

Instituto Politécnico Nacional

Escuela Superior de Cómputo

*Sistemas Operativos*

*“Práctica 5. Comunicación interprocesos (IPC) en Linux y Windows”*

**Grupo:** 2CM8

**Integrantes:**

* Martínez Coronel Brayan Yosafat.
* Monteros Cervantes Miguel Angel.
* Ramírez Olvera Guillermo.
* Sánchez Méndez Edmundo Josue.

**Profesor:** Cortés Galicia Jorge



**Práctica 5. Comunicación interprocesos (IPC) en Linux y Windows**

**Introducción**

La comunicación entre procesos es una función básica de los sistemas operativos que provee un mecanismo que permite a los procesos comunicarse y sincronizarse entre sí, normalmente a través de un sistema de bajo nivel de paso de mensajes que ofrece la red subyacente. Las técnicas de IPC están divididas dentro de métodos para: paso de mensajes, sincronización, memoria compartida y llamadas de procedimientos remotos (RPC).

En un sistema, los procesos pueden ejecutarse independientemente o cooperando entre sí. Los intérpretes de comandos son ejemplos típicos de procesos que no precisan la cooperación de otros para realizar sus funciones. En cambio, los procesos que sí cooperan necesitan comunicarse entre sí para poder completar sus tareas. La comunicación entre procesos puede estar motivada por la competencia o el uso de recursos compartidos o porque varios procesos deban ejecutarse sincronizadamente para completar una tarea.  
Para que puedan realizarse ambos tipos de interacciones, es necesario que el sistema operativo provea de servicios para posibilitar la comunicación entre procesos. El sistema operativo provee mínimamente dos primitivas, "enviar" y "recibir", normalmente llamadas *send* y *receive*. Asimismo, debe implementarse un enlace de comunicación entre los procesos de la comunicación (*pipe*). Este enlace puede ser unidireccional o multidireccional según permita la comunicación en uno o en varios sentidos.

La comunicación puede ser:

- Síncrona: Quien envía permanece bloqueado esperando a que llegue una respuesta del receptor antes de realizar cualquier otro ejercicio.

- Asíncrona: Quien envía continúa con su ejecución inmediatamente después de enviar el mensaje al receptor.

- Persistente: El receptor no tiene que estar operativo al mismo tiempo que se realiza la comunicación, el mensaje se almacena tanto tiempo como sea necesario para poder ser entregado (por ejemplo, un e-mail).

- Momentánea (transient): El mensaje se descarta si el receptor no está operativo al tiempo que se realiza la comunicación. Por lo tanto, no será entregado.

- Directa: Las primitivas “enviar” y “recibir” especifican el nombre del proceso con el que se comunican.

Las operaciones básicas *send* y *receive* se definen de la siguiente manera:

**send**(*P*, *mensaje*): envía un mensaje al proceso *P* (*P* es el proceso destino).  
**receive** *(Q*, *mensaje*): espera la recepción de un mensaje por parte del proceso *Q* (*Q* es el proceso fuente).

Nota: *receive* puede esperar de un proceso cualquiera un mensaje, pero el *send* sí debe especificar a quién va dirigido y cuál es el mensaje.

- Indirecta: Es aquella donde la comunicación está basada en una herramienta o instrumento ya que el emisor y el perceptor están a distancia.

- Simétrica: Todos los procesos pueden enviar o recibir. También llamada bidireccional para el caso de dos procesos.

- Asimétrica: Un proceso puede enviar, los demás procesos sólo reciben. También llamada unidireccional. Suele usarse para hospedar servidores en Internet.

- Uso de búfer automático: El transmisor se bloquea hasta que el receptor recibe el mensaje (capacidad cero).

Hablando un poco sobre las tuberías y memorias compartidas las cuales usamos en estas prácticas:

Esencialmente, las tuberías, ya sean con nombre o anónimas, se utilizan para transmisión de mensajes. Alguien envía una información al destinatario y el destinatario puede recibirla. La memoria compartida es más como publicar datos: alguien coloca datos en la memoria compartida y los lectores (potencialmente muchos) deben usar la sincronización, por ejemplo, a través de semáforos para conocer el hecho de que hay nuevos datos y debe saber cómo leer la región de la memoria para encontrar la información.

Con las tuberías, la sincronización es simple y está integrada en el propio mecanismo de la tubería: sus lecturas y escrituras congelarán y descongelarán la aplicación cuando suceda algo interesante. Con la memoria compartida, es más fácil trabajar de forma asincrónica y buscar nuevos datos solo de vez en cuando, pero a costa de un código mucho más complejo. Además, puede obtener comunicación de muchos a muchos, pero requiere más trabajo nuevamente. Además, debido a lo anterior, la depuración de la comunicación basada en canalizaciones es más fácil que la depuración de la memoria compartida.

La memoria compartida también le brinda más control sobre el almacenamiento en búfer y el uso de recursos; dentro de los límites permitidos por el sistema operativo, es usted quien decide cuánta memoria asignar y cómo usarla. Con las tuberías, el sistema operativo controla las cosas automáticamente, por lo que una vez más pierde algo de flexibilidad, pero se libera de mucho trabajo.

Resumen de los puntos más importantes: uso de tuberías para la comunicación uno a uno, menos codificación y menos dejar que el sistema operativo maneje las cosas, memoria compartida para muchos a muchos, más control manual sobre las cosas, pero a costa de más trabajo y una depuración más difícil.

**1. Competencias.**

El alumno comprende el funcionamiento de las tuberías (pipes) sin nombre y de la memoria compartida como mecanismos de comunicación entre procesos tanto en el sistema operativo Linux como Windows para el desarrollo de aplicaciones concurrentes con soporte de comunicación.

**2. Desarrollo**

**2.1. Sección Linux**

**2.1.1. Información de las llamadas al sistema**

**2.1.1.1. pipe()**



La llamada **pipe()** crea una tubería, es decir, un canal de comunicación unidireccional que puede ser usado para la comunicación interprocesos.

* **pipefd[0]** se refiere al descriptor de archivo del lado de la lectura de la tubería.
* **pipefd[1]** se refiere al descriptor de archivo del lado de la escritura de la tubería.
* En caso de éxito, el valor de retorno es 0. En caso de error, devuelve −1.

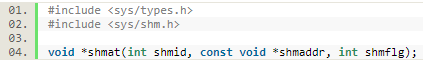
**2.1.1.2. shmget()**



La llamada **shmget()** devuelve el identificador del segmento de memoria compartida asociado con el valor de la llave **key**.

