



INSTITUTO POLITÉCNICO NACIONAL.
ESCUELA SUPERIOR DE CÓMPUTO.



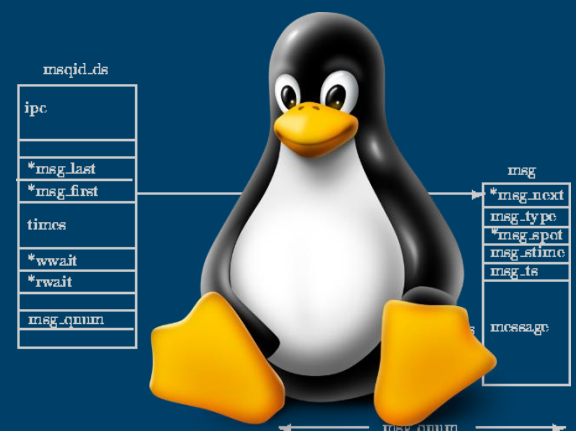
SISTEMAS OPERATIVOS.

PRÁCTICA 4

Herramientas del IPC (Inter Process Communication) del
UNIX-Linux.

Integrantes del equipo:

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



Índice de contenido.

Glosario de términos.	1
IPC.	1
Señal.	1
Comando.	1
FIFOS.	1
Colas.	1
Contenido (Investigación)	2
Herramientas del IPC (Inter Process Communication) de UNIX-Linux	2
Códigos y ventanas de ejecución	6
Programa41.c	6
Código explicado por partes.	6
Código completo (En formato de texto y con explicación general abajo).	8
Explicación de manera general código.	10
Ejecución:	10
Programa42.c	11
Código completo.	11
1) Código completo derivado.c	11
2) Código completo principal.c	12
Explicación general del código.	14
Ejecución del código compilado.	14
Programa43.c	15
Código explicado por partes.	15
3) Vista de los programas en el gestor de archivos.	15
4) Código por partes del proceso_1.c	15
5) Código por partes del proceso_2.c	17
Código completo.	18
1) Código completo proceso_1.c	18
2) Código completo proceso_2.c	19
Explicación general del código completo.	20
1) Explicación general del código de programa proceso_1.c y proceso_2.c	20
Ejecución del código compilado.	20
Programa44.c	21
Código explicado por partes.	21

Código completo.	24
Explicación general del código completo.	26
Ejecución del código compilado.	26
Programa45.c	28
Código explicado por partes.	28
Código completo.	32
Código general del código por partes.	35
Ejecución del código compilado.	35
Conclusiones.	36
Chavarría Vázquez Luis Enrique.	36
Juárez Espinoza Ulises.	37
Machorro Vences Ricardo Alberto.	38
Pastrana Torres Victor Norberto.	39
Bibliografía	40

Índice de figuras

Ilustración 1 ejemplo con sockets.....	3
Ilustración 2 primera parte del código y definiciones.	6
Ilustración 3 explicación de esNumerico.....	6
Ilustración 4 explicación de main.....	7
Ilustración 5 explicación de hilo:1_function	8
Ilustración 6 explicación de hilo_2_function.....	8
Ilustración 7 Ejecución estándar del Programa41.c	10
Ilustración 8 Ejecución del código del programa42.c.....	14
Ilustración 9 Vista de la carpeta en donde estan almacenados los 2 archivos que interactuan en la comunicación de procesos bidireccional.	15
Ilustración 10 Primera parte del código del programa42.c proceso_1.c	15
Ilustración 11 Segunda parte del código del programa42.c proceso_1.c	16
Ilustración 12 Tercera parte del código del programa42.c (proceso_1.c)	16
Ilustración 13 Cuarta parte del código del programa42.c (proceso_1.c).....	16
Ilustración 14 Primera parte del código del programa42.c (proceso_2.c).....	17
Ilustración 15 Segunda parte del código del programa42.c (proceso_2.c).....	17
Ilustración 16 Tercera parte del código del programa42.c (proceso_2.c)	18
Ilustración 17 Código completo de proceso_1.c	18
Ilustración 18 Código completo de proceso_2.c.....	19
Ilustración 19 Ejecución del código del programa43.c.....	20
Ilustración 20. Implementaciones necesarias y creación de la región crítica.	21
Ilustración 21 Creación de la no región crítica para los procesos	22
Ilustración 22 Establecimiento de los identificadores, manejo de la memoria compartida y inicialización de los estados de los procesos.	22
Ilustración 23 Códificación del algoritmo de Dekker para el proceso 1	23
Ilustración 24 Codificación del algoritmo de Dekker para el proceso 2	23
Ilustración 25 Ejecución Programa44.c.....	26
Ilustración 26 Ejecución de programa44.c segunda ilustración.....	27
Ilustración 27 Bibliotecas y macros.....	28
Ilustración 28 pensar	29
Ilustración 29 comer.....	29
Ilustración 30 verifica	30

Ilustración 31 tomar tenedores	30
Ilustración 32 dejar tenedores.....	31
Ilustración 33 filósofos.....	31
Ilustración 34 main filósofos	32
Ilustración 35 Compilación	35
Ilustración 36 ejecución	35
Ilustración 37 ejecución	35

