UNIVERSIDAD SIGLO 21

MARCO TEÓRICO COMUNICACIÓN ENTRE PROCESOS EN C/C++

Algoritmos concurrentes y paralelos

Dr. Miguel Méndez Garabetti, Ing. Eduardo Piray

10/07/2023

Marco teórico de soporte para la realización de los trabajos prácticos.

Contenido

1. Comunicación entre procesos

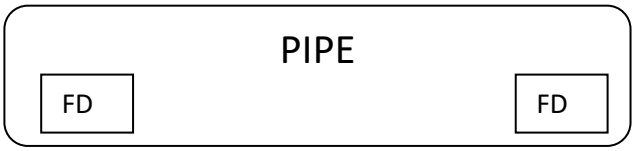
La comunicación entre procesos (IPC) es un mecanismo que permite que los procesos se comuniquen entre sí y sincronicen sus acciones. La comunicación entre estos procesos puede verse como un método de cooperación entre ellos.

Mencionemos dos métodos para comunicar procesos:

1. **Pipe (tubería)**: pipe es un medio de comunicación entre dos o más procesos relacionados o interrelacionados. Puede ser dentro de un proceso o una comunicación entre el proceso secundario y el principal. La comunicación también puede ser de varios niveles, como la comunicación entre el proceso padre, el proceso hijo y el proceso hijo del hijo (nieto), etcétera. La comunicación se logra mediante un proceso que escribe en pipe y otro que lee desde pipe.
2. **Sockets**: este método se utiliza principalmente para comunicarse a través de una red entre un cliente y un servidor. Permite una conexión estándar que es independiente de la computadora y el sistema operativo.

2. Pipes

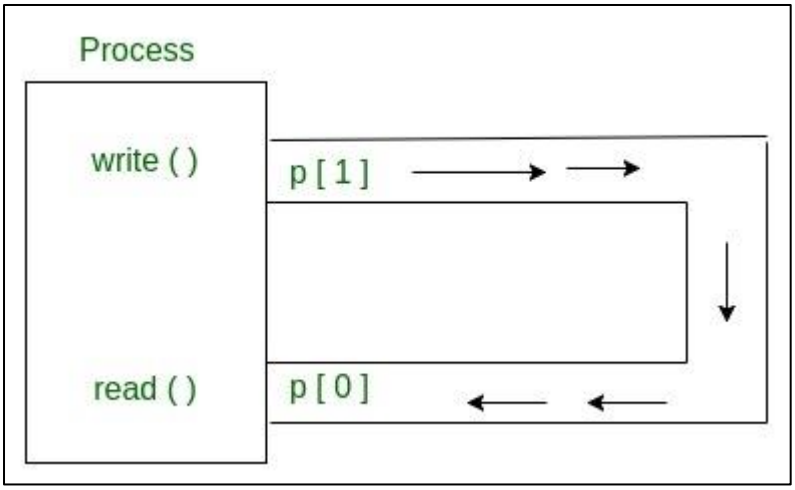
Muchas veces dos procesos necesitan comunicarse entre sí. Una funcionalidad que permite dicha comunicación se conoce como **pipes**. Pipe (tubo, tubería) es un archivo en memoria (buffer), en el cual se puede tanto escribir (write) como leer (read). Se compone de dos files **descriptors**.



Conceptualmente, pipe es una **conexión entre dos procesos**, la salida estándar de un proceso se convierte en la entrada estándar del otro proceso.

Pipe es un **canal de comunicación unidireccional**, es decir, podemos usar pipe de modo que un proceso escriba en el pipe y el otro proceso lea desde el pipe. Abrir un pipe, es un área de la memoria principal que se trata como un "**archivo virtual**".

Pipe puede ser utilizada por el proceso de creación, así como por todos sus procesos hijos, para leer y escribir. Un proceso puede escribir en este pipe o "archivo virtual" y otro proceso relacionado puede leerlo. Si un proceso intenta leer antes de que se escriba algo en el pipe, el proceso se suspende hasta que se escriba algo.



Fuente: https://media.geeksforgeeks.org/wp-content/uploads/Process.jpg

Los pipes son propios de los sistemas operativos UNIX.

2.1. Uso de pipe para comunicar dos procesos[1]

Para utilizar un pipe en C, es necesario usar la llamada al sistema **pipe()**. Esta llamada al sistema tiene como argumento un **arreglo** que se conforma de dos enteros, estos enteros son los **file descriptor** (archivos descriptores) para este pipe. Un **file descriptor** es una clave (key) para acceder a un archivo, este archivo es de donde se pretende leer o escribir datos. La forma general de la llamada al sistema pipe() es:

