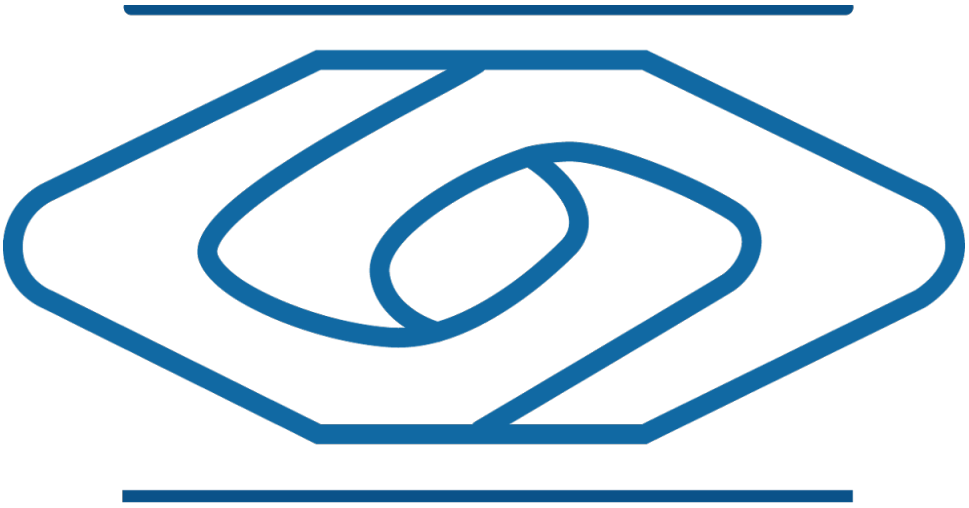

Instituto Politécnico Nacional
Escuela Superior de Computo

Práctica 5: Pipes & Fork

Reporte
Profesor: Ulises Velez Saldaña
Alumno: Meza Madrid Raúl Damián
Clase: Sistemas operativos
Grupo: 2CM7

Contents

Introducción

1.1 Procesos

La principal entidad activa en un sistema Linux es el proceso . Linux es un sistema multiprograma , por lo tanto, múltiples procesos independientes pueden estar corriendo al mismo tiempo. Los procesos son creados en una manera simple. El proceso que ejecuta la llamada al sistema *fork* se llama proceso padre. El nuevo proceso es llamado proceso hijo. El padre y el hijo tienen su propia memoria privada. Si el padre si el padre subsecuentemente cambia cualquiera de sus variables, los cambios no son reflejados en el hijo y vice versa.

1.2 Pipe

Un pipe es una especia de pseudo archivo que puede ser usado para conectar dos procesos. Si un proceso A y N desean comunicarse usando un pipe, necesita escribir y leer del pipe como si fuera un archivo de entrada y salida, de esta manera la comunicación entre dos procesos dentro de UNIX se ve muy similar a una lectura/escritura en archivos.

1.3 Programas y herramientas utilizados

Esta práctica fue desarrollada en el sistema operativo Ubuntu 18.04.1 LTS. Estos son los programas y herramientas utilizados, junto con el comando de instalación, en caso de que no estuvieran instalados ya.

- Doxygen

```
git clone https://github.com/doxygen/doxygen.git
cd doxygen
mkdir build
cd build
cmake -G "Unix Makefiles" ..
make
make install
```

- make
- cmake
- python

Objetivo

Que el alumno aplique la teoría vista en clase implementando la comunicación entre un proceso y sus hijos a través de distintos pipelines. Para esto se creara un programa en C que generará n números aleatorios, escribirá los pares en el pipe para un proceso hijo que se encargara de sumarlos y después lo regresa al proceso padre para que el lo imprima, hará lo mismo para un proceso encargado de los números impares.

Desarrollo

El primer paso es leer los números del parámetro argv[ ]. Se generaran los n números, después se decidira cuales se escriben en que pipe.
Será necesario mantener seguimiento del numero de pares e impares que existen para que cada hijo sepa cuantas veces leer.
Para lograr la comunicación con ambos procesos se generaran dos pipes correspondientes; pipeImp para los números impares y pipePar para los números pares, tal y como se muestra en el siguiente diagrama.