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 Communication) 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. [1]

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 permitirles 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 [2], 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 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 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 en el ejemplo de la ilustración 1.

```
[die@debian ~]$ netstat -npltu
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:17500          0.0.0.0:*               LISTEN      3600/dropbox
tcp        0      0 0.0.0.0:39789         0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:111           0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22            0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:631         0.0.0.0:*               LISTEN      -
tcp6       0      0 :::34394              :::*                   LISTEN      -
tcp6       0      0 :::111                :::*                   LISTEN      -
tcp6       0      0 :::22                 :::*                   LISTEN      -
tcp6       0      0 :::1:8118             :::*                   LISTEN      -
tcp6       0      0 :::1:631              :::*                   LISTEN      -
udp        0      0 0.0.0.0:33335         0.0.0.0:*               -           -
udp        0      0 0.0.0.0:68            0.0.0.0:*               -           -
udp        0      0 0.0.0.0:68            0.0.0.0:*               -           -
udp        0      0 0.0.0.0:17500         0.0.0.0:*               -           3600/dropbox
udp        0      0 0.0.0.0:620           0.0.0.0:*               -           -
udp        0      0 0.0.0.0:111           0.0.0.0:*               -           -
udp        0      0 169.254.10.74:123     0.0.0.0:*               -           -
udp        0      0 192.168.2.7:123       0.0.0.0:*               -           -
udp        0      0 127.0.0.1:123         0.0.0.0:*               -           -
udp        0      0 0.0.0.0:123          0.0.0.0:*               -           -
udp        0      0 127.0.0.1:649         0.0.0.0:*               -           -
udp        0      0 127.0.0.1:161         0.0.0.0:*               -           -
udp        0      0 0.0.0.0:5353          0.0.0.0:*               -           -
udp        0      0 0.0.0.0:40721         0.0.0.0:*               -           -
udp        0      0 0.0.0.0:58324         0.0.0.0:*               -           -
udp        0      0 0.0.0.0:24533         0.0.0.0:*               -           -
```

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 envío. 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. [3]
- msgctl() sirve para realizar operaciones de control sobre la cola, incluyendo su eliminación. [4]
- msgsnd() sirve para colocar un mensaje en la cola. [4]
- msgrcv() sirve para extraer un mensaje de la cola. [4]

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.

En esta primera parte del código lo que estamos haciendo es el agregado de nuestras librerías necesarias para el trabajo con nuestro programa, dentro de las cuales nos interesan las que nos permiten trabajar con precisamente, los semáforos y algunas otras bibliotecas estándar, al igual que implementar algunas otras definiciones en este caso de nuestros dos hilos, esto se aprecia en el código de la ilustración 2.

```
1 #include <stdio.h>
2 #include <ctype.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <pthread.h>
6 #include <semaphore.h>
7 #include <unistd.h>
8
9 static int contador=0;
10 sem_t sem1;
11 static void * hilo_1_funcion(void *arg);
12 static void * hilo_2_funcion(void *arg);
```

Ilustración 2 primera parte del código y definiciones.

Ahora aquí simplemente hacemos algunas cuantas comparaciones para poder entender cuál será el valor que estamos obteniendo y poder validarlo, se observa con claridad lo mencionado en la ilustración 3.

```
1 int esNumerico(char * c){
2
3     if(*c=='0'){
4         return 0;
5     }
6
7     int valorRegreso=0;
8
9     for(;*c!='\0';c++){
10         if(isdigit(*c)==0) {
11             valorRegreso=0;
12             break;
13         }else{
14             valorRegreso=1;
15         }
16     }
17     return valorRegreso;
18 }
```

Ilustración 3 explicación de esNumerico

Pasando a la parte principal del programa, tenemos las referencias a los hilos que ya hemos establecido y que son necesarias para verificar la validez y tener los parámetros que estamos recibiendo en todos los casos de manera correcta, justo como se aprecia abajo.

```
1  int main(int argc, char *argv []){
2
3      long numeroSeg;
4      char *p;
5      pthread_t hilo_1;
6      pthread_t hilo_2;
7
8      if(argc!=2){
9
10         printf("Numero de parametros incorrectos \n");
11
12     }else{
13
14         if(esNumerico(argv[1])==1){
15             numeroSeg=strtol(argv[1],&p,10);
16             sem_init(&sem1,0,1);
17             pthread_create(&hilo_1,NULL,*hilo_1_funcion,
18 NULL);
19             pthread_create(&hilo_2,NULL,*hilo_2_funcion,
20 NULL);
21             //pthread_join(hilo_1,NULL);
22             //pthread_join(hilo_2,NULL);
23             sleep(numeroSeg);
24             printf("Contador :%d \n",contador);
25         }else{
26             printf(
27 "Escriba un valor numerico mayor a cero \n");
28         }
29     }
30     return 0;
31 }
```