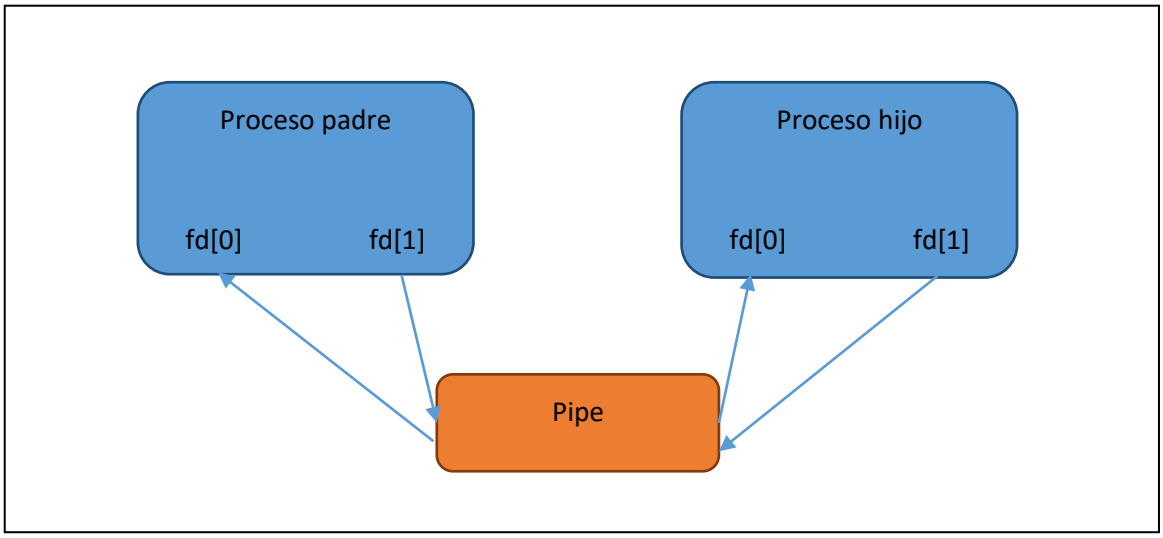
```
int pipe(int fd[2]);
```

- El argumento es el arreglo con dos files descriptors: **fd[0]** es el file descriptor para lectura, y **fd[1]** es el file descriptor para escritura.
- La función retorna 0 si es exitosa, y -1 si hay algún error.

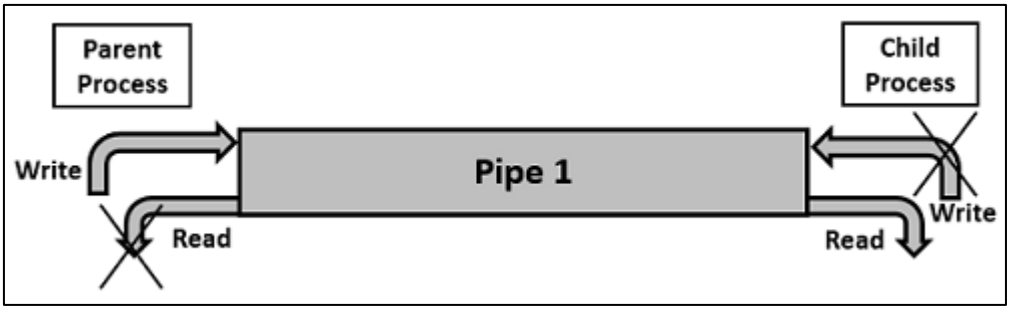
Como concepto general podemos decir que un **file descriptor** es un archivo entero que identifica de forma única un archivo abierto de un proceso.

Cuando usamos fork() en cualquier proceso, los descriptores de archivo permanecen abiertos en el proceso hijo y también en el proceso principal/padre. Si llamamos a fork() después de crear un pipe, entonces el padre y el hijo pueden comunicarse a través del pipe.

[1] https://linux.die.net/man/2/pipe



Otra manera de ver la comunicación entre procesos por medio de un pipe es la siguiente:



Fuente: https://www.tutorialspoint.com

Cada proceso tiene los dos file descriptor (dos por cada proceso, haciendo un total de 4 files descriptors), pero los files descriptors de un proceso son **independientes** del otro proceso, y se cerrarán y abrirán de forma independiente entre los dos procesos. Si, por ejemplo, cierro un file descriptor que está abierto en un proceso, en el otro proceso el file descriptor equivalente permanecerá abierto.

Pipes se comportan como una estructura de datos de cola o FIFO (primero en entrar, primero en salir). El tamaño de lectura y escritura no tiene que coincidir. Podemos escribir 512 bytes a la vez, pero solo podemos leer 1 byte a la vez en pipe.

Las Funciones **read** y **write** se utilizan para leer y escribir en pipes.

Ejemplo: un programa con un proceso padre y un proceso hijo, el hijo va a escribir un número entero, el proceso padre va a leer ese entero y lo multiplicará por 3. Usaremos la llamada fork() para crear el proceso hijo.

El código es el siguiente:

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

int main(int argc, char* argv[])
{
      int fd[2]; //file descriptor para lectura fd[0] y escritura fd[1]

      if (pipe(fd) == -1){  //abrir pipe y manejo de error
            printf("Ha ocurrido un error al abrir el pipe\n");
            return 1; // retorno por si hay algún error
      }

      int id = fork();//llamada fork crear proceso

      if(id == -1){ //manejo de error en fork
            printf("Ha ocurrido un error con fork\n");
            return 2;
      }



      if(id == 0){ //proceso hijo
            close(fd[0]); //cierro el file descriptor de lectura, no se va
            leer en este proceso hijo
            int x;
            printf("Ingrese un número: ");
            scanf("%d", &x);
            write(fd[1], &x, sizeof(int)); //escribir en el file descriptor,
            close(fd[1]); //cierro el file descriptor de escritura
      } else { //proceso padre
            int y;
            read(fd[0], &y, sizeof(int)); //leo el file descriptor de lectura
            close(fd[0]); //cierro el file descriptor de lectura
            printf("Dato obtenido del proceso hijo %d", y * 3);//salida en
pantalla
      }
```

```
	return 0;
}
```

La salida en pantalla para x = 10 es:

```
Ingrese un número: 10

Dato obtenido del proceso hijo 30
```

Lo que ha hecho el programa es que se ha escrito el entero 10 desde el proceso hijo al file descriptor, y luego desde el proceso padre se leyó dicho entero desde el file descriptor, se lo multiplicó por 3 y se imprimió el resultado del producto en consola.

Se puede realizar un manejo de errores al leer y escribir, colocando las llamadas de entrada/salida write y read dentro de una sentencia if. Es decir:

```
if (write(fd[1], &x, sizeof(int)) == -1){

	printf("Ha ocurrido un error al escribir en el pipe\n");
	return 3;

}

if (read(fd[0], &y, sizeof(int)) == -1){

	printf("Ha ocurrido un error al leer del pipe\n");
	return 4;

}
```

2.2. Ejercicio de aplicación de pipe

Definimos un arreglo de enteros, y vamos a sumar sus elementos. Se va a dividir el arreglo en dos partes (a la mitad), una mitad de los elementos será trabajada por uno de los procesos, y la otra mitad de elementos por el otro proceso. Ambos procesos obtendrán una suma parcial, uno de los procesos escribirá su suma parcial en el pipe, y el otro proceso leerá esa suma parcial del pipe y obtendrá la suma total para mostrarla en pantalla.

Para efectuar las sumas parciales, cada proceso debe saber por cual elemento del arreglo debe empezar a sumar, y por cual elemento debe terminar su suma parcial.

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

int main(int argc, char* argv[])
{
	int arr[] = {4,5,7,3,1,8};
```

```
        int inicio, final; //índices para sumas parciales
        int longArr = sizeof(arr) / sizeof(int); //cantidad elementos
        int fd[2]; //file descriptor

        if (pipe(fd) == -1){  //abrir pipe y manejo de error
                printf("Ha ocurrido un error al abrir el pipe\n");
                return 1;
        }

        int id = fork();//crear proceso

        if(id == -1){ //manejo de error en fork
                printf("Ha ocurrido un error con fork\n");
                return 2;
        }


        if(id == 0){ //proceso hijo
                inicio = 0;
                final = longArr / 2;

        } else { //proceso padre
                inicio = longArr / 2;
                final = longArr;
        }

        int  suma = 0;
        int i;
        for(i = inicio; i < final; i++){
                suma += arr[i];
        }

        printf("Suma parcial calculada: %d\n", suma);

        return 0;
}
```

Si ejecutamos el código anterior se obtienen las dos sumas parciales hechas por cada proceso:

```
Suma parcial calculada:12
Suma parcial calculada:16
```

Se busca mostrar cómo se divide el trabajo de sumar entre dos procesos. Queda calcular la suma total. Para ello hay que enviar la suma parcial de uno de los procesos al otro proceso. Para ello, se debe usar un pipe.
Al código anterior hay que añadirle antes del return 0, lo siguiente:

```
    if (id == 0) { //proceso hijo
        close(fd[0]); //cierro lectura
        if (write(fd[1], &suma, sizeof(int)) == -1) {
            return 3;
        }
        close(fd[1]); //cierro escritura
    } else { //proceso padre
        int sumaDesdeHijo;
        close(fd[1]); //cierro escritura
        if (read(fd[0], &sumaDesdeHijo, sizeof(int)) == -1) {
            return 4;
        }
        close(fd[0]); //cierro lectura

        int totalSuma = suma + sumaDesdeHijo;
        printf("La suma total es %d\n", totalSuma);
        wait(NULL);
    }
```

De esta manera, ambos procesos se comunican por medio de pipe para realizar la suma.

2.3. Otro ejemplo de aplicación pipes

Enviar un arreglo usando pipes entre dos procesos. La clave es pasar el tamaño del arreglo (cantidad de elementos del arreglo) y luego el arreglo propiamente dicho. El proceso hijo envía el arreglo al proceso padre. El proceso padre recibe el arreglo y suma sus elementos, para finalmente mostrar el resultado de la suma.

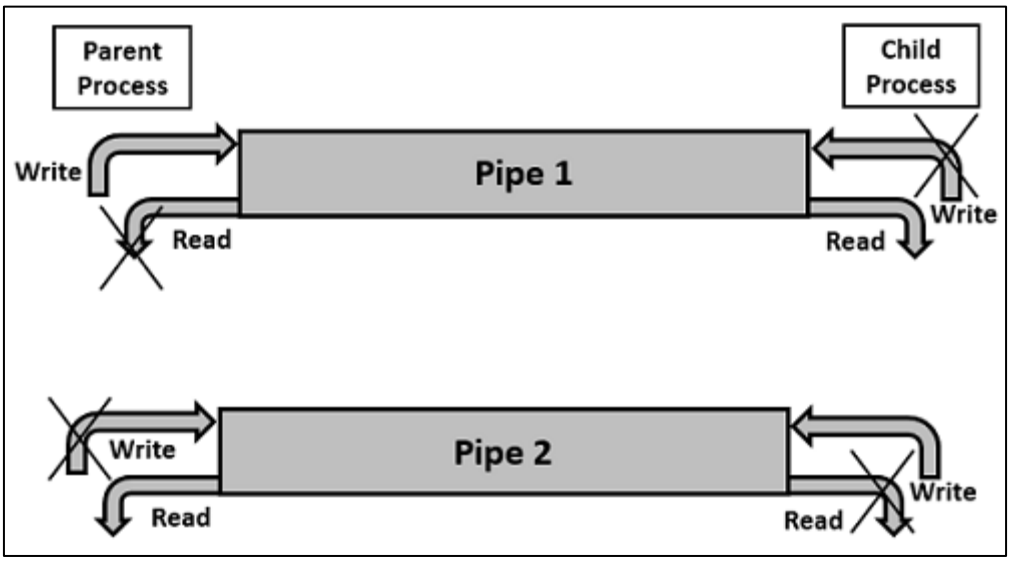
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>

int main(int argc, char* argv[]) {
    int fd[2]; //file descriptor
    if (pipe(fd) == -1) { //llamada a pipe y manejo de error
        return 1;
    }

    int pid = fork(); //llamada a fork() creo proceso hijo
    if (pid == -1) {
        return 2;
    }

    if (pid == 0) {
        //Proceso hijo
        close(fd[0]); //cerrar lectura
```