* **key\_t key**: La llave del segmento de memoria compartida.
* **size\_t size**: El tamaño de la memoria compartida, redondeado al múltiplo más cercano de **PAGE\_SIZE**.
* **int shmflg**: En sus últimos 9 bits menos significativos guarda los permisos de la memoria compartida. Ademas, puede tomar los siguientes valores usando máscara de bits:
  + IPC\_CREAT: Crea un nuevo segmento. Si este **flag** no se especifica, la funcion tratara de encontrar el segmento asociado a **key** y revisar si tenemos permisos para acceder al segmento.
  + IPC\_EXCL: Se usa con el **flag** anterior para garantizar que el segmento se cree, es decir, si no se crea ocurrirá un error.
* En caso de éxito, devuelve un identificador válido de la memoria compartida. En caso de error, devuelve −1.

**2.1.1.3. shmat()**

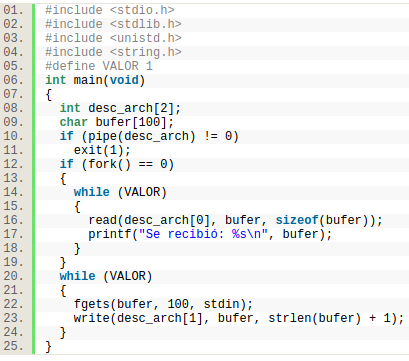


La llamada **shmat()** adjunta el segmento de memoria compartida asociado con el identificador al espacio de direcciones del proceso que realiza la llamada.

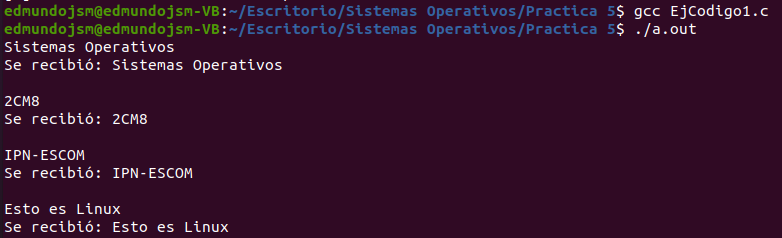
* int **shmid**: El identificador de la memoria compartida.
* const void **\*shmaddr**: La dirección de memoria del proceso actual en donde se adjuntará la memoria compartida. El comportamiento es el siguiente:
  + Si **shmaddr** es un puntero nulo, el segmento se adjunta en la primera dirección disponible seleccionada por el sistema operativo.
  + Si **shmaddr** no es nulo y el flag **SHM\_RND** está activado, el segmento se adjunta en la dirección dada por *shmaddr* − *((uintptr\_t)shmaddr %SHMLBA*.
  + Si **shmaddr** no es nulo y el flag **SHM\_RND** está desactivado, el segmento se adjunta en la dirección dada por *shmaddr*.
  + Si el flag **SHM\_RDONLY** está activado y el proceso actual tiene permisos de lectura, el segmento se adjunta para solo lectura. Si el flag está desactivado y el proceso actual tiene permisos de lectura y escritura, el segmento se adjunta para lectura y escritura.
* En caso de éxito, devuelve el comienzo de la dirección de memoria del segmento que se adjuntó. En caso de error, la memoria no se adjunta y devuelve −1.

**2.1.2. Ejemplo de tuberías en Linux**

Código(EjCodigo1.c)



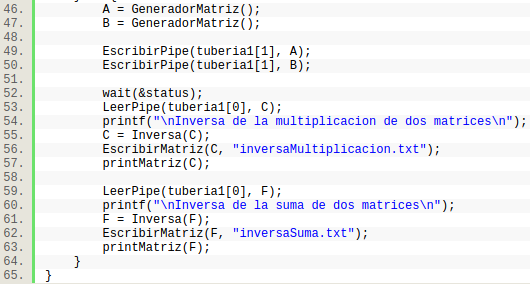
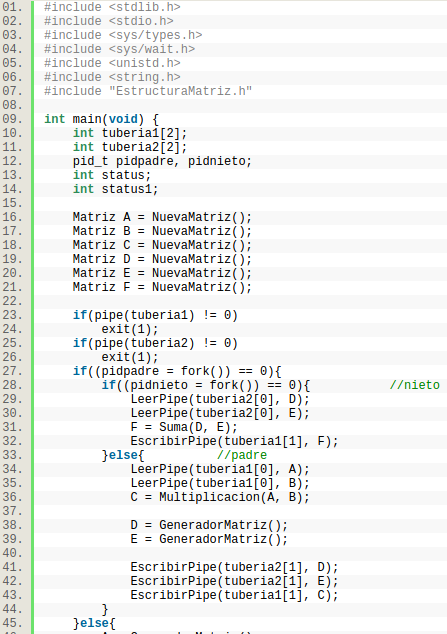
Compilación y ejecución del programa:



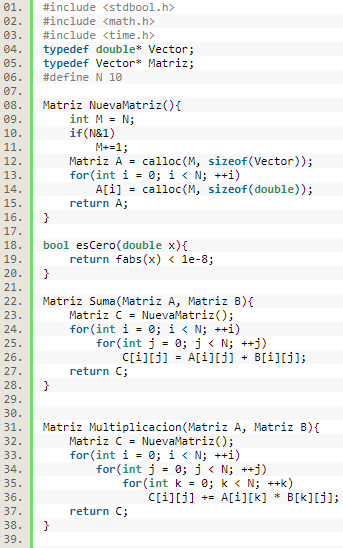
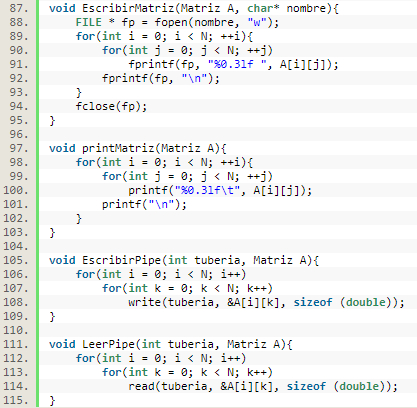
En este código podemos ver el uso de la llamada al sistema pipe() la cual funciona para crear una tubería y compartir una cadena que se escriba en la terminal entre dos procesos los cuales tienen un ciclo que siempre se va a cumplir y la única manera de poder finalizar el programa es pulsando las combinación de teclas Ctrl+Z, ademas, podemos observar cómo recibe una cadena “bufer” el proceso padre con la función fgets() y la comparte por medio de un pipe “desc\_arch[1]” el cual es la entrada y escribe en la tubería con la funcion write() y en su respectiva salida del pipe que es “des\_rch[0]” se lee la cadena “bufer” con la funcion read(). Finalmente hay que mencionar que se añadió una librería al programa original y se cambió la funcion gets por fgets ya que se generaba errores de compilación.

