



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

PRÁCTICA CINCO

Uso de Pipe y Fork

Alumno:

Rojas Zepeda Luis Eduardo

Profesor:

Velez Saldaña Ulises

2CM6, 09/04/19



Índice

1. Teoría	3
2. Material	5
3. Desarrollo	6
4. Bibliografía	14

1. Teoría

Tuberías Una tubería (pipe, cauce o ‘—’) consiste en una cadena de procesos conectados de forma tal que la salida de cada elemento de la cadena es la entrada del próximo. Permiten la comunicación y sincronización entre procesos. Es común el uso de buffer de datos entre elementos consecutivos. Una tubería es unidireccional.

Una tubería tiene en realidad dos descriptores de fichero: uno para el extremo de escritura y otro para el extremo de lectura. Como los descriptores de fichero de UNIX son simplemente enteros, un pipe o tubería no es más que un array de dos enteros.

Para crear la tubería se emplea la función `pipe()`, que abre dos descriptores de fichero y almacena su valor en los dos enteros que contiene el array de descriptores de fichero. El primer descriptor de fichero es abierto como `ORDONLY`, es decir, sólo puede ser empleado para lecturas. El segundo se abre como `OW-ONLY`, limitando su uso a la escritura. De esta manera se asegura que el pipe sea de un solo sentido: por un extremo se escribe y por el otro se lee, pero nunca al revés.

```
int tuberia [2];
pipe (tuberia);
```

La tubería “p” se hereda al hacer el `fork()` que da lugar al proceso hijo, pero es necesario que el padre haga un `close()` de `p[0]` (el lado de lectura de la tubería), y el hijo haga un `close()` de `p[1]` (el lado de escritura de la tubería). Una vez hecho esto, los dos procesos pueden emplear la tubería para comunicarse (siempre unidireccionalmente), haciendo `write()` en `p[1]` y `read()` en `p[0]`, respectivamente.

```
int main( int argc , char **argv )
{
    pid_t pid;
    int p[2], readbytes;
    char buffer [SIZE];

    pipe( p );

    if ( (pid=fork()) == 0 )
    { // hijo
        close( p[1] ); /* cerramos el lado de escritura del pipe */

        while( (readbytes=read( p[0], buffer, SIZE )) > 0 )
            write( 1, buffer, readbytes );

        close( p[0] );
    }
    else
    { // padre
        close( p[0] ); /* cerramos el lado de lectura del
                        pipe */

        strcpy( buffer, "Esto_llega_a_traves_de_la_tuberia\
n" );
        write( p[1], buffer, strlen( buffer ) );

        close( p[1] );
    }
}
```

```
waitpid( pid, NULL, 0 );  
exit( 0 );  
}
```

Makefile Make es una herramienta de gestión de dependencias, típicamente, las que existen entre los archivos que componen el código fuente de un programa, para dirigir su recompilación o "generación" automáticamente. Si bien es cierto que su función básica consiste en determinar automáticamente qué partes de un programa requieren ser recompiladas y ejecutar los comandos necesarios para hacerlo, también lo es que Make puede usarse en cualquier escenario en el que se requiera, de alguna forma, actualizar automáticamente un conjunto de archivos a partir de otro, cada vez que éste cambie

La estructura básica es la siguiente:

```
objetivo: dependencias  
comandos
```

En "objetivo" definimos el módulo o programa que queremos crear, después de los dos puntos y en la misma línea podemos definir qué otros módulos o programas son necesarios para conseguir el "objetivo". Por último, en la línea siguiente y sucesivas indicamos los comandos necesarios para llevar esto a cabo. Es muy importante que los comandos estén separados por un tabulador del comienzo de línea.