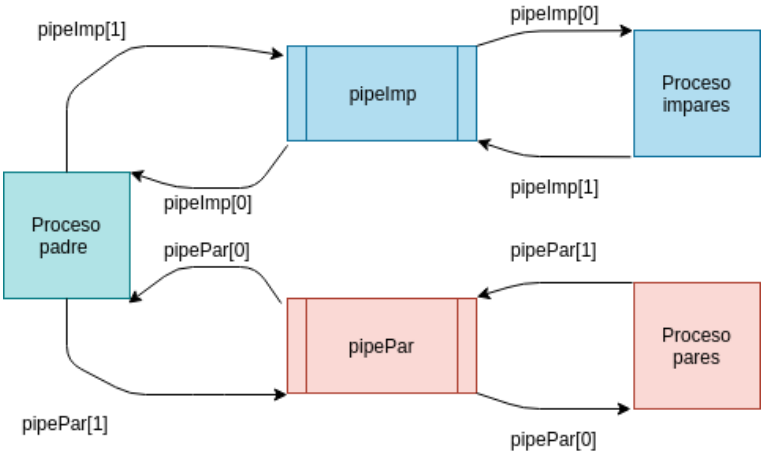

Figure 1: Comunicación entre proceso padre(izquierda) e hijos (derecha)

3.1 Codigo

El codigo fuente contiene comentarios que describen el programa. Son utilizados tambien para documentar con doxygen.

3.1.1 Código fuente

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

int main(int argc, char const *argv[]) {
  /// Se crean los dos pipes y las variables auxiliares par leer y guardar la suma de los numeros pares e impares. Si alguno falla, termina el programa
  /// \code
  int buf ;
  int pipePar[2], sumpar=0, parC=0;
  int pipeImp[2], sumimp=0, impC=0;
  if (pipe(pipePar)==-1 || pipe(pipeImp)==-1) {
    printf("Error al crear los pipes \n");
    return 1;
  }
  /// \endcode

  /// Se crea un arreglo de integers de tamano igual al primer parametro en argv[]
  /// \code
  char *p;
  int n  = strtol(argv[1],&p,10);
  int *nums = (int*)malloc(sizeof(int)*n);
  srand(time(NULL));
  for (unsigned char i = 0; i < n; i++) {
    nums[i]=rand()%10;
    if (nums[i]%2) {
      impC++;
    }else{
      parC++;
    }
```

```
32      printf("Padre (%d): num alteatorio %d:
        ↪  %d\n",getpid(),i+1,nums[i]);
33    }
34    /// \endcode
35
36    /// El padre hace 2 forks
37    /// \code
38    pid_t cpid;
39    cpid = fork();
40    if (cpid<0) {
41      return 1;
42    }
43    if (cpid) {
44      cpid = fork();
45        if (cpid<0) {
46          return 1;
47        }
48    /// \endcode
49
50    /// Despues el padre escribe los numeros en su pipe
      ↪  correspondiente
51    /// \code
52        if (cpid) {
53          for (size_t i = 0; i < n; i++) {
54            if (nums[i]%2==0) {
55              write(pipePar[1],&nums[i], sizeof(int));
56            }else{
57              write(pipeImp[1],&nums[i], sizeof(int));
58            }
59          }
60    /// \endcode
61
62    /// Cierra los pipes y espera una respeusta
63    /// \code
64          close(pipePar[1]);
65          close(pipeImp[1]);
66          wait(NULL);
67          wait(NULL);
68    /// \endcode
69
```

```
70    /// Se leen e imprimen la respuestas correspondientes de cada
      ↪ pipe , despues se cierran
71    /// \code
72          read(pipePar[0],&sumpar, sizeof(int));
73          close(pipePar[0]);
74          printf("Padre (%d): Suma pares %d \n",getpid() ,sumpar);
75          read(pipeImp[0],&sumimp, sizeof(int));
76          close(pipeImp[0]);
77          printf("Padre (%d): Suma impares: %d \n",getpid(), sumimp);
78          /// \endcode
79
80          /// El segundo hijo (innermost) se dedica a sumar los
            ↪ numeros pares segun el contador, cierra el pipe
            ↪ despues.
81          /// \code
82        }else{
83          for (size_t i = 0; i < parC; i++) {
84            read(pipePar[0],&buf, sizeof(int));
85            printf("Hijo de pares(%d) : sumando %d \n",getpid(),buf );
86            sumpar += buf;
87          }
88          close(pipePar[0]);
89    /// \endcode
90
91    /// Escribe el resultado en el pipe, cierra el pipe y termina el
      ↪ proceso hijo con estado (EXIT_SUCCESS); para que el padre
      ↪ salga de wait(NULL)
92    /// \code
93          write(pipePar[1], &sumpar, sizeof(sumpar));
94          close(pipePar[1]);
95          exit(EXIT_SUCCESS);
96        }
97    /// \endcode
98
99    /// El el primer hijo hace lo mismo, pero para sus numeros
      ↪ correspondientes (impares)
100   /// \code
101     }else{
102       for (size_t i = 0; i < impC; i++) {
103         read(pipeImp[0],&buf, sizeof(int));
```