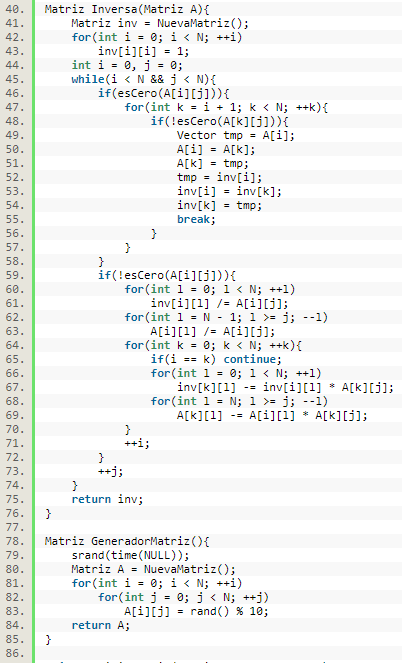
**2.1.3. Programa con tuberías en Linux (Usando Matrices)**

Código(MatricesPipes.c)

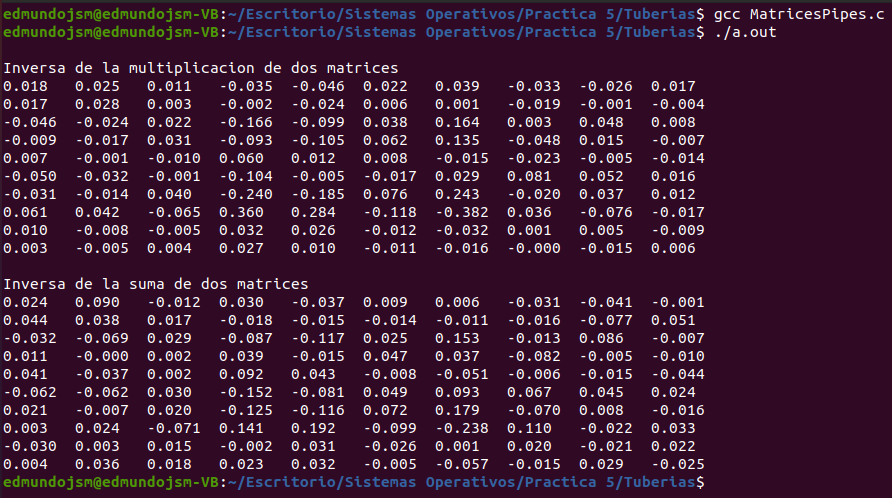
 

Código(EstructuraMatriz.h)

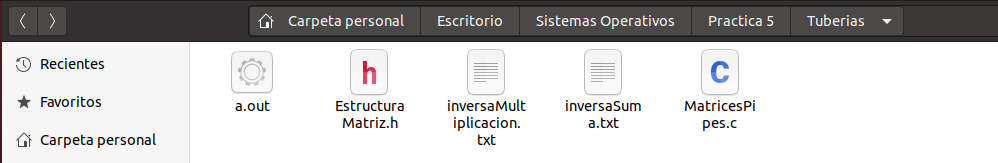




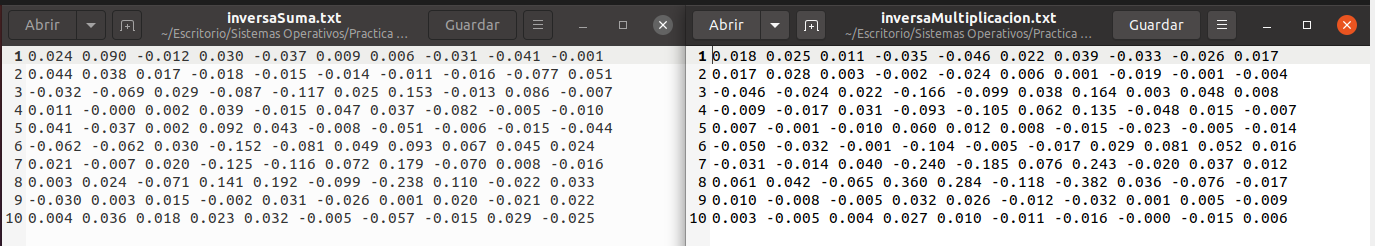
Compilación y ejecución del programa:



Archivos generados en la carpeta raíz.



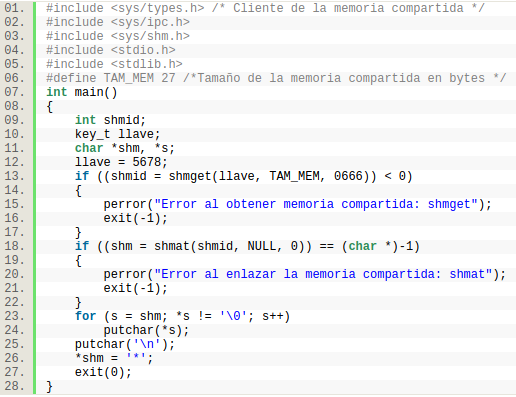
Archivos generados exitosamente con el mismo contenido que el impreso.



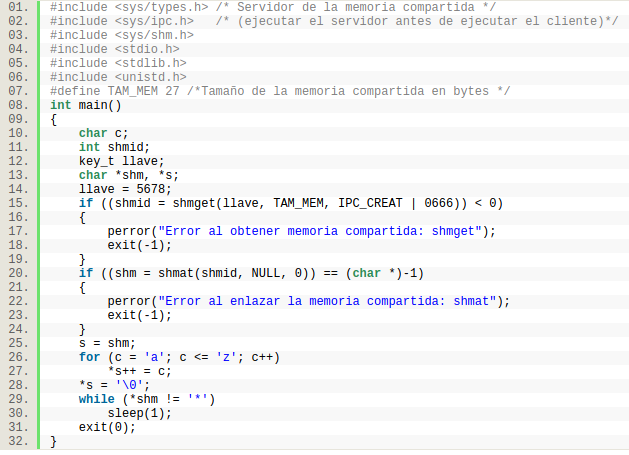
Mediante el uso de las llamadas al sistema pipe(), write() y read() nos ayuda a la manipulación de tuberías para el traspaso de datos entre procesos que en este caso serán datos de matrices que se manipularán. Donde la llamada al sistema pipe() crea la tubería, write() escribe los datos que se desea enviar en la entrada de la tubería y read() recibe los datos de la salida de la tubería. Se observa cómo se envía dos matrices entre tres procesos, el primer proceso (proceso abuelo) le manda dos matrices a su proceso hijo a través de una tubería y obtendrá la multiplicación de ambas matrices que a su vez mandara el resultado a su proceso padre, después este hijo creara un hijo de él y le mandara dos matrices a su hijo a través de una tubería de las cuales obtendrá la suma de las dos matrices recibidas y mandara al primer proceso creado (proceso abuelo) el resultado de la suma. Finalmente, el primer proceso obtendrá la matriz inversa de la matriz resultante de la multiplicación y de la suma. Donde se va a escribir ambas inversas resultantes en un archivo .txt distinto, la matriz inversa de la multiplicación con el nombre “inversaMultiplicacion.txt” y la matriz inversa de la suma con el nombre “inversaSuma.txt”.

