

**INSTITUTO POLITÉCNICO NACIONAL.**

ESCUELA SUPERIOR DE CÓMPUTO.

**SISTEMAS OPERATIVOS.**

**PRÁCTICA 4**

Herramientas del IPC (Inter Process Comunication) del

UNIX-Linux.

**Integrantes del equipo:**

* Chavarría Vázquez Luis Enrique.
* Juárez Espinoza Ulises.
* Machorro Vences Ricardo Alberto.
* Pastrana Torres Víctor Norberto.



**Unidad 2**

**2CM6**

Índice de contenido.

[**Glosario de términos.** 1](#_Toc55874728)

[**IPC.** 1](#_Toc55874729)

[**Señal.** 1](#_Toc55874730)

[**Comando.** 1](#_Toc55874731)

[**FIFOS.** 1](#_Toc55874732)

[**Colas.** 1](#_Toc55874733)

[**Contenido (Investigación)** 2](#_Toc55874734)

[**Herramientas del IPC (Inter Process Comunication) de UNIX-Linux** 2](#_Toc55874735)

[**Códigos y ventanas de ejecución** 6](#_Toc55874736)

[Código explicado por partes. 6](#_Toc55874737)

[Código completo. 8](#_Toc55874738)

[Explicación de manera global código 9](#_Toc55874739)

[Ejecución: 10](#_Toc55874740)

[Código explicado por partes. 11](#_Toc55874741)

[Código completo. 11](#_Toc55874742)

[Código general del código por partes. 11](#_Toc55874743)

[Ejecución del código compilado. 11](#_Toc55874744)

[Código explicado por partes. 11](#_Toc55874745)

[Código completo. 11](#_Toc55874746)

[Código general del código por partes. 11](#_Toc55874747)

[Ejecución del código compilado. 11](#_Toc55874748)

[Código explicado por partes. 11](#_Toc55874749)

[Código completo. 11](#_Toc55874750)

[Código general del código por partes. 11](#_Toc55874751)

[Ejecución del código compilado. 11](#_Toc55874752)

[Código explicado por partes. 12](#_Toc55874753)

[Código completo. 12](#_Toc55874754)

[Código general del código por partes. 12](#_Toc55874755)

[Ejecución del código compilado. 12](#_Toc55874756)

[**Conclusiones.** 12](#_Toc55874757)

[**Chavarría Vázquez Luis Enrique.** 13](#_Toc55874758)

[**Juárez Espinoza Ulises.** 13](#_Toc55874759)

[**Machorro Vences Ricardo Alberto.** 13](#_Toc55874760)

[**Pastrana Torres Victor Norberto.** 13](#_Toc55874761)

[Bibliografía 13](#_Toc55874762)

Índice de figuras

[Ilustración 1 ejemplo con sockets. 3](#_Toc55874763)

[Ilustración 2 primera parte del código y definiciones. 6](#_Toc55874764)

[Ilustración 3 explicación de esNumerico 6](#_Toc55874765)

[Ilustración 4 explicación de main 7](#_Toc55874766)

[Ilustración 5 explicación de hilo:1\_function 7](#_Toc55874767)

[Ilustración 6 explicación de hilo\_2\_function 8](#_Toc55874768)

[Ilustración 7 Ejecución estándar del Programa41.c 10](#_Toc55874769)

Índice de tablas

**No se encuentran elementos de tabla de ilustraciones.**

# **Glosario de términos.**

## **IPC.**

Comprende una serie de mecanismos diferenciados que permiten, desde un proceso enviar un dato a otro, y de esta forma, comunicarse

## **Señal.**

Evento que debe ser procesado y que puede interrumpir el flujo normal de un programa.

## **Comando.**

Es la instrucción que se da a un sistema operativo de un ordenador para que ejecute cualquier tarea.

## **FIFOS.**

Es un concepto utilizado en estructuras de datos, contabilidad de costes y teoría de colas. Guarda analogía con las personas que esperan en una cola y van siendo atendidas en el orden en que llegaron, es decir, que "la primera persona que entra es la primera persona que sale".

## **Colas.**

Es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pop por el otro.