2. Material

Editor de Texto Nano

Solo se instala con la siguiente linea:

```
sudo apt-get install nano
```

Copilador gcc

Se instala con la siguiente linea:

```
sudo apt-get install gcc
```

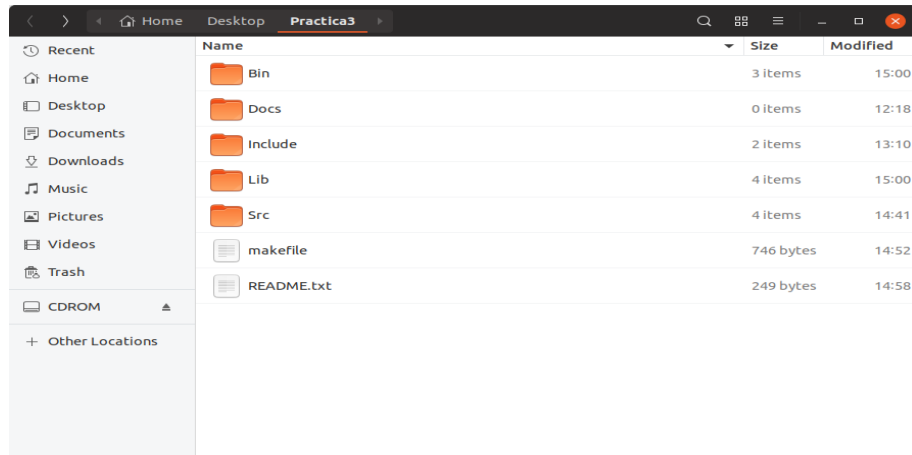
Makefile

Se instala con la siguiente linea:

```
sudo make install
```

3. Desarrollo

Crearemos las siguientes carpetas:



Después se debe generar le siguiente códigos que es el algoritmo de Round Robin:

```

/*
\mainpage
\author: Rojas Zepeda Luis Eduardo
\version 1.0
\date April 19 2019

*/
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
#include <time.h>
#include<sys/wait.h>
#include "../Include/libreria.h"

/**Cabecera funcion numRandom*/
void numRandom(int*, int);
/**Cabecera funcion numPares*/
void numPares(int*, int*, int, int);
/**Cabecera funcion numImpares*/
void numImpares(int*, int*, int, int);
/**Cabecera funcion imprimeArreglo*/
void imprimeArreglo(int, int*, char*);
/**Cabecera funcion sumaElementos*/
int sumaElementos(int*, int);

/**\brief Programa principal. */
int main()
{
    int fd1[2];
    int fd2[2];

```

```
int fd3[2];
int fd4[2];
int aux1[2];
int aux2[2];

srand(time(NULL));
fflush(stdin);

pid_t p;
pid_t p2;
int n;

if (pipe(fd1)==-1)
{
    fprintf(stderr, "Error_al_iniciar_pipe");
    return 1;
}
if (pipe(fd2)==-1)
{
    fprintf(stderr, "Error_al_iniciar_pipe");
    return 1;
}
if (pipe(fd3)==-1)
{
    fprintf(stderr, "Error_al_iniciar_pipe");
    return 1;
}
if (pipe(fd4)==-1)
{
    fprintf(stderr, "Error_al_iniciar_pipe");
    return 1;
}
if (pipe(aux1)==-1)
{
    fprintf(stderr, "Error_al_iniciar_pipe");
    return 1;
}
if (pipe(aux2)==-1)
{
    fprintf(stderr, "Error_al_iniciar_pipe");
    return 1;
}

printf("Numeros_aleatorios:\n");
scanf("%d", &n);

//Primer hijo
p = fork();

if (p < 0)
{
    fprintf(stderr, "Error_al_iniciar_fork");
    return 1;
}
else if (p > 0)
{
    int random[n], cont=0, i, suma;
    //Se llana un arreglo con n numeros aleatorios
    numRandom(random, n);

    for(i=0; i<n; i++)
        if(random[i]%2)
```

```
        cont++;

        int impares[cont];
        //Se llena un arreglo con los valores impares del
        //arreglo inicial
        numImpares(random, impares, n, cont);

        close(fd1[0]);
        close(aux1[0]);

        write(aux1[1], &cont, sizeof(cont));
        write(fd1[1], impares, sizeof(impares)+1);
        close(aux1[1]);
        close(fd1[1]);

        wait(NULL);

        close(fd2[1]);

        read(fd2[0], &suma, sizeof(suma));
        printf("Suma de impares: %d\n", suma);
        close(fd2[0]);

        //Segundo hijo
        p2 = fork();

        if (p2 < 0)
        {
            fprintf(stderr, "Error al iniciar fork");
            return 1;
        }
        else if (p2 > 0)
        {
            cont = n - cont;
            int pares[cont], suma;
            numPares(random, pares, n, cont);

            close(fd3[0]);
            close(aux2[0]);

            write(aux2[1], &cont, sizeof(cont));
            write(fd3[1], pares, sizeof(pares)+1);
            close(aux2[1]);
            close(fd3[1]);

            wait(NULL);

            close(fd4[1]);

            read(fd4[0], &suma, sizeof(suma));
            printf("Suma de pares: %d\n", suma);
            close(fd4[0]);
        }
        else
        {
            close(fd3[1]);
            close(aux2[1]);

            int cont, suma;

            read(aux2[0], &cont, sizeof(cont));
            int pares[cont];
```



```

        read(fd3[0], pares, sizeof(pares));

        suma = sumaElementos(pares, cont);

        close(fd3[0]);
        close(fd4[0]);

        write(fd4[1], &suma, sizeof(suma));
        close(fd4[1]);

        exit(0);
    }
}
else
{
    close(fd1[1]);
    close(aux1[1]);

    int cont, suma;

    read(aux1[0], &cont, sizeof(cont));
    int impares[cont];
    read(fd1[0], impares, sizeof(impares));

    suma = sumaElementos(impares, cont);

    close(fd1[0]);
    close(fd2[0]);

    write(fd2[1], &suma, sizeof(suma));
    close(fd2[1]);

    exit(0);
}
}