```
        int n, i;
        int arr[10];

        srand(time(NULL)); //semilla
        n = rand() % 10 + 1;

        printf("Generated: ");
        for (i = 0; i < n; i++) {
            arr[i] = rand() % 11;
            printf("%d ", arr[i]);
        }
        printf("\n");

        if (write(fd[1], &n, sizeof(int)) < 0) { //enviar tamaño arreglo
            return 3;
        }
        printf("Sent n = %d\n", n);

        if (write(fd[1], &arr, sizeof(int) * n) < 0) { //enviar arreglo
            return 4;
        }

        printf("Sent array\n");
        close(fd[1]); //cerrar escritura
    } else {
        //Proceso padre
        close(fd[1]); //cerrar escritura
        int arr[10];
        int n, i, sum = 0;

        if (read(fd[0], &n, sizeof(int)) < 0) { //leer cantidad
            return 5;
        }
        printf("Received n = %d\n", n);
        if (read(fd[0], &arr, sizeof(int) * n) < 0) { //leer arreglo
            return 6;
        }
        printf("Received array\n");

        close(fd[0]); //cerrar arreglo
        for (i = 0; i < n; i++) {
            sum += arr[i];
        }
        printf("Result is %d\n", sum);
        wait(NULL);
    }

    return 0;
}
```

3. Comunicación bidireccional entre procesos con pipes

Como ya se mencionó, la llamada al sistema pipe() se usa para pasar información de un proceso a otro. La llamada pipe() es unidireccional, es decir, el proceso principal escribe y el proceso secundario lee o viceversa, pero no ambos.

Sin embargo, ¿qué sucede si tanto el proceso padre como el proceso hijo necesitan escribir y leer de pipes simultáneamente? La solución es una comunicación bidireccional mediante pipes. Se requieren dos pipes para establecer una comunicación bidireccional entre procesos, uno para cada dirección de la comunicación.



Fuente: https://www.tutorialspoint.com/index.htm

3.1. Ejemplo de aplicación de comunicación bidireccional con pipes

Se analiza un algoritmo en el que se crean dos procesos, que podemos llamar P1 y P2. P1 toma un string y se lo envía a P2. El proceso P2 concatena el string recibido con otra cadena (sin usar la función de string) y la envía de regreso al proceso P1 para que se imprima en pantalla. Para este algoritmo se usarán dos pipes, y por ende dos arreglos de files descriptors, y se usará fork() para crear el proceso hijo.

El algoritmo buscar concatenar cadenas de la siguiente manera, por ejemplo:

Entrada	Otra cadena	Salida
www	google.com	www.google.com
www.argentina	.gob.ar	www.argentina.gob.ar

Se usarán dos pipes, en el primero se envía un string de entrada desde el proceso padre al hijo. En el segundo pipe se envía un string concatenado desde el proceso hijo al padre.

El código completo es:

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


int main()
{
    int fd1[2]; // Usado para almacenar dos extremos del 1er. pipe
    int fd2[2]; // Usado para almacenar dos extremos del 2do. pipe
    char fixed_str[] = ".gob.ar"; //string fijo
    char input_str[100]; //string de entrada
    pid_t p;

    if (pipe(fd1) == -1) { //manejo de error
        fprintf(stderr, "Fallo de Pipe");
        return 1;
    }
    if (pipe(fd2) == -1) { //manejo de error
        fprintf(stderr, " Fallo de Pipe ");
        return 1;
    }

    scanf("%s", input_str); //espera ingreso de cadena

    p = fork(); //llamada fork()
    if (p < 0) { //manejo de error en fork
        fprintf(stderr, "Falla de fork");
        return 1;
    }

    // Proceso padre
    else if (p > 0) {
        char concat_str[100];

        close(fd1[0]); // Cerrar la lectura del primer pipe

        // Escribir string de entrada y cerrar escritura en primer pipe
        write(fd1[1], input_str, strlen(input_str) + 1);
        close(fd1[1]);

        // Esperar que el proceso hijo envíe una cadena
        wait(NULL);

        close(fd2[1]); // Cerrar escritura del segundo pipe
```

```
        // Leer string del proceso hijo, imprimir y cerrar lectura 2do. pipe
        read(fd2[0], concat_str, 100);
        printf("Cadena concatenada %s\n", concat_str);
        close(fd2[0]); //Cerrar lectura del Segundo pipe
    }

    // proceso hijo
    else {
        close(fd1[1]); // Cerrar escritura en primer pipe

        // Leer un string usando el primer pipe
        char concat_str[100];
        read(fd1[0], concat_str, 100);

        // Concatenar una cadena fija
        int k = strlen(concat_str);
        int i;
        for (i = 0; i < strlen(fixed_str); i++)
            concat_str[k++] = fixed_str[i];

        concat_str[k] = '\0'; // string termina con '\0'

        // Cerrar ambos finales de lectura
        close(fd1[0]);
        close(fd2[0]);

        // Escribir string concatenado y cerrar final de escritura
        write(fd2[1], concat_str, strlen(concat_str) + 1);
        close(fd2[1]);

        exit(0);
    }
}
```

