resuelto:

2. Tengo que trabajar algo más los conceptos sobre:

3. Comentarios y sugerencias:

b) Cuestionario de conocimientos adquiridos.

Mi solución al **ejercicio 1** ha sido:

Primero hay que ejecutar el programa consumidor. Una vez comenzado ejecutamos el programa productor. Al hallarse en bucle infinito el programa consumidor esperando a la entrada estándar, imprime el mensaje recibido hasta recibir la cadena 'fin'.

Estamos usando un cauce FIFO porque el archivo FIFO que hemos utilizado permanece.

Mi solución a la **ejercicio 2** ha sido:

Mediante la orden pipe se crea un cauce, y se le da un vector de enteros fd como parámetro. Fd[0] será modo lectura y fd[1] de escritura. Después se crea el hijo con fork y se cierra el descriptor de escritura (fd[1]) con la orden write.

En la zona del código del proceso padre lo primero que se hace es cerrar el descriptor de escritura fd[1] y después se leen los datos del cauce usando el descriptor de lectura fd[0] con la orden read. Después se imprime el número de bytes de la cadena recibida y esa misma cadena.

Mi solución a la **ejercicio 4** ha sido:

Las diferencias que se encuentran en ambos códigos son varias.

En el primer programa se utiliza la orden close para cerrar la salida estándar y dejar la entrada del descriptor de lectura del hijo libre y luego dup para duplicar el descriptor en el cauce. Hace lo mismo con el proceso padre pero en la entrada estándar.

El segundo programa utiliza dup2, que hace lo mismo que las otras dos órdenes, cerrando el descriptor antiguo y duplicando el siguiente descriptor. La llamada es atómico por lo que si llega una señal al proceso, la operación se realizará antes de devolverle el control al núcleo para gestionar la señal. De esta manera se ha creado un cauce en el que el proceso hijo ejecuta la orden ls y la redirecciona al descriptor de escritura de la salida deseada, no la salida estándar, por lo que el proceso padre recibe la información de la orden que ejecutó el hijo y después ejecuta la orden sort. La salida que da el programa certifica que el programa se ha ejecutado correctamente.

Mi solución a la **ejercicio 5** ha sido:

Esclavo.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <math.h> // Incluir en la compilación con gcc -lm



int esPrimo(int n){
	int i;
	int limite = sqrt(n);
	int es_primo = 1;
	for (i = 2; i <= limite && es_primo; i++)
		if (n % i == 0)
			es_primo = 0;
	return es_primo;
};


int main(int argc, char *argv[]){
	int inicio, fin, i;
	inicio = atoi(argv[1]);
	fin = atoi(argv[2]);
	for (i = inicio; i < fin; i++)
		if (esPrimo(i))
			write(STDOUT_FILENO, &i, sizeof(int));
	return 0;
```

```
}
```

maestro.c

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include<stdio.h>
#include<errno.h>
#include<stdlib.h>
#include<string.h>
#include<fcntl.h>
int main(int argc, char *argv[]){
        int fd1[2];
        int fd2[2];
        int bytesLeidos, bytesLeidos2, val1, val2;
        pid_t esclavo1, esclavo2;
        int intervalos[6];
        char ini[10];
        char fin[10];
        if (argc < 3){
                perror("Uso: ./maestro <inicio> <fin>\n");
                exit(-1);
}

        // División de intervalos:
        intervalos[0] = atoi(argv[1]);
        // Maestro ini.
        intervalos[1] = atoi(argv[2]);
        // Maestro fin.
        intervalos[2] = intervalos[0];
```

```
// Esclavo1 ini.
intervalos[3] = ((intervalos[1]+intervalos[0]) / 2) - 1; // Esclavo1 fin.
intervalos[4] = intervalos[3] + 1; // Esclavo2 ini.
intervalos[5] = intervalos[1];
// Esclavo2 fin.
// Se crean dos cauces.
pipe(fd1);
pipe(fd2);
printf("\nNúmeros primos en el intervalo [%d,%d]:\n", intervalos[0],
intervalos[1]);
// Primer esclavo
esclavo1 = fork();
sprintf(ini, "%d", intervalos[2]);
sprintf(fin, "%d", intervalos[3]);
if (esclavo1 == 0){ // Proceso hijo1.
        close(fd1[0]); //Cierro descriptor de lectura
        dup2(fd1[1],STDOUT_FILENO);
if(execl("./esclavo", "esclavo", ini, fin, NULL) < 0) {
        perror("\nError en el execl");
        exit(-1);
}
}else{ // Proceso padre.
        close(fd1[1]); // Se cierra el descriptor de escritura
while((bytesLeidos = read(fd1[0],&val1, sizeof(int))) > 0){
        printf("%d ", val1);
}
close(fd1[0]);
printf("\n");
}
// Segundo esclavo
esclavo2 = fork();
```

```
        sprintf(ini, "%d", intervalos[4]);
        sprintf(fin, "%d", intervalos[5]);
        if (esclavo2 == 0){
                close(fd2[0]); //Cierro descriptor de lectura
                dup2(fd2[1],STDOUT_FILENO);
                if(execl("./esclavo", "esclavo", ini, fin, NULL) < 0) {
                        perror("\nError en el execl");
                        exit(-1);
                }
                }else{ // Proceso padre.
                        close(fd2[1]);
                        // Se cierra el descriptor de escritura.
                while((bytesLeidos2 = read(fd2[0],&val2, sizeof(int))) > 0){
                        printf("%d ", val2);
                }
                close(fd2[0]);
                printf("\n");
        }
        return 0;
}
```