```
104         printf("Hijo de impares(%d) : sumando %d \n",getpid(),buf );
105         sumimp += buf;
106       }
107       close(pipeImp[0]);
108       write(pipeImp[1], &sumimp, sizeof(sumimp));
109       close(pipeImp[1]);
110       exit(EXIT_SUCCESS);
111   }
112   /// \endcode
113   return 0;
114 }
```

Resultados

El programa funciona de manera adecuada. A continuacion se muestran dos test case para ilustrar la salida del programa.

4.1 Caso de prueba: 5

```
1  Padre (10583): num alteatorio 1: 5
2  Padre (10583): num alteatorio 2: 2
3  Padre (10583): num alteatorio 3: 5
4  Padre (10583): num alteatorio 4: 7
5  Padre (10583): num alteatorio 5: 2
6  Hijo de impares(10584) : sumando 5
7  Hijo de impares(10584) : sumando 5
8  Hijo de impares(10584) : sumando 7
9  Hijo de pares(10585) : sumando 2
10 Hijo de pares(10585) : sumando 2
11 Padre (10583): Suma pares 4
12 Padre (10583): Suma impares: 17
```

4.2 Caso de prueba: 15

```
1  Padre (10641): num alteatorio 1: 6
2  Padre (10641): num alteatorio 2: 5
3  Padre (10641): num alteatorio 3: 0
4  Padre (10641): num alteatorio 4: 9
5  Padre (10641): num alteatorio 5: 9
```

```
Padre (10641): num alteatorio 6: 2
Padre (10641): num alteatorio 7: 0
Padre (10641): num alteatorio 8: 9
Padre (10641): num alteatorio 9: 7
Padre (10641): num alteatorio 10: 3
Padre (10641): num alteatorio 11: 3
Padre (10641): num alteatorio 12: 7
Padre (10641): num alteatorio 13: 8
Padre (10641): num alteatorio 14: 9
Padre (10641): num alteatorio 15: 3
Hijo de impares(10642) : sumando 5
Hijo de impares(10642) : sumando 9
Hijo de impares(10642) : sumando 9
Hijo de impares(10642) : sumando 9
Hijo de impares(10642) : sumando 7
Hijo de impares(10642) : sumando 3
Hijo de pares(10643) : sumando 6
Hijo de impares(10642) : sumando 3
Hijo de impares(10642) : sumando 7
Hijo de pares(10643) : sumando 0
Hijo de impares(10642) : sumando 9
Hijo de pares(10643) : sumando 2
Hijo de impares(10642) : sumando 3
Hijo de pares(10643) : sumando 0
Hijo de pares(10643) : sumando 8
Padre (10641): Suma pares 16
Padre (10641): Suma impares: 64
```

Cabe notar que en el ultimo ejemplo se puede apreciar como los procesos hijos están corriendo de manera simultanea.

Errores y problemas

La primera versión enviaba en el pipe el arreglo entero de números, y cada hijo determinaba cuales eran pares y cuales eran impares, después cuando se trato de enviar solamente los números pares e impares dentro del pipe, el programa se congelaba esperando una lectura del pipe con *while(read(pipeImp[0],buf, sizeof(int))>0)* entonces se opto por crear una variable que contara la cantidad de números pares e impares que se debían leer. Como dichos contadores existen desde antes del fork, los dos procesos hijos tienen el contador respectivo.

Codigo (Github)

Todo el codigo de esta practica se puede encontrar en :`https://github.com/asdf1234Damian/Operating-Systems/tree/master/Practica05`

References

[1] Michael Kerrisk. pipe(7) - Linux manual page. `http://man7.org/linux/man-pages/man7/pipe.7.html`. [Online; consultado en 7 de abril 2019].

[2] Andrew S. Tanenbaum and Garcîa Roberto Escalona. *Sistemas operativos modernos*. Pearson Educación, 2 edition, 2003.