La salida en pantalla será con dos mensajes uno del proceso principal y otro del proceso hijo:

```
Cadena concatenada: www.argentina.gob.ar
```

La explicación del funcionamiento del algoritmo es el siguiente:

- Crear dos pipes: en el primero el padre escribe y el hijo lee. En el segundo pipe el hijo escribe y el padre lee.
- El **proceso principal/padre** en primera instancia cierra el extremo de lectura del primero de los pipes (fd1 [0]) y luego escribe la cadena a través del extremo de escritura del pipe (fd1 [1]). El padre esperará hasta que finalice el proceso hijo con wait.

- El **proceso hijo** leerá la primera cadena enviada por el proceso padre cerrando el extremo de escritura del primer pipe (fd1 [1]) y, después de leer, concatenará ambas cadenas y pasará la cadena al proceso padre escribiendo en el segundo pipe fd2 [1] y finalizará.
- Después que finaliza el proceso hijo (por ello se usa el wait()), el **proceso padre** cerrará el extremo de escritura del segundo pipe (fd2[1]) y leerá la cadena del extremo de lectura de este segundo pipe (fd2[0]), para finalmente imprimir la cadena completa.

3.2. Crear múltiples pipes

Por ejemplo, para crear 10 pipes en C, una alternativa es usar una matriz int de 2 dimensiones como se muestra a continuación.

```
int fds[10][2];
for(i=0; i<10; i++){
   pipe(fds[i]);
}
```

Donde fds[i][0] representa el final de lectura y fds[i][1] representa el final de escritura.

4. Ejercicio de aplicación: simulación de agente de viajes en C++[2]

El programa genera 6 "agentes de viajes" como procesos secundarios de la función principal (proceso padre o principal). Cada proceso tiene acceso a los asientos de un vuelo y puede reservarlos. El programa protegerá los asientos de otros agentes de viajes para evitar problemas de sobreventa. Esto se logra utilizando comunicación entre procesos mediante pipes.

```
#include<iostream>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
using namespace std;

const int NUMBER_OF_CHILDREN = 6;  //constante con el número de agente de viajes

const int PARENT_CONTROL = (2 - NUMBER_OF_CHILDREN); // constant para control

const int SIZE_OF_INT = sizeof(int);  //parámtro requerido para enviar enteros por el pipe

const int INITIAL_NUMBER_OF_SEATS = 100;  //constant con el número de asientos iniciales del vuelo

//declaración de la función travel agent
void travel_agent (int* child_to_parent, int* parent_to_child, int id);

int main(){

        int child_to_parent[2]; //arreglo file descriptor del primer pipe
        int parent_to_child[2]; //file descriptor del segundo pipe
        pipe(child_to_parent); //primer pipe
        pipe(parent_to_child); //segundo pipe


        pid_t shut_down[NUMBER_OF_CHILDREN];  //arreglo de 6 pidID para unir al final del programa

        int seats_left = INITIAL_NUMBER_OF_SEATS; //asientos restantes de reservar

        int pid; //para realizar un seguimiento si estamos en el proceso padre del hijo por cada fork()

         //ciclo for para crear los agentes de viajes
         for (int i = 0; i < NUMBER_OF_CHILDREN; i++){

                pid = fork(); //llamada al Sistema fork

                if (pid < 0){ //manejo de error:
                        cout << "OH SNAP! Child " << i <<  " failed";
```

[2] Por Steve Reilander. SIMON FRASER UNIVERSITY. Canadá. https://www.sfu.ca/

```
                return -1;
        }


        // Si el pid es 0, estamos en el proceso hijo
        if (pid == 0 ) {

                //Salida que muestra cuando un proceso hijo es creado
                cout << "CHILD: " << i << " CREATED " <<  getpid() << endl;

                //iniciar la función travel_agent
            travel_agent(child_to_parent, parent_to_child, i);
            break; //cortar generación de más hijos (hijos de hijos)
        }

        else{
                shut_down[i] = pid; //arreglo de pids para esperar que finalicen los hijos
        }
} //fin del ciclo for de creación de hijos.

// Si el pid es mayor a 0, se ejecuta el proceso padre
if (pid>0){
        bool loop = true;

        //in a loop
        while (loop){

                write(parent_to_child[1], &seats_left, SIZE_OF_INT);
                read(child_to_parent[0], &seats_left, SIZE_OF_INT);

            // si quedan asientos muestra cuantos quedan
            if (seats_left > 0)
                    cout << "the main just read seats_left: " << seats_left << endl;

            //Cuando seats_left < PARENT_CONTROL, todos los hijos terminaron
            if (seats_left < PARENT_CONTROL){
                    loop = false;
                    cout << "Parent exit loop. PID: " << getpid() << endl;
            }
        }

        //manejo de procesos Zombis.
        for (int i = 0; i < NUMBER_OF_CHILDREN; i++){
                cout << "Waiting for PID: " << shut_down[i] << " to finish" << endl;
                waitpid(shut_down[i], NULL, 0);
                cout << "PID: " << shut_down[i] << " has shut down" << endl;
        }
}
```

```
        //comprobar para ver todos los hijos terminados
        cout << "Did we all Join?  There will be 7 of us if we did.  PID: "  << getpid() << endl;
        if (pid>0)
                cout << "There are no ZOMBIES!" << endl;

        return 0;
} //fin main

//function para cada agente de viaje
void travel_agent (int* child_to_parent, int* parent_to_child, int id){
        bool loop = true;

        //do a loop
        while (loop){

                int seats_left = 0; //variable local seats_left
                //esperar a que el padre escribe, luego procesar los asientos restantes
                read(parent_to_child[0], &seats_left, SIZE_OF_INT); //lectura

                //si quedan asientos, escriba cuántos y luego 'reserve' un asiento,
                //declare thread number, and pid
                if (seats_left > 0){
                        cout << "I am Child: " << id << " There are: " << seats_left << " seats, booking
                one!";
                        cout << " My PID is : " << getpid() << endl;
                        seats_left--;

                        //Si no quedan asientos por reservar, detener el ciclo
                        if (seats_left == 0)
                        loop=false;

                        //Hijo escribe al padre cuantos asientos quedan y detener ejecución con usleep
                        write(child_to_parent[1], &seats_left, SIZE_OF_INT);
                        usleep(100); //detener ejecución por un intervalo de tiempo
                }

                //si no quedan asientos detener el bucle, y escribir nuevamente al padre
                else{
                        loop = false;
                        seats_left--;
                        write(child_to_parent[1], &seats_left, SIZE_OF_INT);
                }
        }
}
```