Ilustración 4 explicación de main

Ya finalmente en nuestro código mostrados en la ilustración 5 y la ilustración 6, lo único que tenemos que hacer es la llamada lo función del hilo uno y la llamada en la función del hilo dos, para generar los decrementos e incrementos controlados por medio del uso de los semáforos, queremos destacar que para no ser tan reiterativos o redundantes la explicación a más detalle la hemos incluido la parte de abajo con la explicación de todo el código en su conjunto.



```
1 static void * hilo_1_funcion(void * arg){
2
3     while(1){
4         sem_wait(&sem1);
5         contador++;
6         sem_post(&sem1);
7     }
8
9 }
```

Ilustración 5 explicación de hilo:1_function



```
1 static void * hilo_1_funcion(void * arg){
2
3     while(1){
4         sem_wait(&sem1);
5         contador++;
6         sem_post(&sem1);
7     }
8
9 }
```

Ilustración 6 explicación de hilo_2_function

Código completo (En formato de texto y con explicación general abajo).

A continuación, se presenta todo el código implementado, pero cabe resaltar que hay una explicación general y resumida del mismo en la parte inferior del mismo en la sección destinada a estos menesteres.

```
#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 númeroSeg;
    char *p;
    pthread_t hilo_1;
    pthread_t hilo_2;
    if(argc!=2){
        printf("Número de parametros incorrectos \n");
    }else{
        if(esNumerico(argv[1])==1){
            númeroSeg=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(númeroSeg);
            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 general código.

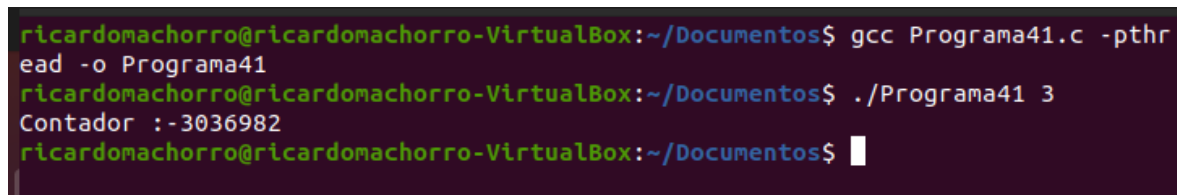
En primera instancia, se declara un semáforo y contador global para usar en el resto del código. Dentro del “main” primero se chequea por medio de un if aplicado a la variable argc si es que el número de parámetros de entrada es correcto y al llamar al código es exactamente 2, que en caso de no cumplirse manda un mensaje diciendo que el número de parámetros es incorrecto. En caso de que los condicionantes se cumplan en términos del número de parámetros y se supere el primer if, se chequea si el segundo argumento proveniente debe ser 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 es igual a cero.

Si se cumple la condición con la función esNumerico se toma el segundo parámetro de argumentos del programa y por medio de la función se pasa a un valor para el número 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 primer 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 decreuenta el contador. Luego el programa principal espera o se duerme por los segundos ingresados y termina imprimiendo el valor del contador.

Ejecución:

En la siguiente imagen (ilustración 7) podemos ver la compilación de nuestro programa con el uso de la directiva especial y también podemos apreciar la compilación del mismo dando como resultado lo que precisamente esperábamos.



```

ricardomachorro@ricardomachorro-VirtualBox:~/Documentos$ gcc Programa41.c -pthread
-ead -o Programa41
ricardomachorro@ricardomachorro-VirtualBox:~/Documentos$ ./Programa41 3
Contador :-3036982
ricardomachorro@ricardomachorro-VirtualBox:~/Documentos$

```

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 genera 10 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 completo.

A continuación, podemos ver el código que fue implementado en esta parte de la práctica, pero cabe destacar que la explicación al mismo se encuentra en la parte de abajo en la sección destinada para ello.

1) Código completo derivado.c

```
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

#define LLENADO 0
#define listo 1
#define Nolisto -1

struct memoria {
    char buff[300];
    int estado, primer_pid1, segundo_pid2;
};
struct memoria* shared_memory_seguidor;
void manejador(int signum) {
    // Recibimos mensaje
    if (signum == seguidor_dos) {
        printf("Recibo = ");
        puts(shared_memory_seguidor->buff);
    }
}

int main() {
    /*
        Acceso al identificador del derivado
    */
    int pid = getpid();
```



```

int shared_memory_identificador;
/*
    Generamos una llave para nuestro
    sistema de memoria compartida.

    *Procedemos a crear la memoria
    compartida.

    *Pegamos la memoria compartida.

*/
int llave = 123;
shared_memory_identificador = shmget(llave, sizeof(struct memoria), IPC_CREA
T | 0666);
shared_memory_seguidor = (struct memoria*)shmat(shared_memory_identificador,
NULL, 0);

// Guardamos la ID en la memoria compartida
// Guardamos el estado en la memoria compartida.
shared_memory_seguidor->segundo_pid2 = pid;
shared_memory_seguidor->estado = Nolisto;

// usamos la seña para el llamado y la
// interacción con memoria compartida.
signal(seguidor_dos, manejador);

while (1) {
    sleep(1);
    printf("-----Escritura == ");
    fgets(shared_memory_seguidor->buff, 300, stdin);