```

Haremos uso de la siguiente libreria.

```

/*
\Libreria de funciones para arreglos
\author: Rojas Zepeda Luis Eduardo
\version 1.0
\date Aprl 9 2019

*/

/**
\brief Funcion que imprime los elemtos de un arreglo.
\param cont Entero con valor del tamano del arreglo.
\param array Apuntador a entero con direccion del arreglo.
\param cadena Apuntador a cadena del nombre del arreglo.
*/
void imprimeArreglo(int cont, int* array, char* cadena){
    int i;
    printf("\n");
    for(i=0; i<cont; i++)
        printf("%s[%d] = %d\n", cadena, i, array[i]);
}

/**
\brief Funcion que asigna valores aleatorios a un arreglo.

```

```

\param random Apuntador a entero con direccion del arreglo.
\param tam Entero con valor del tamaño del arreglo.
\return int entero con valor 0 para falso y algo diferente
para verdadero.
*/
void numRandom(int* random, int tam){
    int i;
    for(i=0; i<tam; i++)
        random[i] = rand() % 100;

    imprimeArreglo(tam, random, "random");
}

/**
\brief Funcion que asigna los valores impares de un arreglo
a otro.
\param random Apuntador a entero con direccion del arreglo
completo.
\param impares Apuntador a entero con direccion del arreglo
al que solo se le quiere asignar valores impares.
\param tam Entero con valor del tamaño del arreglo.
\param cont Entero con valor del número de números impares
en el arreglo completo.
*/
void numImpares(int* random, int* impares, int tam, int
cont){
    int j=0, i=0;
    for(i=0; i<tam; i++)
        if(random[i]%2){
            impares[j] = random[i];
            j++;
        }

    imprimeArreglo(cont, impares, "impares");
}

/**
\brief Funcion que asigna los valores pares de un arreglo a
otro.
\param random Apuntador a entero con direccion del arreglo
completo.
\param pares Apuntador a entero con direccion del arreglo
al que solo se le quiere asignar valores pares.
\param tam Entero con valor del tamaño del arreglo.
\param cont Entero con valor del número de números pares en
el arreglo completo.
*/
void numPares(int* random, int* pares, int tam, int cont){
    int j=0, i=0;
    for(i=0; i<tam; i++)
        if(!(random[i]%2)){
            pares[j] = random[i];
            j++;
        }

    imprimeArreglo(cont, pares, "pares");
}

/**
\brief Funcion que suma todos los elementos de un arreglo.
\param array Apuntador a entero con direccion del arreglo.