Consideraciones y explicación del algoritmo:

- El programa usa **seat_left** para saber cuándo el asiento del vuelo está reservado (asientos quedan libres). El algoritmo para esto requiere **seats_left = (2 - number of children)**
- La función que maneja el agente de viaje es **void travel_agent(…..){…..}**
- En la función **main** se define lo siguiente:
    - Se crean dos pipes para comunicar procesos hijos con el proceso padre y viceversa. Se definen como parámetros los files descriptors (arreglo de tamaño 2).
    - Si el pid es 0, estamos en el proceso hijo. Se comienza a ejecutar la función travel_agent. Es importante cortar con break en los procesos hijos del ciclo for para que no se generen procesos hijos de los hijos (nietos del proceso principal).
    - Si el pid es mayor que 0, el proceso padre se está ejecutando. Se construye un arreglo de cada pid, **shut_down[i] = pid**, para esperar que finalicen los hijos.
      En la parte de comunicación entre procesos de la función **main**, queremos asegurarnos de que solo un agente de viajes pueda leer la cantidad de asientos disponibles a la vez. También debemos asegurarnos de que solo un agente pueda reservar un asiento a la vez. Si no protegemos la variable **seat_left**, podremos reservar en exceso, y es lo que se pretende evitar. Realizamos la protección de la sección crítica con pipes. Cuando un proceso lee de un pipe, si no hay nada para leer, esperará a que se escriba algo antes de continuar. Cuando algo se escribe en un pipe, esos datos solo se pueden leer una vez y solo por un proceso. Si hay cinco procesos todos esperando para leer el mismo pipe y luego hay una escritura en el pipe, solo UNO de los cinco procesos en espera podrá leer lo que se escribió y continuar, mientras que los otros 4 seguirán esperando. Se escribe la variable actual de **seats_left** en el pipe de **parent_to_child**. Y en la función **travel_agent**, comenzarán leyendo desde el pipe **parent_to_child**. El agente de viajes, después de leer del pipe **parent_to_child**, reservará un asiento y luego escribirá el valor de los asientos restantes al padre mediante el pipe **child_to_parent**. Esto está obligando a todos los agentes de viajes a leer la variable **seat_left** uno a la vez, y no permite que la función **main** escriba los **seats_left** mientras un agente de viajes está reservando un asiento.
    - En la función main también se realiza un manejo de procesos **Zombies**. Un zombi es un proceso hijo que continúa ejecutándose después de que el padre ha terminado. Sin no se realiza el bucle for, es posible obtener zombis. Para eso se usa **waitpid**. waitpid obligará al proceso actual a esperar a que termine otro proceso antes de continuar. En el ciclo for, forzamos a main a esperar a que cada uno de sus hijos termine antes de continuar. En la última sección de la función main se encuentra la sección join del programa. Todos los procesos, padres e hijos, ejecutarán esta sección con el código waitpid, la última línea que se muestra siempre será “There are no ZOMBIES”, es decir, "No hay ZOMBIES".

Salida en pantalla para comprender el funcionamiento.

```
CHILD: 0 CREATED 1868

I am Child: 0 There are: 100 seats, booking one! My PID is : 1868

CHILD: 3 CREATED 1871
```

```
the main just read seats_left: 99
I am Child: 3 There are: 99 seats, booking one! My PID is : 1871
CHILD: 4 CREATED 1872
the main just read seats_left: 98
I am Child: 4 There are: 98 seats, booking one! My PID is : 1872
the main just read seats_left: 97
I am Child: 0 There are: 97 seats, booking one! My PID is : 1868
the main just read seats_left: 96
CHILD: 2 CREATED 1870
I am Child: 2 There are: 96 seats, booking one! My PID is : 1870
the main just read seats_left: 95
CHILD: 1 CREATED 1869
I am Child: 3 There are: 95 seats, booking one! My PID is : 1871
the main just read seats_left: 94
I am Child: 1 There are: 94 seats, booking one! My PID is : 1869
the main just read seats_left: 93
I am Child: 0 There are: 93 seats, booking one! My PID is : 1868
the main just read seats_left: 92
I am Child: 4 There are: 92 seats, booking one! My PID is : 1872
the main just read seats_left: 91
CHILD: 5 CREATED 1873
I am Child: 5 There are: 91 seats, booking one! My PID is : 1873
the main just read seats_left: 90
I am Child: 2 There are: 90 seats, booking one! My PID is : 1870
the main just read seats_left: 89
I am Child: 3 There are: 89 seats, booking one! My PID is : 1871
the main just read seats_left: 88
I am Child: 0 There are: 88 seats, booking one! My PID is : 1868
the main just read seats_left: 87
```