    /*Enviamos la info al otro lado con
    nuestra función de KILL*/
    shared_memory_seguidor->estado = listo;
    kill(shared_memory_seguidor->primer_pid1, seguidor_uno);
    while (shared_memory_seguidor->estado == listo)
        continue;
}shmdt((void*)shared_memory_seguidor);
return 0;
}

```

2) Código completo principal.c

La explicación a este código se encuentra en la parte inferior, en la sección destinada para ello. (En el siguiente subtítulo)

```

#include <signal.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

```

```

#define LLENADO 0
#define Listo 1
#define Nolisto -1

struct memoria {
    char buff[300];
    int status, primer_pid1, segundo_pid2;
};
struct memoria* shared_memory_seguidor;
void manejador(int signum) { }
    //Recibimos mensajes.
    if (signum == seguidor_uno) {
        printf("Recibo derivado ==");
        puts(shared_memory_seguidor->buff);
    }
}

int main() {
    /*
        Acceso al identificador del derivado
    */
    int pid = getpid();

    int shared_memory_identificador;
    /*
        Generamos una llave para nuestro
        sistema de memoria compartida.

        *Procedemos a crear la memoria
        compartida.

        *Pegamos la memoria compartida.

    */
    int llave = 123;
    shared_memory_identificador = shmget(llave, sizeof(struct memoria), IPC_CREA
T | 0666);
    shared_memory_seguidor = (struct memoria*)shmat(shared_memory_identificador,
NULL, 0);

    // Guardamos la ID en la memoria compartida
    // Guardamos el estado en la memoria compartida.
    shared_memory_seguidor->primer_pid1 = pid;
    shared_memory_seguidor->status = Nolisto;

    // usamos la seña para el llamado y la
    // interacción con memoria compartida.
    signal(seguidor_uno, manejador);

    while (1) {
        while (shared_memory_seguidor->status != Listo)
            continue;
        sleep(1);
        printf("-----Principal== ");
        fgets(shared_memory_seguidor->buff, 100, stdin);
    }
}

```

```

        /*Enviamos la info al otro lado con
        nuestra función de KILL*/
        shared_memory_seguidor->status = LLENADO;
        kill(shared_memory_seguidor->segundo_pid2, seguidor_dos);
    }
    shmdt((void*)shared_memory_seguidor);
    shmctl(shared_memory_identificador, IPC_RMID, NULL);
    return 0;
}

```

Explicación general del código.

En la práctica todo el código que hemos desarrollado ha hecho uso de la implementación de la memoria compartida dentro del sistema operativo Ubuntu, ha sido bastante interesante ver como es el uso de la memoria compartida y como través del uso de una llave directamente podamos hacer que de manera directa el código pueda interactuar con lo que es almacenado en esa sección de memoria, la cuestión que fue más desafiante de la practica fue poder hacer que precisamente un array se enviará, ya que de hecho es bastante sencillo poder enviar un dato por sí solo pero la parte el arreglo es más complicada por la naturaleza del mismo, lo que es un hecho es que en trabajar con memoria compartida es una gran ventaja en comparación al trabajo que se había hecho el prácticas anteriores con directamente colas, más sin en cambio consideramos que es una parte esencial en la implementación de programas que puedan hacer uso realmente los recursos que tiene el sistema y la verdad es que nos sentimos bastante entusiasmados respecto a las posibilidades que esto representa el desarrollo de software llevado múltiples escalones más allá.

Ejecución del código compilado.

En la siguiente imagen (ilustración 8) podemos ver la ejecución del código directamente en nuestra terminal, con el proceso del archivo llamado derivado y el otro llamado principal.

```

luis@luis-compu: ~/Desktop/Programas Luis/42
luis@luis-compu:~/Desktop/Programas Luis/42$ ./principal
1 2 3 4 5 6 7 8 9 10
Recibo derivado ==.
-----Principal== Recibo derivado ==1 2 3 4 5 6 7 8 9 10
-----Principal== 12345678910
Recibo derivado ==1.1 1.2 1.3 1.4 1.5 1.6 1.7
-----Principal== □