**2.1.4. Ejemplo de memoria compartida en Linux**

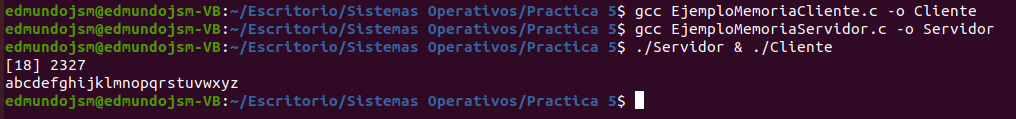
Código(EjemploMemoriaCliente.c)



Código(EjemploMemoriaServidor.c)



Compilación y ejecución del código.

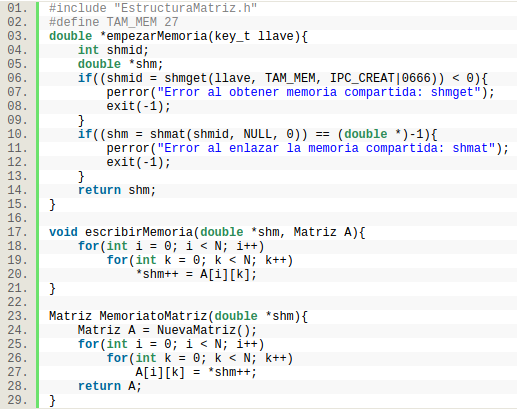


Con los siguientes códigos mostramos la funcion de la memoria compartida con las llamadas al sistema shmget() para obtener la memoria compartida y shmat() para enlazar con la memoria compartida. Finalmente observamos su funcionamiento a través de la impresión del abecedario, el cual lo escribe en la memoria compartida “EjemploMemoriaServidor.c” y lo lee e imprime “EjemploMemoriaCliente.c”.

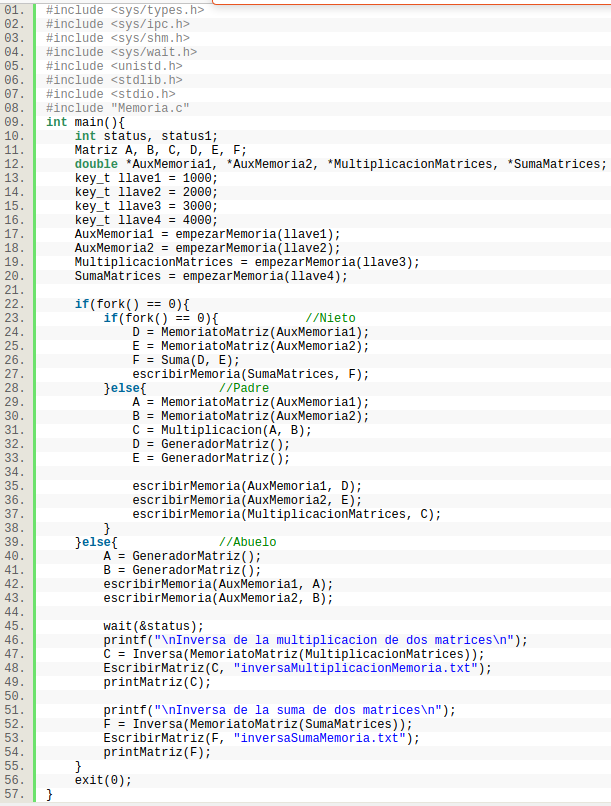
**2.1.5. Programa con memoria compartida en Linux (Usando Matrices)**

Como se vuelve a usar de nueva manera el archivo “EstructuraMatriz.h” sin las funciones creadas para el desarrollo de la parte de tuberías, se decidió no volver a agregar el código, para ahorrar mas espacio en el documento y que no esté tan pesado.

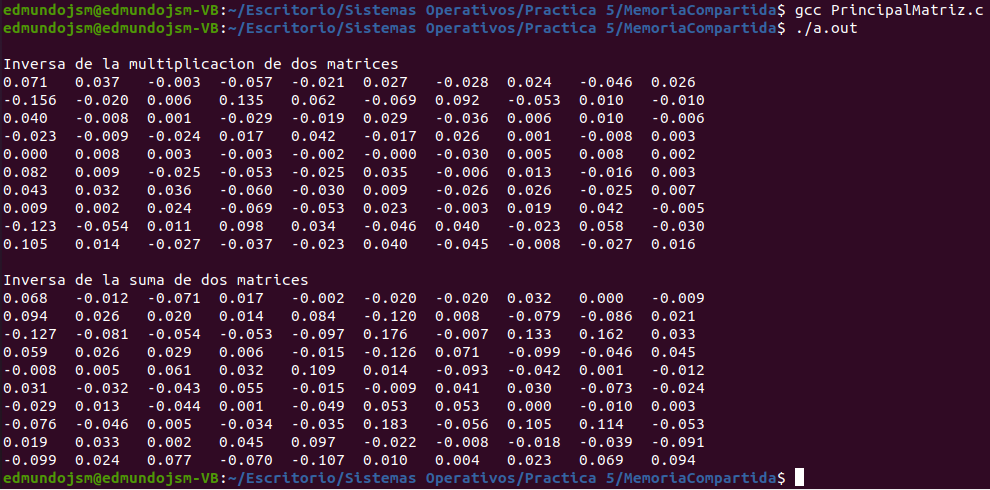
Código(Memoria.c)



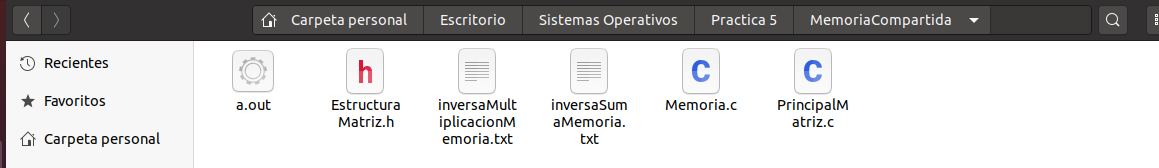
Código(PrincipalMatriz.c)