I am Child: 1 There are: 87 seats, booking one! My PID is : 1869

the main just read seats_left: 86

I am Child: 5 There are: 86 seats, booking one! My PID is : 1873

the main just read seats_left: 85

I am Child: 2 There are: 85 seats, booking one! My PID is : 1870

the main just read seats_left: 84

I am Child: 3 There are: 84 seats, booking one! My PID is : 1871

the main just read seats_left: 83

I am Child: 4 There are: 83 seats, booking one! My PID is : 1872

the main just read seats_left: 82

I am Child: 1 There are: 82 seats, booking one! My PID is : 1869

the main just read seats_left: 81

I am Child: 2 There are: 81 seats, booking one! My PID is : 1870

the main just read seats_left: 80

I am Child: 5 There are: 80 seats, booking one! My PID is : 1873

the main just read seats_left: 79

I am Child: 0 There are: 79 seats, booking one! My PID is : 1868

the main just read seats_left: 78

I am Child: 4 There are: 78 seats, booking one! My PID is : 1872

the main just read seats_left: 77

I am Child: 2 There are: 77 seats, booking one! My PID is : 1870

the main just read seats_left: 76

I am Child: 1 There are: 76 seats, booking one! My PID is : 1869

the main just read seats_left: 75

I am Child: 5 There are: 75 seats, booking one! My PID is : 1873

the main just read seats_left: 74

I am Child: 0 There are: 74 seats, booking one! My PID is : 1868

the main just read seats_left: 73

I am Child: 4 There are: 73 seats, booking one! My PID is : 1872

```
the main just read seats_left: 72
I am Child: 1 There are: 72 seats, booking one! My PID is : 1869
the main just read seats_left: 71
I am Child: 2 There are: 71 seats, booking one! My PID is : 1870
the main just read seats_left: 70
I am Child: 5 There are: 70 seats, booking one! My PID is : 1873
the main just read seats_left: 69
I am Child: 0 There are: 69 seats, booking one! My PID is : 1868
the main just read seats_left: 68
I am Child: 4 There are: 68 seats, booking one! My PID is : 1872
the main just read seats_left: 67
I am Child: 1 There are: 67 seats, booking one! My PID is : 1869
the main just read seats_left: 66
I am Child: 2 There are: 66 seats, booking one! My PID is : 1870
the main just read seats_left: 65
I am Child: 5 There are: 65 seats, booking one! My PID is : 1873
the main just read seats_left: 64
I am Child: 0 There are: 64 seats, booking one! My PID is : 1868
the main just read seats_left: 63
I am Child: 4 There are: 63 seats, booking one! My PID is : 1872
the main just read seats_left: 62
I am Child: 2 There are: 62 seats, booking one! My PID is : 1870
the main just read seats_left: 61
I am Child: 3 There are: 61 seats, booking one! My PID is : 1871
the main just read seats_left: 60
I am Child: 5 There are: 60 seats, booking one! My PID is : 1873
the main just read seats_left: 59
I am Child: 0 There are: 59 seats, booking one! My PID is : 1868
the main just read seats_left: 58
```