luis@luis-compu:~/Desktop/Programas Luis/42$ ./derivado
-----Escritura == .
Recibo = 1 2 3 4 5 6 7 8 9 10
-----Escritura == 1 2 3 4 5 6 7 8 9 10
Recibo = 12345678910
-----Escritura == 1.1 1.2 1.3 1.4 1.5 1.6 1.7
□

```

Ilustración 8 Ejecución del código del programa42.c

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.

3) Vista de los programas en el gestor de archivos.

Primero que nada, procedemos a mostrar los dos distintos archivos con los cual estamos trabajando el código de nuestra solución, esto lo hacemos con la finalidad de que usted pueda ver que lo hemos trabajado de manera separada por medio de dos archivos para poderlos ejecutar en la consola de manera simultánea y poder recibir los mensajes en todo momento por la comunicación bidireccional de los procesos, para no confundir los hemos decidido llamar a uno proceso uno y el otro proceso dos. La imagen de abajo muestra cabalmente lo descrito (ilustración 9)

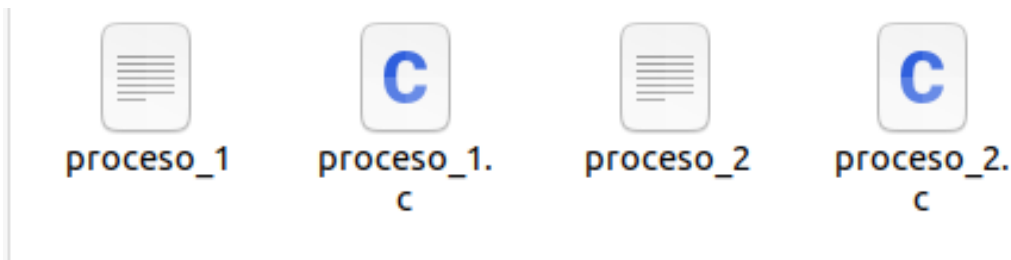


Ilustración 9 Vista de la carpeta en donde están almacenados los 2 archivos que interactúan en la comunicación de procesos bidireccional.

4) Código por partes del proceso_1.c

Primeramente necesitamos hacer todo los importes de las bibliotecas y también hacer algunas definiciones, en este caso hacemos una para la llave de envío y otra definición para la llave de recibo, éstas deberán ser intercambiables entre los dos procesos ya que una llave deberá acceder a la llave de envío y otro llegada llave de recibo dependiendo de qué lado se está enviando, con esto lo que nos referimos es que por ejemplo, la llave de envío número tres deberá concordar con la llave de recibo número tres y por otra parte la llave de recibo número dos deberá concordar con la llave de envío número dos para que de esta forma se cierre el sistema; podemos ver dicha concordancia en la ilustración 10.

```
#include <sys/msg.h>
#include <sys/types.h>
#include <string.h>

#include <stdio.h>
#include <sys/ipc.h>

#define SEND_KEY 3
#define RCV_KEY 2
```

Ilustración 10 Primera parte del código del programa42.c proceso_1.c

Ya una vez hecho esto definiremos una estructura con las definiciones básicas del lugar donde mostraremos el mensaje con la longitud establecida para nuestra cadena de caracteres como se muestra en la ilustración 11.

```
typedef struct msgbuf {
    long tipo_mensaje_evitar_vacios;
    char mensajito[100]; //Aqui va la info del mensaje.
}MSG_BUF;
```

Ilustración 11 Segunda parte del código del programa42.c proceso_1.c

Ahora bien, ya una vez hecho esto es necesario que realicemos algunas validaciones para poder evitar el mensaje sea totalmente inválido, cabe destacar que tenemos dos partes en este código muy esenciales dentro de nuestra función principal una tanto para enviar y otra para recibir, pero debemos tener siempre en consideración que se cumplan con las condiciones deseadas tanto de longitud y tanto de tipo e inclusive de caracteres que requerimos para poder enviar el mensaje de forma satisfactoria. Esto es posible de ver en la ilustración 12

```
int main(){
    int identificadorMensaje; // La id de nuestro mensaje
    MSG_BUF escritor, lector;
    /*
        Iniciamos las validaciones.
    */

    if((identificadorMensaje=msgget(5,IPC_CREAT | 0644))<0)
        perror("error");
    if(fork()==0){
        // El primero lee
        while(1){
            if((msgrcv(identificadorMensaje,&lector,sizeof(lector.mensajito),RCV_KEY,0))<0)
                perror("error");
            printf("Mensaje que te enviaron proceso_2 = %s\n",lector.mensajito);
        }
    }else{
        // El otro escribe
        /*En esta parte podemos hacer la validación
        e caso de que el mensaje no cumpla
        con las características.*/
        while(1){
            escritor.tipo_mensaje_evitar_vacios=SEND_KEY;

            /*
                Impresion
            */
            printf("--[P1]Escribe algo\n");

            scanf("%s",escritor.mensajito);

            if((msgsnd(identificadorMensaje,&escritor,strlen(escritor.mensajito)+1,0))<0)
                perror("error");
        }
    }return 0;
}
```

Ilustración 12 Tercera parte del código del programa42.c (proceso_1.c)

Ahora enfocándose en la parte de lectura, algo que siempre debemos tener en cuenta, es la llave que nos permitirá vincular cada uno los procesos de manera bidireccional de forma que uno pueda tanto enviar al otro como el otro pueda leer los datos y de manera inversa, entonces teniendo esto en mente cabe destacar que ya una vez hecha la impresión para la escritura indicando al usuario, en este caso al proceso que debe escribir le damos la pauta para que entonces por medio de la llave a través de el paso de parámetros por msgnd podamos indicarle de una forma precisa que debemos enviar y también validar en caso de que sea un número inferior a cero, esto lo vemos implementado en el código de a continuación de la ilustración 13.