# **Contenido (Investigación)**

## **Herramientas del IPC (Inter Process Comunication) de UNIX-Linux**

Como se puede ver en el glosario de términos de esta práctica los IPC son una forma de comunicación entre procesos, que en el caos de los sistemas Unix/Linux este los utiliza todo el tiempo y que por ello incluir en programas fuentes en C es muy sencillo. Las IPC proveen además de 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. Esta necesidad de comunicación nace de que los procesos en UNIX no comparten memoria, ni siquiera los padres con sus hijos. Por tanto, se tiene que establecer algún mecanismo en caso de que se quiera comunicar información entre procesos concurrentes.

Algunos mecanismos son simples, como el envío de una señal de un proceso a otro, que el proceso receptor capturará y trabajará de un modo determinado, dependiendo del código de señal. Esto puede programarse e implementarse en un programa C de múltiples procesos, y permitirle a estos procesos enviarse señales que bifurquen sus ejecuciones.

En el sistema operativo, por ejemplo, cuando ejecutamos un proceso y lo cancelamos con la combinación de teclas ctrl+c, o cuando usamos los comandos kill o killall, estamos siempre enviando algún tipo de señal a algún proceso. El intérprete de comandos solo nos facilita algún tipo de facilidad.

Un ejemplo puede ser:

[die@debian ~]$ sleep 10 ^C

Otros mecanismos no menos útiles son los pipes, o tuberías. Este mecanismo permite que 2 o más procesos envíen información a cualquier otro.

El símbolo «|» comúnmente es el que se usa para comunicar la salida de un proceso con la entrada de otro, logrando una salida enriquecida, es un pipe que se genera al vuelo durante la ejecución de los comandos, y permite procesar la salida de un comando mediante otros.

Una variante de las tuberías o pipes son las conocidas como FIFO, estos pipes pueden programarse en C y trabajarse de una manera muy flexible, incluso escribiendo pipes en disco, que se verán como archivos, pero que servirán de punto de contacto entre procesos totalmente independientes, para intercambio de información. Este tipo de pipe en sistema de archivos se lo conoce como FIFO. A continuación presentamos un ejemplo.

**[die@debian ~]$ ls -l /var/run/acpi\_fakekey**

**p-w------- 1 root root 0 Feb 19 10:35 /var/run/acpi\_fakekey**

Otra de las herramientas IPC es el socket. El socket automáticamente trae a la cabeza conexiones de red. Sin embargo, esto no siempre es así. Existen en sistemas nix dos grandes tipos de sockets. Los sockets UNIX son similares a los FIFOS, archivos en disco que permiten la comunicación entre procesos locales, por ejemplo:

**[die@debian ~]$ ls -l /var/run/cups/cups.sock**

**srwxrwxrwx 1 root root 0 Feb 19 10:35 /var/run/cups/cups.sock**

El otro tipo de socket es el socket INET, que efectivamente, permite conectar procesos en red, o localmente, utilizando redes TCP/IP. Estos sockets no tienen una representación en el sistema de archivos, pero sí pueden analizarse y ver las conexiones que nuestros procesos están efectuando contra otros procesos. En la imagen de abajo se puede ver un ejemplo.

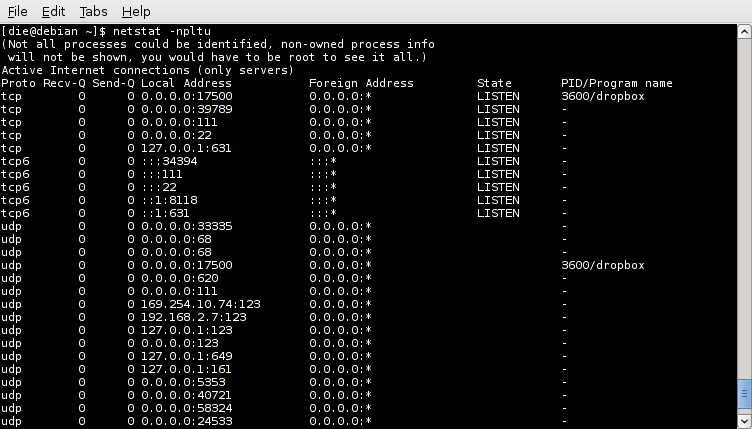


Ilustración 1 ejemplo con sockets.