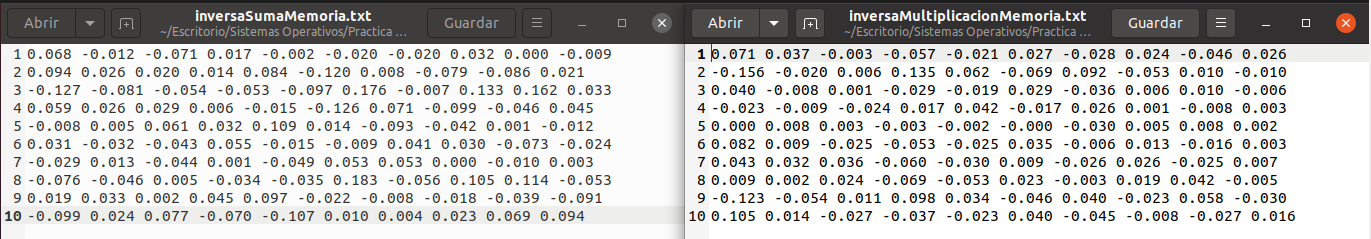
Compilación y ejecución del código.



Archivos generados en la carpeta raíz.



Archivos generados exitosamente con el mismo contenido que el impreso.

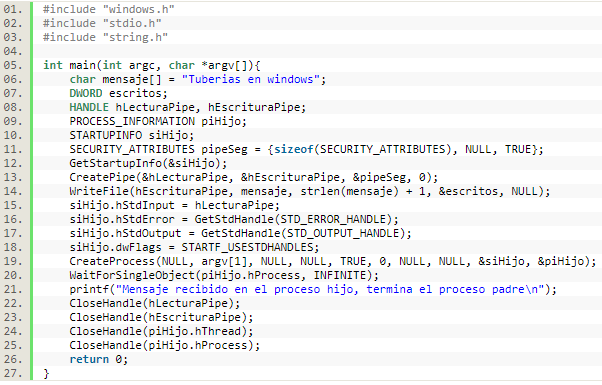


Tiene el mismo objetivo que el programa de “ Programa con tuberías en Linux (Usando Matrices)”, a excepción de que la información se va transfiriendo por memoria compartida los datos de las matrices. La memoria compartida se crea a través de las llamadas al sistema shmget() para obtener el identificador del segmento de memoria compartida y shmat() adjuntar a la memoria compartida con el identificador generado por shmget() en donde esta se conectará entre procesos por medio de la llamada al sistema fork().

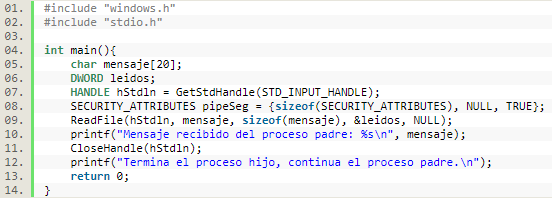
**2.2. Sección Windows**

**2.2.1. Ejemplo de tuberías en Windows**

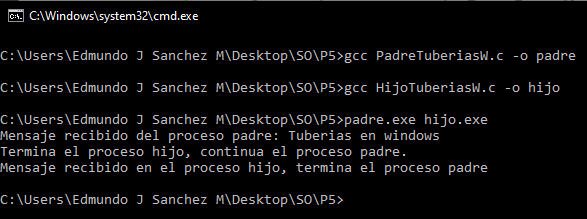
Código(PadreTuberiasW.c)



Código(HijoTuberiasW.c)



Compilación y ejecución del código.



En el programa PadreTuberiasW.c se crea un proceso que ejecuta el programa HijoTuberiasW.c. Para que se puedan comunicar es necesario la creación de la tubería mediante funcion: CreatePipe(), que mediante una serie de HANDLE como argumentos se pueden comunicar, aunque sean programas distintos.

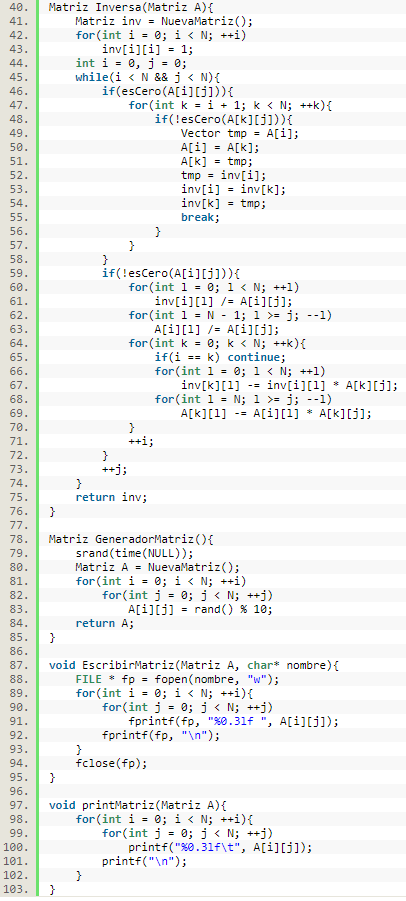
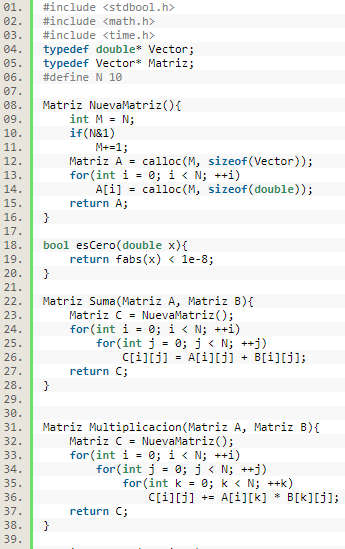
**2.2.2. Programa con tuberías en Windows (Usando Matrices)**

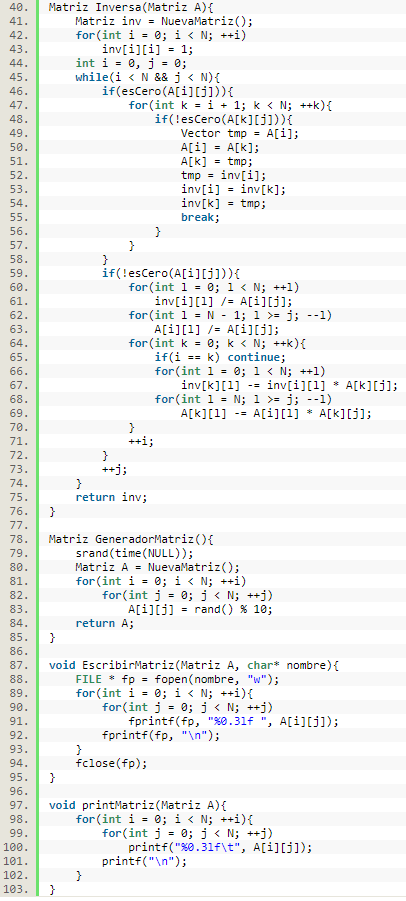
Código(MatricesPipes.c)