I am Child: 1 There are: 58 seats, booking one! My PID is : 1869

the main just read seats_left: 57

I am Child: 3 There are: 57 seats, booking one! My PID is : 1871

the main just read seats_left: 56

I am Child: 5 There are: 56 seats, booking one! My PID is : 1873

the main just read seats_left: 55

I am Child: 1 There are: 55 seats, booking one! My PID is : 1869

the main just read seats_left: 54

I am Child: 0 There are: 54 seats, booking one! My PID is : 1868

the main just read seats_left: 53

I am Child: 3 There are: 53 seats, booking one! My PID is : 1871

the main just read seats_left: 52

I am Child: 5 There are: 52 seats, booking one! My PID is : 1873

the main just read seats_left: 51

I am Child: 1 There are: 51 seats, booking one! My PID is : 1869

the main just read seats_left: 50

I am Child: 0 There are: 50 seats, booking one! My PID is : 1868

the main just read seats_left: 49

I am Child: 3 There are: 49 seats, booking one! My PID is : 1871

the main just read seats_left: 48

I am Child: 5 There are: 48 seats, booking one! My PID is : 1873

the main just read seats_left: 47

I am Child: 2 There are: 47 seats, booking one! My PID is : 1870

the main just read seats_left: 46

I am Child: 4 There are: 46 seats, booking one! My PID is : 1872

the main just read seats_left: 45

I am Child: 3 There are: 45 seats, booking one! My PID is : 1871

the main just read seats_left: 44

I am Child: 0 There are: 44 seats, booking one! My PID is : 1868

```
the main just read seats_left: 43
I am Child: 1 There are: 43 seats, booking one! My PID is : 1869
the main just read seats_left: 42
I am Child: 2 There are: 42 seats, booking one! My PID is : 1870
the main just read seats_left: 41
I am Child: 4 There are: 41 seats, booking one! My PID is : 1872
the main just read seats_left: 40
I am Child: 3 There are: 40 seats, booking one! My PID is : 1871
the main just read seats_left: 39
I am Child: 1 There are: 39 seats, booking one! My PID is : 1869
the main just read seats_left: 38
I am Child: 5 There are: 38 seats, booking one! My PID is : 1873
the main just read seats_left: 37
I am Child: 0 There are: 37 seats, booking one! My PID is : 1868
the main just read seats_left: 36
I am Child: 4 There are: 36 seats, booking one! My PID is : 1872
the main just read seats_left: 35
I am Child: 1 There are: 35 seats, booking one! My PID is : 1869
the main just read seats_left: 34
I am Child: 3 There are: 34 seats, booking one! My PID is : 1871
the main just read seats_left: 33
I am Child: 2 There are: 33 seats, booking one! My PID is : 1870
the main just read seats_left: 32
I am Child: 0 There are: 32 seats, booking one! My PID is : 1868
the main just read seats_left: 31
I am Child: 4 There are: 31 seats, booking one! My PID is : 1872
the main just read seats_left: 30
I am Child: 1 There are: 30 seats, booking one! My PID is : 1869
the main just read seats_left: 29
```

I am Child: 3 There are: 29 seats, booking one! My PID is : 1871

the main just read seats_left: 28

I am Child: 5 There are: 28 seats, booking one! My PID is : 1873

the main just read seats_left: 27

I am Child: 0 There are: 27 seats, booking one! My PID is : 1868

the main just read seats_left: 26

I am Child: 2 There are: 26 seats, booking one! My PID is : 1870

the main just read seats_left: 25

I am Child: 4 There are: 25 seats, booking one! My PID is : 1872

the main just read seats_left: 24

I am Child: 1 There are: 24 seats, booking one! My PID is : 1869

the main just read seats_left: 23

I am Child: 3 There are: 23 seats, booking one! My PID is : 1871

the main just read seats_left: 22

I am Child: 5 There are: 22 seats, booking one! My PID is : 1873

the main just read seats_left: 21

I am Child: 2 There are: 21 seats, booking one! My PID is : 1870

the main just read seats_left: 20

I am Child: 4 There are: 20 seats, booking one! My PID is : 1872

the main just read seats_left: 19

I am Child: 0 There are: 19 seats, booking one! My PID is : 1868

the main just read seats_left: 18

I am Child: 3 There are: 18 seats, booking one! My PID is : 1871

the main just read seats_left: 17

I am Child: 1 There are: 17 seats, booking one! My PID is : 1869

the main just read seats_left: 16

I am Child: 2 There are: 16 seats, booking one! My PID is : 1870

the main just read seats_left: 15

I am Child: 4 There are: 15 seats, booking one! My PID is : 1872

```
the main just read seats_left: 14
I am Child: 5 There are: 14 seats, booking one! My PID is : 1873
the main just read seats_left: 13
I am Child: 0 There are: 13 seats, booking one! My PID is : 1868
the main just read seats_left: 12
I am Child: 3 There are: 12 seats, booking one! My PID is : 1871
the main just read seats_left: 11
I am Child: 1 There are: 11 seats, booking one! My PID is : 1869
the main just read seats_left: 10
I am Child: 2 There are: 10 seats, booking one! My PID is : 1870
the main just read seats_left: 9
I am Child: 5 There are: 9 seats, booking one! My PID is : 1873
the main just read seats_left: 8
I am Child: 4 There are: 8 seats, booking one! My PID is : 1872
the main just read seats_left: 7
I am Child: 0 There are: 7 seats, booking one! My PID is : 1868
the main just read seats_left: 6
I am Child: 3 There are: 6 seats, booking one! My PID is : 1871
the main just read seats_left: 5
I am Child: 1 There are: 5 seats, booking one! My PID is : 1869
the main just read seats_left: 4
I am Child: 2 There are: 4 seats, booking one! My PID is : 1870
the main just read seats_left: 3
I am Child: 5 There are: 3 seats, booking one! My PID is : 1873
the main just read seats_left: 2
I am Child: 4 There are: 2 seats, booking one! My PID is : 1872
the main just read seats_left: 1
I am Child: 0 There are: 1 seats, booking one! My PID is : 1868
Did we all Join?  There will be 7 of us if we did.  PID: 1871
```

```
Did we all Join?  There will be 7 of us if we did.  PID: 1869
Did we all Join?  There will be 7 of us if we did.  PID: 1868
Did we all Join?  There will be 7 of us if we did.  PID: 1870
Did we all Join?  There will be 7 of us if we did.  PID: 1872
Did we all Join?  There will be 7 of us if we did.  PID: 1873
Parent exit loop. PID: 1864
Waiting for PID: 1868 to finish
PID: 1868 has shut down
Waiting for PID: 1869 to finish
PID: 1869 has shut down
Waiting for PID: 1870 to finish
PID: 1870 has shut down
Waiting for PID: 1871 to finish
PID: 1871 has shut down
Waiting for PID: 1872 to finish
PID: 1872 has shut down
Waiting for PID: 1873 to finish
PID: 1873 has shut down
Did we all Join?  There will be 7 of us if we did.  PID: 1864
There are no ZOMBIES!
```