Estos dos tipos de sockets, de más está decir, también se pueden programar y utilizar su gran potencia para crear aplicaciones más complejas y eficientes.

Uno de los mecanismos utilizables desde código más importantes son los semáforos, que como su nombre lo dice, permiten o deniegan el acceso a ciertos recursos, a un proceso. Un semáforo es un mecanismo de comunicación con el cual no se mueven datos, puesto que solo se puede consultar y modificar su valor al tener un carácter puramente informativo.

Dijkstra define un semáforo como una variable entera positiva o nula sobre la que sólo se pueden realizar dos operaciones: wait (también denominada P) y signal (también denominada V). La operación wait decrementa el valor del semáforo siempre que éste tenga un valor mayor que 0; por lo tanto esta operación se utiliza para adquirir el semáforo o para bloquearlo en el caso de que valga 0. La operación signal incrementa el valor del semáforo y por tanto se utiliza para liberarlo o inicializarlo.

Ambas operaciones deben ser atómicas para que funcionen correctamente; es decir que una operación wait no puede ser interrumpida por otra operación wait o signal sobre el mismo semáforo, y lo mismo ocurre para la operación signal. Este hecho garantiza que cuando varios procesos compitan por la adquisición de un semáforo, sólo uno de ellos va a poder realizar la operación.

Además, se ha de indicar que este mecanismo memoriza las peticiones de operaciones wait no satisfechas y despierta por tanto a los procesos en espera.

El mecanismo IPC de semáforos implementado en UNIX es una generalización más compleja del concepto descrito por Dijkstra, ya que va a permitir manejar un conjunto de semáforos mediante el uso de un identificador asociado y realizar operaciones wait y signal que actualizan de forma atómica todos los semáforos asociados bajo un mismo identificador. Esta complejidad en la utilización de los semáforos, se justifica mediante la imposibilidad de resolver una cierta categoría de problemas con los semáforos manipulados individualmente, por medio únicamente de las operaciones wait y signal.

Otro tipo de mecanismo IPC es la cola de mensajes. En pocas palabras, el primer mensaje que se introduce en la cola es el primer mensaje que se extrae de la misma. Esto se dice que es una comunicación síncrona dado que los mensajes se leen en el mismo orden en que se enviaron, en contraposición a asíncrono, donde el orden de recepción puede ser diferente del orden de envio. Hay cuatro llamadas al sistema asociadas con colas de mensajes :

* msgget() sirve para crear una cola de mensajes y/o obtener un identificador (id) de cola de mensajes a partir de una clave. La clave es un número único que sirve para identificar la cola de mensajes. Cada proceso que desee comunicarse con la cola de mensajes debe conocer su clave. El id es un número asignado por el sistema y obtenido mediante la llamada msgget() y la clave. El id es un parámetro para los otros comandos que actúan sobre la cola de mensajes.
* msgctl() sirve para realizar operaciones de control sobre la cola, incluyendo su eliminación.
* msgsnd() sirve para colocar un mensaje en la cola.
* msgrcv() sirve para extraer un mensaje de la cola.

Las colas de mensajes son relativamente fáciles de usar. El sistema operativo gestiona los detalles internos de la comunicación. Cuando se envía un mensaje a la cola, se alerta a cualquier proceso que esté esperando obtener un mensaje de la misma. El bloqueo de las colas de mensajes es innecesario dado que el sistema operativo verifica la integridad de la cola y no permitirá que dos procesos accedan a la cola de una forma destructiva.

Las colas de mensajes tienen dos inconvenientes importantes. Son mecanismos lentos para transferir gran cantidad de datos, y hay una fuerte limitación sobre el tamaño de los paquetes de datos que pueden transferirse. Por tanto, las colas de mensajes se usan principalmente para pequeñas transferencias de datos, con un ancho de banda limitado. Las colas de mensajes son un mecanismo excelente para pasar información de control de unos procesos a otros.

Por último, otro de los mecanismos que podemos analizar, son los segmentos de memoria compartida. Esta es una región de memoria que puede ser accedida por múltiples procesos. Por ejemplo, si declaramos un vector de 1000 bytes en un programa, sólo puede acceder a él ese programa. Si declaramos un segmento de memoria compartida de 1000 bytes, muchos procesos pueden realizar operaciones de lectura y escritura sobre esa memoria compartida.