```
/* El otro escribe
/*En esta parte podemos hacer la validación
e caso de que el mensaje no cumpla
con las características.*/
while(1){
    escritor.tipo_mensaje_evitar_vacios=SEND_KEY;

    /*
        Impresion
    */
    printf("--[P1]Escribe algo\n");

    scanf("%s",escritor.mensajito);

    if((msgsnd(identificadorMensaje,&escritor,strlen(escritor.mensajito)+1,0))<0)
        perror("error");
    }
}return 0;
}
```

Ilustración 13 Cuarta parte del código del programa42.c (proceso_1.c)

5) Código por partes del proceso_2.c

En esencia, como no queremos ser tan redundantes con esta parte el código, pero tampoco queremos dejar cabos sueltos respecto como hicimos la implementación; queremos mostrar que precisamente la parte de las llaves que hemos mencionado que debe ser acoplada entre los distintos procesos en este caso al ser una comunicación bidireccional contaremos con solamente dos iteraciones en las llaves, se puede ver que precisamente la llave de envío coincide con la llave de recibo del otro proceso al igual que con la llave de recibo que coincide con la llave de envío del primero, también hacemos la definición de la estructura en la cual basta con que tengamos un espacio para almacenar la cadena de caracteres que queramos a nuestros mensajes y también otra para el tipo. Vemos en la ilustración 14 lo mencionado.

```
#define SEND_KEY 2
#define RCV_KEY 3

#include <stdio.h>
#include <sys/types.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

typedef struct msgbuf {
    char mensajito[200]; //Aquí va la info del mensaje.
    long tipo_mensaje_evitar_vacios;
} MSG_BUF;
```

Ilustración 14 Primera parte del código del programa42.c (proceso_2.c)

En esta parte no cabe más que decir que del mismo modo como se hizo en el anterior proceso, también se llevó a cabo de una manera muy similar, tal como se ve en la ilustración 15.

```
int main(){
    int identificadorMensaje;
    MSG_BUF escritor, lector;
    /*
     * Iniciamos las validaciones.
     */
    if((identificadorMensaje=msgget(5,IPC_CREAT | 0644))<0)
        perror("error");

    if(fork()==0){
        // el proceso derivado que recibe.
        while(1) {
            if((msgrcv(identificadorMensaje,&lector,sizeof(lector.mensajito),RCV_KEY,0))<0)
                perror("error");
            printf("Mensaje que te enviaron proceso_2 = %s\n",lector.mensajito);
        }
    }
```

Ilustración 15 Segunda parte del código del programa42.c (proceso_2.c)

Como podemos ver en la imagen (ilustración 16) debajo también se hizo la validación para la escritura en el proceso dos, con su correspondiente llave y desde luego también su correspondiente validación en caso de que algunos parámetros no cumplieren, por tanto, tuviésemos un error.

```

    }else{
        // Escribimos en el proceso // PADRE
        /*En esta parte podemos hacer la validación
        e caso de que el mensaje no cumpla
        con las características.*/
        while(1){
            escritor.tipo_mensaje_evitar_vacios=SEND_KEY;

            /*
            Impresion
            */
            printf("--[P2]Escribe algo\n");

            scanf(" %s\n", escritor.mensajito);

            if((msgsnd(identificadorMensaje,&escritor,strlen(escritor.mensajito)+1,0))<0){
                perror("error");
            }
        }
    }
    return 0;
}

```

Ilustración 16 Tercera parte del código del programa42.c (proceso_2.c)

Código completo.

Cabe destacar que la explicación general del código se encuentra abajo, estas imágenes son solo para ver todo en su conjunto.

1) Código completo proceso_1.c

En esta imagen se aprecia la implementación del proceso 1.

```

#include <sys/msg.h>
#include <sys/types.h>
#include <string.h>

#include <stdio.h>
#include <sys/ipc.h>

#define SEND_KEY 3
#define RCV_KEY 2

typedef struct msgbuf {
    long tipo_mensaje_evitar_vacios;
    char mensajito[100]; //Aquí va la info del mensaje.
}MSG_BUF;

int main(){
    int identificadorMensaje; // La id de nuestro mensaje
    MSG_BUF escritor,lector;
    /*
    Iniciamos las validaciones.
    */

    if((identificadorMensaje=msgget(5,IPC_CREAT | 0644))<0)
        perror("error");
    if(fork()==0){
        // EL primero lee
        while(1){
            if((msgrcv(identificadorMensaje,&lector,sizeof(lector.mensajito),RCV_KEY,0))<0)
                perror("error");
            printf("Mensaje que te enviaron proceso_2 = %s\n",lector.mensajito);
        }
    }else{
        // El otro escribe
        /*En esta parte podemos hacer la validación
        e caso de que el mensaje no cumpla
        con las características.*/
        while(1){
            escritor.tipo_mensaje_evitar_vacios=SEND_KEY;

            /*
            Impresion
            */
            printf("--[P1]Escribe algo\n");

            scanf(" %s\n", escritor.mensajito);

            if((msgsnd(identificadorMensaje,&escritor,strlen(escritor.mensajito)+1,0))<0)
                perror("error");
        }
    }
    return 0;
}

```

Ilustración 17 Código completo de proceso_1.c

2) Código completo proceso_2.c.

En esta ilustración se aprecia el proceso2; ambos procesos serán explicados en la siguiente sección.