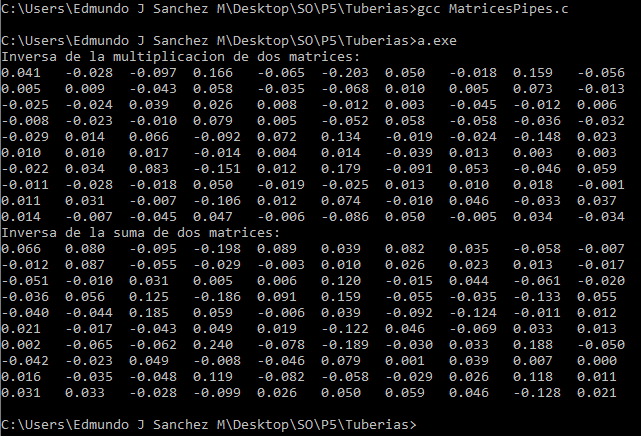


Código(EstructuraMatriz.h)

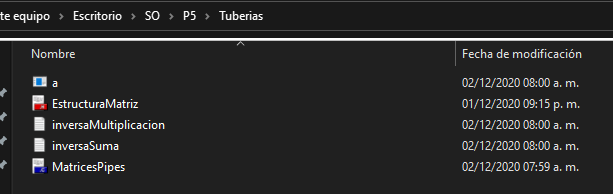




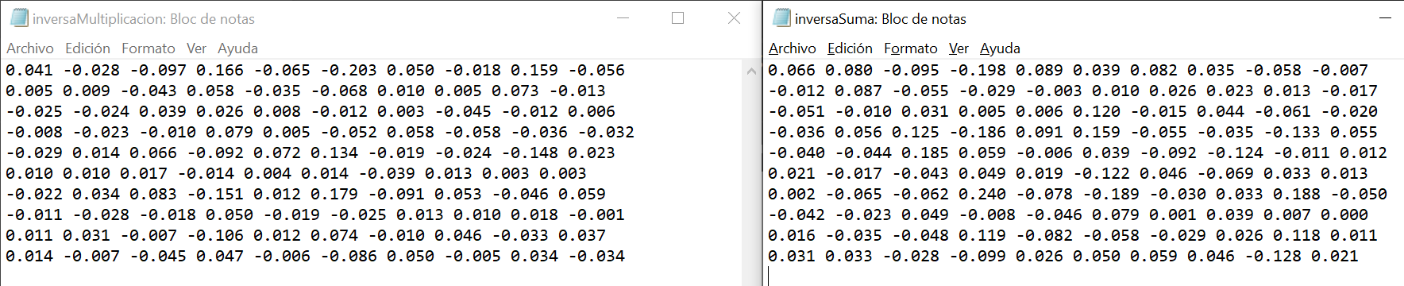
Compilación y ejecución del programa:



Archivos generados en la carpeta raíz.



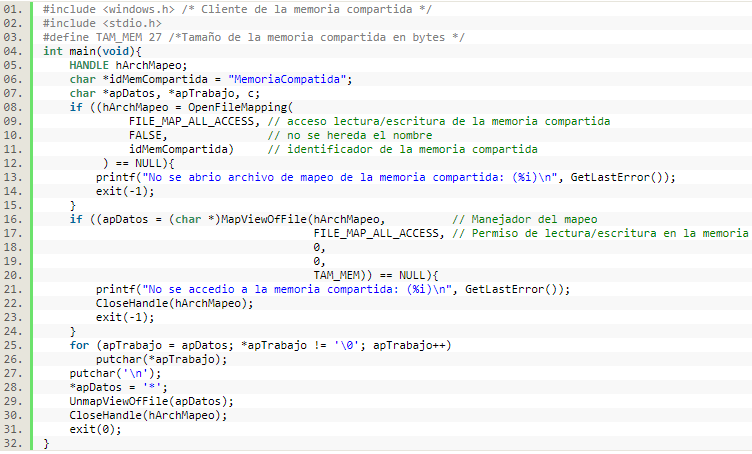
Archivos generados exitosamente con el mismo contenido que el impreso.



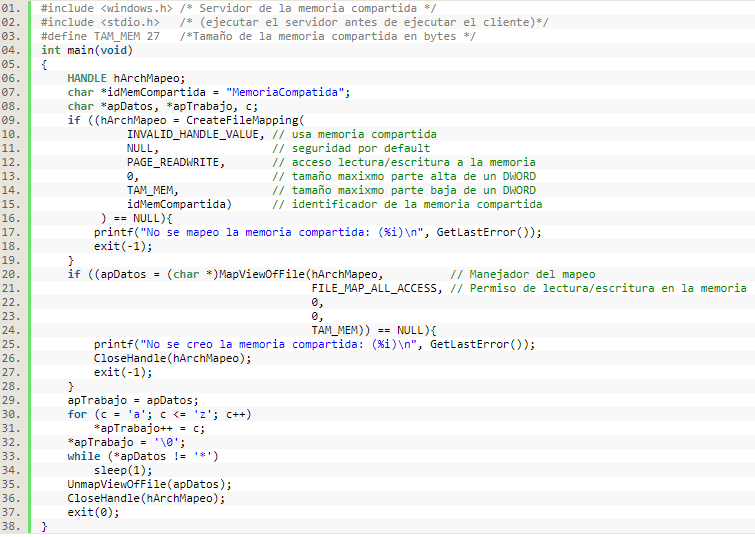
La aplicación es de forma similar a la del punto anterior, sin embargo, debido a que en Windows los procesos creados no comparten las variables, tuvimos que enviar como tal el HANDLE de las tuberías del lado de la escritura por la misma tubería para poder escribir en ella desde el proceso hijo. El programa recibirá un argumento por consola que indicara si es el proceso abuelo, padre o hijo (nivel 0, 1 y 2 respectivamente) esto enviado por el mismo programa.

**2.2.3. Ejemplo de memoria compartida en Windows**

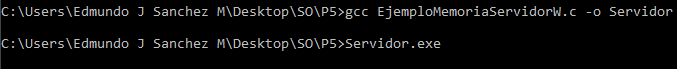
Código(EjemploMemoriaClienteW.c)



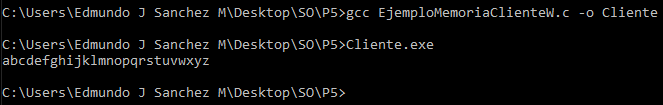
Código(EjemploMemoriaServidorW.c)



Compilación y ejecución del código, hay que mencionar que hay que ejecutar en do consolas separadas, una se usara para el servidor y otra para el cliente.