La ventaja principal de la memoria compartida es que un programa la ve exactamente de la misma forma que si fuera memoria normal. Además, las operaciones de lectura y escritura son muy rápidas. Su utilización es relativamente simple. De la misma forma que las colas de mensajes, cada segmento de memoria compartida tiene asociado una clave. Esta identifica de forma unívoca el segmento de memoria compartida, y cualquier proceso que desee acceder a él necesita conocer la clave.

* La llamada shmget() se usa para obtener un id para una clave asociada. Este id es similar al id de cola de mensajes y se usa como parámetro en otras llamadas al sistema relacionadas con memoria compartida. La llamada shmget() también se usa para crear segmentos de memoria compartida.
* shmctl() se usa para realizar operaciones de control sobre la memoria compartida, entre ellas, la de eliminar segmentos de memoria compartida del sistema.
* shmat() devuelve un puntero que referencia al segmento de memoria compartida. Este puntero se emplea para acceder al segmento de memoria compartida para realizar tanto operaciones de lectura como escritural.
* shmdt() se emplea para desconectar del segmento de memoria compartida.

En general, la memoria compartida se emplea cuando se necesita transferir gran cantidad de datos en un corto período de tiempo.

# **Códigos y ventanas de ejecución**

**Programa41.c**

Elabore un programa que tenga una variable con un valor inicial de cero. Posteriormente se deben crear dos hilos independientes, uno de ellos incrementa permanentemente la variable en uno y el otro la disminuya en uno. Después de n segundos el proceso debe imprimir el valor final de la variable y terminar. El número de segundos que el proceso padre espera, se debe pasar en la línea de comandos. Sincronice los hilos mediante el uso de semáforos.

### Código explicado por partes.

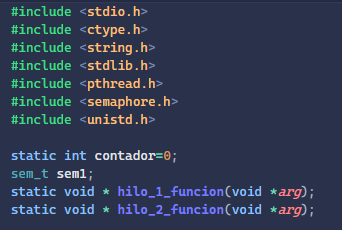


Ilustración 2 primera parte del código y definiciones.