```
#define SEND_KEY 2
#define RCV_KEY 3

#include <stdio.h>
#include <sys/types.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

typedef struct msgbuf {
    long tipo_mensaje_evitar_vacios;
    char mensajito[100]; //Aqui va la info del mensaje.
} MSG_BUF;

int main(){
    int identificadorMensaje;
    MSG_BUF escritor, lector;
    /*
        Iniciamos las validaciones.
    */
    if((identificadorMensaje=msgget(5,IPC_CREAT | 0644))<0)
        perror("error");

    if(fork()==0){
        // el proceso derivado que recibe.
        while(1) {
            if((msgrcv(identificadorMensaje,&lector,sizeof(lector.mensajito),RCV_KEY,0))<0)
                perror("error");
            printf("Mensaje que te enviaron proceso_2 = %s\n",lector.mensajito);
        }
    }else{
        // Escribimos en el proceso // PADRE
        /*En esta parte podemos hacer la validación
        e caso de que el mensaje no cumpla
        con las características.*/
        while(1){
            escritor.tipo_mensaje_evitar_vacios=SEND_KEY;

            /*
                Impresion
            */
            printf("--[P2]Escribe algo\n");

            scanf(" %[^\\n]",escritor.mensajito);

            if((msgsnd(identificadorMensaje,&escritor,strlen(escritor.mensajito)+1,0))<0){
                perror("error");
            }
        }
    }
    return 0;
}
```

Ilustración 18 Código completo de proceso_2.c

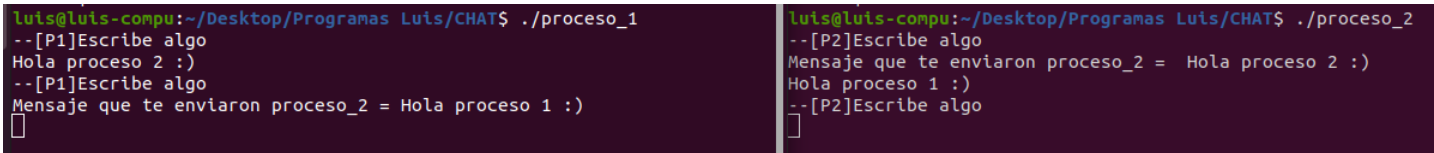
Explicación general del código completo.

1) Explicación general del código de programa proceso_1.c y proceso_2.c

Realmente, la manera en que nuestro programa funciona es bastante simple, como lo mencionamos requerimos del uso de dos llaves una la llave de recibo y otras la llave de envió las cuales deben ser coincidentes con la que tienen otro proceso, claro está pensando siempre de manera inversa, la implementación ha sido llevada a cabo por una cola de mensajes, lo cual es un poco más complicado que si se hubiera hecho con memoria compartida; pero eso sólo nuestra opinión, en términos generales resultó bastante interesante ver cómo es que realmente interactúan los dos procesos de manera conjunta siempre de forma bidireccional por el simple hecho de tener esas llaves que nos dan la pauta para poder hacer que los mensajes lleguen de forma eficiente.

Ejecución del código compilado.

Ya una vez ejecutado y desde luego primeramente compilado nuestro código, procedemos a la ejecución del proceso uno y el proceso dos en el cual nos aparecerán dos mensajes en ambas consolas en donde nosotros podemos escribir algo ya sea del proceso uno al proceso dos o de manera inversa del proceso dos al proceso uno, y como se muestra en pantalla podemos ver cada uno de los mensajes desplegándose en la terminal de nuestro sistema operativo.



```
luis@luis-compu:~/Desktop/Programas Luis/CHAT$ ./proceso_1
--[P1]Escribe algo
Hola proceso 2 :)
--[P1]Escribe algo
Mensaje que te enviaron proceso_2 = Hola proceso 1 :)
□

luis@luis-compu:~/Desktop/Programas Luis/CHAT$ ./proceso_2
--[P2]Escribe algo
Mensaje que te enviaron proceso_2 = Hola proceso 2 :)
Hola proceso 1 :)
--[P2]Escribe algo
□
```

Ilustración 19 Ejecución del código del programa43.c

Programa44.c

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

Código explicado por partes.

Para el desarrollo de esta parte del programa, lo primero que tenemos que hacer tiene que ver directamente con todas las implementaciones de bibliotecas básicas para el manejo primordial del programa, ya una vez que hemos realizado esto, procederemos posteriormente a la creación de la región crítica la cual nos servirá para ambos procesos, tanto para proceso uno como para el proceso dos, en el cual mostraremos el identificador de proceso que está en dicha región crítica y del mismo modo utilizaremos un ciclo que imprimirá a su vez el mensaje de región crítica. Todo esto se puede apreciar en ilustración 20 que aparece en la parte inferior a este párrafo.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<wait.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include<sys/types.h>

//Creamos una región crítica para ambos procesos
void regionCritica()
{
    int i;
    printf("\nProceso en la region critica proceso con pid=%d\n",getpid());
    for(i=0;i<3;i++)
    {
        printf("\nRegión crítica: %d\n",i);
        sleep(1);
    }
}
```

Ilustración 20. Implementaciones necesarias y creación de la región crítica.