Compilación y ejecución del servidor de nuestro programa.



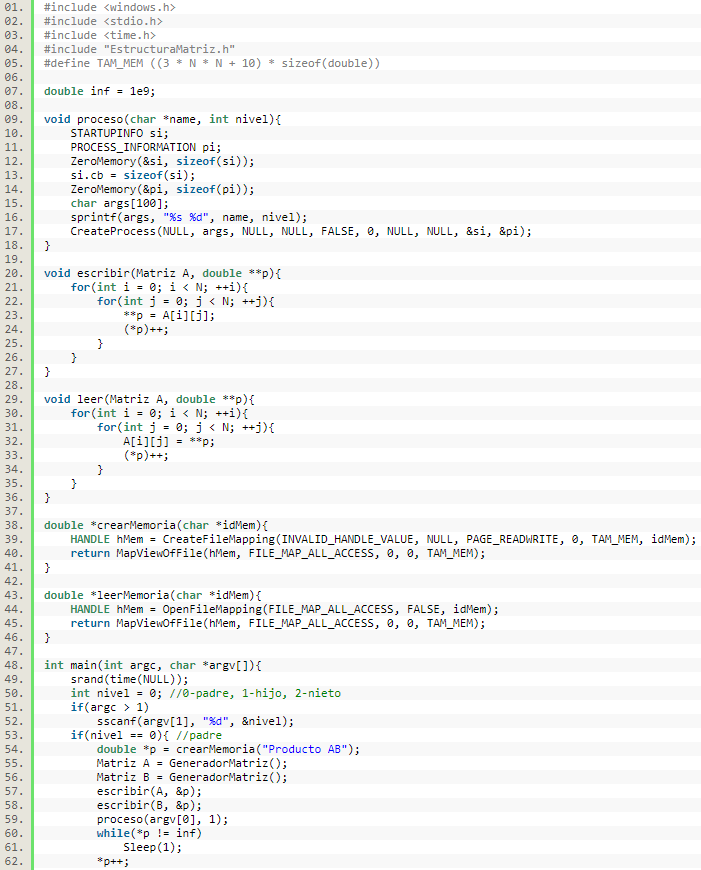
Compilación y ejecución del servidor de nuestro programa.

A diferencia de las tuberías, aquí no es necesario crear procesos para que envíe la información a comunicar. Para que algún programa pueda acceder a la memoria compartida, donde se guarda la información, debe tener el mismo ID (clave) del programa que creo la memoria compartida, mencionar que el programa servidor se detiene cuando el cliente termina de ejecutarse.

**2.2.4. Programa con memoria compartida en Windows (Usando Matrices)**

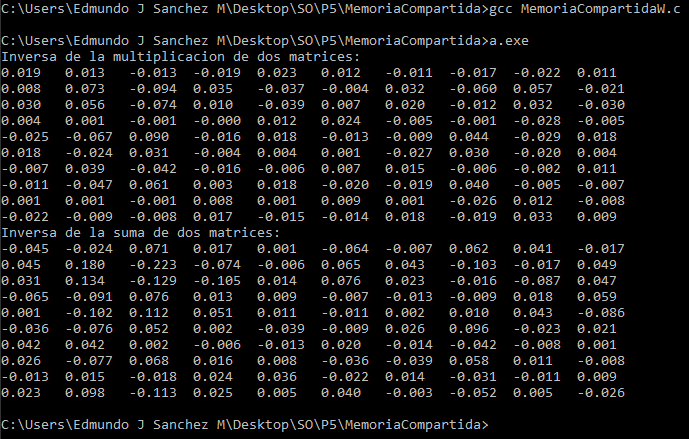
Usando de nueva manera el archivo “EstructuraMatriz.h”, tenemos el siguiente código para el programa.

Código(MemoriaCompartidaW.c)





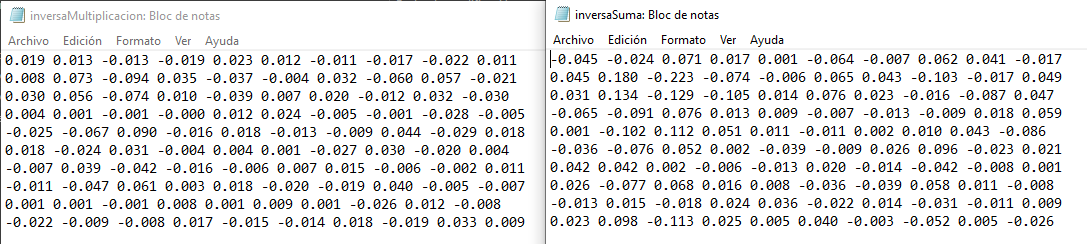
Compilación y ejecución del programa:



Archivos generados en la carpeta raíz.



Archivos generados exitosamente con el mismo contenido que el impreso.



Aquí se complicaron las cosas, porque cuando el proceso padre que crea la memoria compartida finaliza, también lo hacen sus memorias compartidas, por lo que tuvimos que simular una pequeña sincronización.

El programa recibirá un argumento por consola que indicara si es el proceso abuelo, padre o hijo (nivel 0, 1 y 2 respectivamente). Iniciando en el nivel 0 haremos lo siguiente:

1. Crearemos una memoria compartida llamada Producto AB con tamaño para tres matrices. Mandaremos dos matrices aleatorias A y B y crearemos un nuevo proceso con nivel 1. Luego, tendremos un while esperando a que el proceso de nivel 1 ya tenga listo el producto AB, usando un valor especial de infinito ∞ como carácter de control (≈ ).
2. Al comienzo del proceso de nivel 1 accederemos a la memoria compartida Producto AB, leemos A y B, calculamos el producto AB, escribimos el carácter de control ∞ e inmediatamente escribimos la matriz AB.
3. Lo anterior provoca que el proceso de nivel 0 se desbloquee, entonces leemos la matriz AB que nos acaba de enviar el nivel 1 y de nuevo volvemos a esperar por el carácter de control.
4. De vuelta en el nivel 1 creamos otra memoria compartida llamada Suma C+D, creamos dos matrices aleatorias C y D y las mandamos por ahí; creamos el proceso de nivel 2 y nos ponemos en espera por el carácter de control y por el resultado de C + D.
5. En el nivel 2 accedemos a la memoria compartida Suma C+D, leemos C y D, hallamos C + D, escribimos el carácter de control e inmediatamente escribimos la matriz C + D.
6. Con la acción anterior regresamos al nivel 1, el cual se desbloquea y manda el carácter de control por la memoria compartida Producto AB. Aquí termina el proceso de nivel 1.
7. De esa forma, regresamos al nivel 0, el cual se vuelve a desbloquear. Imprimimos el resultado de AB y accedemos a la memoria compartida Suma C+D, avanzamos el puntero dos matrices, un carácter de control (recordemos que estas matrices eran C y D), y ahora si ya leemos la matriz C + D, mandando un carácter de control inmediatamente después.
8. Así, el proceso de nivel 2 se desbloquea y finaliza. 9. De regreso al nivel 0, calculamos (AB) −1 y (C+D) −1 , las guardamos en un archivo y las imprimimos. Aquí termina el proceso de nivel 0.