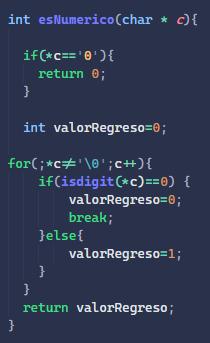


Ilustración 3 explicación de esNumerico



Ilustración 4 explicación de main

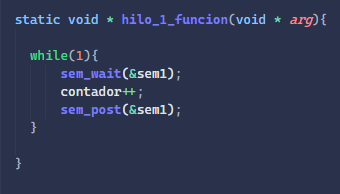


Ilustración 5 explicación de hilo:1\_function

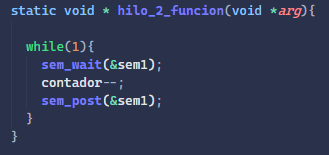


Ilustración 6 explicación de hilo\_2\_function

### Código completo.

**#include <stdio.h>**

**#include <ctype.h>**

**#include <string.h>**

**#include <stdlib.h>**

**#include <pthread.h>**

**#include <semaphore.h>**

**#include <unistd.h>**

**static int contador=0;**

**sem\_t sem1;**

**static void \* hilo\_1\_funcion(void \*arg);**

**static void \* hilo\_2\_funcion(void \*arg);**

**int esNumerico(char \* c){**

**if(\*c=='0'){**

**return 0;**

**}**

**int valorRegreso=0;**

**for(;\*c!='\0';c++){**

**if(isdigit(\*c)==0) {**

**valorRegreso=0;**

**break;**

**}else{**

**valorRegreso=1;**

**}**

**}**

**return valorRegreso;**

**}**

**int main(int argc, char \*argv []){**

**long numeroSeg;**

**char \*p;**

**pthread\_t hilo\_1;**

**pthread\_t hilo\_2;**

**if(argc!=2){**

**printf("Numero de parametros incorrectos \n");**

**}else{**

**if(esNumerico(argv[1])==1){**

**numeroSeg=strtol(argv[1],&p,10);**

**sem\_init(&sem1,0,1);**

**pthread\_create(&hilo\_1,NULL,\*hilo\_1\_funcion,NULL);**

**pthread\_create(&hilo\_2,NULL,\*hilo\_2\_funcion,NULL);**

**sleep(numeroSeg);**

**printf("Contador :%d \n",contador);**

**}else{**

**printf("Escriba un valor numerico mayor a cero \n");**

**}**

**}**

**return 0;**

**}**

**static void \* hilo\_1\_funcion(void \* arg){**

**while(1){**

**sem\_wait(&sem1);**

**contador++;**

**sem\_post(&sem1);**

**}**

**}**

**static void \* hilo\_2\_funcion(void \*arg){**

**while(1){**

**sem\_wait(&sem1);**

**contador--;**

**sem\_post(&sem1);**

**}**

**}**

### Explicación de manera global código

En primero se declara un semáforo y contador global para usar en el resto del código. Dentro del Main primero se checa por medio de un if aplicado a la variable argc si es que el número de parámetros de entrada al llamar al código en el CLI es exactamente 2, que en caso de no cumplirse manda un mensaje diciendo que el número de parámetros es incorrecto. En caso de que los se cumpla el número de parámetros y se supere el primer if , se checa si el segundo argumento proveniente del CLI es un número entero. Para esto se usó una función propia llamada esNumerico que recibe un puntero del tipo de dato char para así poder ver si el primer elemento de este array o String es cero y si es así regresar un cero. En caso de que no se recorre todo el String buscando que todos sus valores conformados sean dígitos del 0 al 9. Si se cumple la condición con la función esNumerico se toma el segundo parámetro de argumentos del programa y por medio de la función strtol se pasa aun valor long int para el numero de segundos que se va esperar a los hilos. Después se inicializa el semáforo para controlar los hilos, para luego crear esos dos hilos con las funciones hilo\_1\_funcion y hilo\_2\_funcion. El primero hilo toma la función hilo\_1\_funcion que incrementa indefinidamente el contador declarado al principio, siendo estos incrementos controlados por el semáforo declarado en el paso anterior. Para el segundo se toma la función hilo\_2\_funcion que es igual que la función hilo\_1\_funcion solo que esta decrementa el contador. Luego el programa principal espera o se duerme por los segundos ingresados y termina imprimiendo del valor del contador.

### Ejecución:

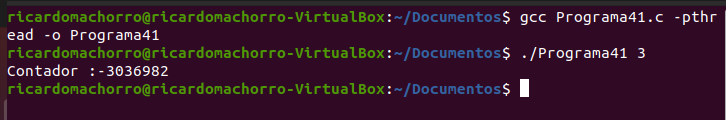


Ilustración 7 Ejecución estándar del Programa41.c

**Programa42.c**

Realizar un programa que utilice memoria compartida donde un proceso padre crea un arreglo con tipos de dato float de 10 posiciones y lo comparte con un proceso hijo. El proceso hijo genera10 números aleatorios de tipo float y los guarda en el arreglo compartido. Al final el proceso padre muestra los números que grabó el proceso hijo en el arreglo.

### Código explicado por partes.

### Código completo.

### Código general del código por partes.

### Ejecución del código compilado.

**Programa43.c**

Crear una comunicación bidireccional (chat) entre dos procesos que no tengan ancestros en común por medio de colas de mensajes.

### Código explicado por partes.

### Código completo.

### Código general del código por partes.

### Ejecución del código compilado.

**Programa44.c**

Programe el Algoritmo de Dekker en dos procesos no emparentados utilizando semáforos y memoria compartida ejecutándolos en terminales diferentes.

### Código explicado por partes.

### Código completo.

### Código general del código por partes.

### Ejecución del código compilado.

**Programa45.c**

Programe el problema de los filósofos utilizando semáforos y memoria compartida.