```

```
\param tam Entero con valor del tamaño del arreglo.  
\return suma Entero con el valor de la suma de los  
        elementos.  
*/  
int sumaElementos(int* array, int tam){  
    int suma=0, i;  
    for(i=0; i<tam; i++)  
        suma+=array[i];  
    return suma;  
}
```

Archivo Makefile

```
1  compile: Src/programa.c
2      $(CC) Src/programa.c -o Src/progra
3
4  correr:
5      Src/progra
6
```

Y finalmente crearemos el archivo README.txt con el siguiente texto:

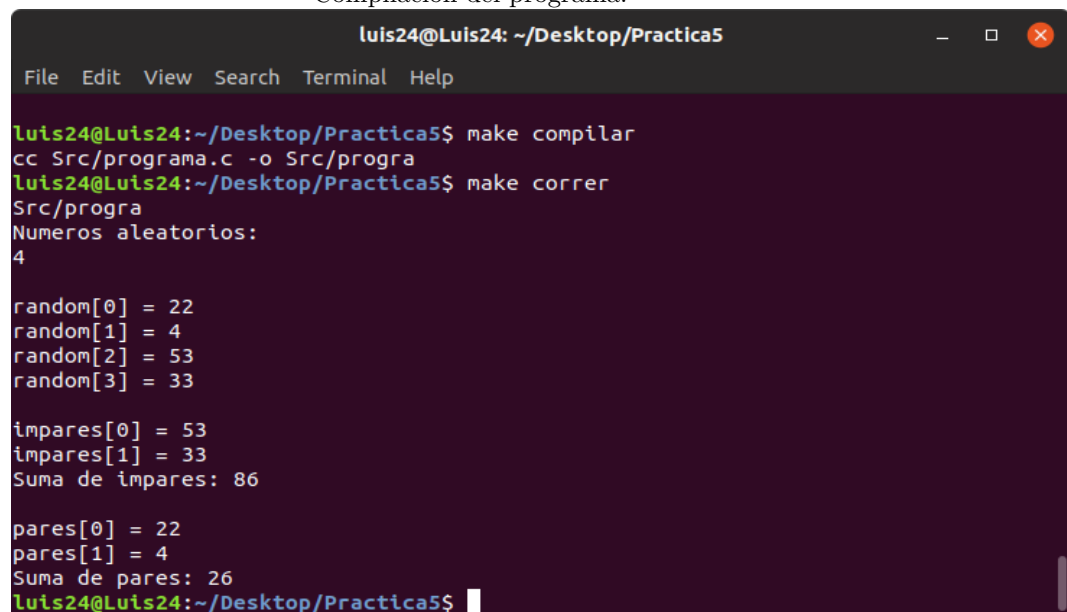
Archivo README

```
1  COMPILACION DEL PROGRAMA:
2      make compile
3
4  EJECUCION:
5      make correr
6
```

Este archivo servirá como manual para el usuario para correr los programas de manera más fácil. Este archivo, junto con el makefile, irán en la raíz de la carpeta como se muestra en la primera imagen de esta sección de desarrollo.

Posteriormente para compilar el programa se hará de la siguiente forma:

Compilación del programa:



```
luis24@Luis24: ~/Desktop/Practica5
File Edit View Search Terminal Help

luis24@Luis24:~/Desktop/Practica5$ make compile
cc Src/programa.c -o Src/progra
luis24@Luis24:~/Desktop/Practica5$ make correr
Src/progra
Numeros aleatorios:
4

random[0] = 22
random[1] = 4
random[2] = 53
random[3] = 33

impares[0] = 53
impares[1] = 33
Suma de impares: 86

pares[0] = 22
pares[1] = 4
Suma de pares: 26
luis24@Luis24:~/Desktop/Practica5$
```

Y se verá de la siguiente forma la ejecución:

```
luis24@Luis24: ~/Desktop/Practica5/Src
File Edit View Search Terminal Help
Numeros aleatorios:
10

random[0] = 17
random[1] = 22
random[2] = 71
random[3] = 57
random[4] = 22
random[5] = 47
random[6] = 38
random[7] = 10
random[8] = 22
random[9] = 11

impares[0] = 17
impares[1] = 71
impares[2] = 57
impares[3] = 47
impares[4] = 11
Suma de impares: 203

pares[0] = 22
pares[1] = 22
pares[2] = 38
pares[3] = 10
pares[4] = 22
Suma de pares: 114
luis24@Luis24:~/Desktop/Practica5/Src$
```

4. Bibliografía

Referencias

- [1] <https://www.programacion.com.py/escritorio/c/pipes-en-c-linux>
- [2] <https://www.geeksforgeeks.org/c-program-demonstrate-fork-and-pipe/>