**3. Análisis de la práctica**

**3.1 Linux**

Usamos las llamadas al sistema para la creación y manipulación, se pueden crear varios al mismo momento. Las funciones de las llamadas al sistema de las tuberías son:

* Crear una tubería o pipe con pipe().
* Para identificar a la tubería es por medio de su nombre.
* Escribir datos en la tubería con write().
* Obtener o leer los datos de una tubería con read()

Si se quiere volver a ocupar los mismos datos de una tubería se tiene que volver a escribir en ella los datos, ya que una vez obtenidos los datos de la tubería con la llamada al sistema read() se borran.

Por otra parte, las llamadas al sistema para la creación y manipulación de memorias compartidas, de las cuales se pueden crear varias en el mismo código. El funcionamiento que se observa de las llamadas al sistema es:

* Obtener el identificador del segmento de memoria compartida con shmget().
* Adjuntar la memoria compartida con el identificador por medio de shmat().
* Se escribe en el segmento de memoria compartida al igual que en un apuntador.
* Se obtienen los datos igual que un apuntador.

Cuando un proceso escribe en una memoria compartida se puede hacer uso de estos datos varias veces, los datos permanecen estáticos.

Además, realizar la aplicación de memoria compartida fue mucho más fácil en Linux que en Windows, porque la creación de procesos por copia exacta nos permite que la memoria compartida permanezca aun cuando finalizan dichos procesos, en cambio en Windows al finalizar los procesos creados se borraban las memorias compartidas que había creado.

**3.2 Windows**

Tanto para creación y manejo de tuberías se deben ocupar llamadas al sistema; también se usan nuevos parámetros, tales como:

* CreatePipe(): Crea la tubería que se necesita ocupar.
* WriteFile(): Permite la escritura en la tubería que se creó.
* ReadFile(): Permite la lectura de la tubería que se mandó al proceso hijo dentro del mismo proceso hijo.
* GetStdHandle(): Retorna un HANDLE hacia un “dispositivo” dependiendo de qué tipo le especifiquemos como argumento.
* SECURITY\_ATTRIBUTES: Es una estructura donde lo más importante que se especifica es si se hereda, de un proceso padre al proceso hijo, los HANDLE (manejadores) con los que se crea la tubería (hLecturaPipe, hEscrituraPipe) para que el proceso hijo pueda manipular la tubería.

**4. Observaciones**

**4.1. Linux**

* Para ingresar y extraer los datos de una tubería son diferentes los accesos para cada una, ya que cuando se crea por medio de la llamada al sistema pipe que recibe un arreglo de tamaño dos y uno representa la entrada y otro la salida.
* La manipulación de una tubería hace uso de las llamadas al sistema pipe () para crearla, write() para escribir en ella y read() para leer sus datos.
* Se puede hacer uso de varios tubería en un código, no tiene un límite de invocación solo que tiene que ser distinto el nombre de la tubería.
* El identificador de la memoria compartida son las llaves key\_t.
* La forma de manipular una memoria compartida y una tubería son similares.
* Para acceder a los datos de la tubería y de la memoria es por medio de su nombre.
* Se debe tener cuidado con la llave key\_t a la hora de crear y acceder a los datos de la memoria compartida ya que puede mandar error o acceder a otra memoria diferente que la deseada.
* Con la llamada al sistema shmget() obtenemos un identificador para el segmento de la memoria compartida y shmat() para enlazar la memoria compartida con el identificador dado por shmget().
* Tanto en tuberías como en memoria compartida se tiene que copiar dato por dato, pero a la hora de obtener el dato estos tienen diferente comportamiento, por lo que hay que hacer un cast.
* La memoria compartida y tubería puede ser creada con la mayoría de los tipos de datos.
* Al igual que la tubería, se puede hacer uso de varias memorias compartidas al mismo tiempo solo que debe tener distinto nombre y key\_t la memoria.
* En Linux, la memoria compartida se destruye al finalizar la ejecución del **último** proceso padre.

**4.2. Windows**

* Las tuberías se deben asociar a un proceso en particular para que sean manipuladas.
* El proceso en donde se ocupa la tubería debe ocupar un HANDLE asociado a la entrada estándar (STD\_INPUT\_HANDLE); el cual es el buffer de entrada de la consola, ahora usado por la tubería.
* Para que haya retroalimentación para comunicar las tuberías, se debe mandar el HANDLE de escritura de la tubería adicional a la información que queremos mandar por la misma tubería, y leerlo del otro lado de forma sincronizada.
* Al finalizar un proceso en Windows, todas las memorias compartidas que creo se destruyen **inmediatamente** de forma intercalada (a diferencia de Linux), por lo que, si requerimos que otro proceso use alguna memoria compartida, el proceso que la creo debe mantenerse en ejecución.

**4.3. Generales**

* La forma de navegar dentro de una tubería es mediante un stream (usamos funciones tales como read o write, y seek para movernos); mientras que en la memoria compartida es a través de un apuntador (usando la aritmética usual de apuntadores).
* En ambos sistemas operativos se cuenta con las llamadas al sistema necesarias para la manipulación de tubería y memorias, nada más que con distinta forma de invocación. La lógica de las aplicaciones no cambio.
* Es factible, tanto en tuberías como en memoria compartida, mandar todos los datos que se necesiten mediante estructuras: arreglos, structs, etc.

**5. Conclusiones**

En ambos sistemas operativos contamos con distintas llamadas al sistema para la creación y manipulación de tuberías y memoria compartida, ambos sistemas crean, escriben y obtienen los datos de ambos canales de comunicación o de transferencia de datos entre procesos. Las llamadas al sistema de Linux y Windows varían en sus parámetros y forma de invocación. Pudimos observar cómo varía el comportamiento entre las tuberías y la memoria compartida, incluso entre los sistemas operativos varía el comportamiento. Si se busca ocupar datos entre dos procesos se recomienda usar tuberías, si en caso de que se busca ocupar los mismos datos entre varios se recomienda utilizar memoria compartida.