### Código explicado por partes.

### Código completo.

### Código general del código por partes.

### Ejecución del código compilado.

# **Conclusiones.**

## **Chavarría Vázquez Luis Enrique.**

Primero que nada cabe destacar, que en esta práctica se pudo entender de manera precisa realmente cómo funcionan los diversos mecanismos para la comunicación de los procesos con los cuales podemos jugar dentro de nuestro sistema operativo, siendo la parte esencial el trabajo influjo de información entre los mismos ya sea para proyectos pequeños o proyectos más ambiciosos. A decir verdad en un principio parecía bastante complicado, siendo que para mí era un tema bastante novedoso y desde mi perspectiva desafiante al ser un descubrimiento en mi aprendizaje como aspirante genere sistemas computacionales. La parte que más disfruta es que con esta práctica tuve la oportunidad de entender cómo funcionan los hilos, trabajar directamente en implementación también de un proyecto con el uso de memoria compartida y hasta cierto punto trabajar de cerca con los semáforos, pero con la finalidad de conseguir una gestión eficiente en los distintos problemas.

La parte del desarrollo del chat, una de las que más me enseño a ver la aplicación práctica que este tipo de cosas tienen en el mundo real y que a decir verdad es bastante útil ya que nosotros en nuestro proyecto final implementaremos algo muy similar, por lo que se agradece bastante el hecho de que podamos poner en una escala mucho menor en práctica la comunicación entre en dos procesos de manera bidireccional como lo hemos visto, dándonos la posibilidad de que pensemos en nuevas formasde implementar lo una mayor escala y desde luego a un nivel en el cuerpo a satisfacer necesidades del mercado.

Para ser honesto, la información de la comunicación entre procesos y las múltiples herramientas que existen es bastante amplia, vena muchas posibilidades y por supuesto facilidades; pero fue precisamente esto lo que más me fascina en términos de que la flexibilidad que tienen la comunicación entre los procesos en nuestro sistema operativo es bastante grande y el mundo de posibilidades es realmente tremendo.

Ya finalmente se nos remontamos a problemas más clásicos, como lo es el problema de los filósofos pues también ser interesante poder trasladar dichos problemas hipotéticos a situaciones reales en las cuales tuviésemos que involucrar ciertos procesos similares a lo que vimos en problemas como estos que son totalmente hipotéticos pero que trasladandolos a diferentes contextos sin duda se puede lograr bastantes cosas y muy interesantes, como producto final puedo aseverar que he aprendido cómo hacer distintas interacciones entre los procesos y lo mejor de todo es que el tráfico de información o de datos entre dichos procesos, la comunicación misma se llevó a cabo y se observó directamente en la aplicación de problemas prácticos, lo cual será de ser bastante.

## **Juárez Espinoza Ulises.**

## **Machorro Vences Ricardo Alberto.**

En esta práctica aprendí como hay una gran variedad de medios de comunicación entre procesos que permiten dar un control al sistema de una manera que estos estén sincronizados con el fin de así poder coordinar sus acciones sin molestarse. Dejándome ver que la creación de estos medios como lo son hilos, semáforos y memoria compartida es relativamente sencilla ya que solamente se tiene crear un archivo y una llave para poder administrarlos de manera correcta, dejando como cosa sencilla la implementación de estos. Pero algo que si me complico fue un poco ver como se solucionan algunos problemas con memoria compartida, hilos y semáforos por el hecho de que no estoy realmente acostumbrado a que dentro de un mismo código se hagan varias acciones en paralelo por lo que me confundí un poco de estos además de que algunos de los problemas planteados necesitaban no solo ver la lógica para solucionarlo sino también algo de imaginación de como se pueden solucionar.

Otra complicación que tuve con esta práctica fue entender verdaderamente como funcionan los semáforos porque se me complicaba el uso de una variable para controlar el paso de otros procesos a una sección critica porque en las lecturas que consultaba no las entendía pero viendo su funcionamiento de manera práctica entendí mejor que son y como funcionan . Además, la investigación requerida de esta práctica me ayudo a ver de manera más profunda como es que los procesos se relacionan entre si y como las diferentes herramientas IPC se utilizan en diferentes tareas tanto sencillas como complicadas.

En resumen, esta práctica me sirvió para ver en primera mano que tipo de problemas requieren de hacer comunicaciones entre procesos, la facilidad de las herramientas que permiten solucionar este tipo de problemas y ver que es más cuestión de abstraer estos problemas de manera correcta.

## **Pastrana Torres Victor Norberto.**

El ánodo un

# Bibliografía

**No hay ninguna fuente en el documento actual.**

**Antonio Ovando Sistemas Operativos / Diseño e Implementación - Apuntes**