Procedemos a la creación de la región crítica tanto para proceso uno como para el proceso dos, con esto podemos indicar a través de un mensaje de texto o el impresión de nuestra terminal precisamente si el proceso dentro de la región no crítica con el identificador esta adentro lo cual le corresponde a la ilustración 21.

```

//creamos la region no critica para el proceso 1
void regionNoCritica()
{
    int i;
    printf("\nEn la region NO critica proceso con pid=%d\n",getpid());
    for(i=0;i<5;i++)
    {
        printf("\nContando(%d): %d\n",getpid(),i);
        sleep(1);
    }
}

//creamos la region no critica para el proceso 2
void regionNoCritica2()
{
    int i;
    printf("\nEn la region NO critica proceso con pid=%d\n",getpid());
    for(i=0;i<7;i++)
    {
        printf("\nContando(%d): %d\n",getpid(),i);
        sleep(1);
    }
}

```

Ilustración 21 Creación de la no región crítica para los procesos

Ahora de manera posterior lo que hemos implementado en nuestra parte principal, es crear el identificador para ambos de los procesos y hacer algunas otras declaraciones para la parte de la memoria compartida, de la misma manera se crearan todas las inicializaciones para la llave compartida como lo son la llaves y la inicialización de los distintos estados para todos los procesos, lo ya mencionado puede ser visto claramente en la ilustración 22.

```

int main()
{
    //identificadores de los dos procesos
    int PID,PID2;

    int shmid1,shmid2,shmid3;
    int *proceso1;
    int *proceso2;
    int *Proceso_favorecido;
    //se crea la memoria compartida para los procesos
    key_t llave1;
    key_t llave2;
    key_t llave3;
    llave1=ftok("Prueba1",'k');
    llave2=ftok("Prueba2",'l');
    llave3=ftok("Prueba3",'m');
    shmid1=shmget(llave1,sizeof(int),IPC_CREAT|0600);
    shmid2=shmget(llave2,sizeof(int),IPC_CREAT|0600);
    shmid3=shmget(llave3,sizeof(int),IPC_CREAT|0600);
    proceso1=shmat(shmid1,0,0);
    proceso2=shmat(shmid2,0,0);
    Proceso_favorecido=shmat(shmid3,0,0);
    //se inicializan en diferente estado los procesos
    *proceso1=1;
    *proceso2=0;
    *Proceso_favorecido=1;
}

```

Ilustración 22 Establecimiento de los identificadores, manejo de la memoria compartida y inicialización de los estados de los procesos.

Ahora se procederá, con la parte de la codificación del algoritmo para la parte del proceso uno, lo cual ya depende enteramente del análisis del mismo, pero que en esencia será prácticamente la misma implementación tanto para el primer proceso como para el segundo proceso, en la imagen de abajo podemos ver la implementación para la parte del proceso uno pero también posteriormente en la siguiente ilustración podremos ver la parte del implementación para el segundo proceso, en la ilustración 23 y 24 podemos apreciar en todo su esplendor su implementación.

```
//se codifica el algoritmo de dekker
if(*proceso1==0)
{
    while (1)
    {
        *proceso1=1;
        while(*proceso2)
        {
            if(*Proceso_favorecido==2)
            {
                *proceso1=0;
                while(*Proceso_favorecido==2);
                *proceso1=1;
            }
        }
        PID=getpid();
        regionCritica(PID);
        *Proceso_favorecido=2;
        *proceso1=0;
        regionNoCritica(PID);
    }
}
```

Ilustración 23 Codificación del algoritmo de Dekker para el proceso 1

```
else
{
    while(1)
    {
        *proceso2=1;
        while(*proceso1)
        {
            if(*Proceso_favorecido==1)
            {
                *proceso2=0;
                while(*Proceso_favorecido==1);
                *proceso2=1;
            }
        }
        PID2=getpid();
        regionCritica(PID2);
        *Proceso_favorecido=1;
        *proceso2=0;
        regionNoCritica2(PID);
    }
}
return 0;
}
```

Ilustración 24 Codificación del algoritmo de Dekker para el proceso 2

Código completo.

A continuación, podremos ver todo el código completo que nuestra solución para el algoritmo ya mencionado y ya implementado.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<wait.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include<sys/types.h>

//Creamos una región critica para ambos procesos
void regionCritica()
{
    int i;
    printf("\nProceso en la region critica proceso con pid=%d\n",getpid());
    for(i=0;i<3;i++)
    {
        printf("\nRegión critica: %d\n",i);
        sleep(1);
    }
}

//creamos la region no critica para el proceso 1
void regionNoCritica()
{
    int i;
    printf("\nEn la region NO critica proceso con pid=%d\n",getpid());
    for(i=0;i<5;i++)
    {
        printf("\nContando(%d): %d\n",getpid(),i);
        sleep(1);
    }
}

//creamos la region no critica para el proceso 2
void regionNoCritica2()
{
    int i;
    printf("\nEn la region NO critica proceso con pid=%d\n",getpid());
    for(i=0;i<7;i++)
    {
        printf("\nContando(%d): %d\n",getpid(),i);
        sleep(1);
    }
}

int main()
{
    //identificadores de los dos procesos
    int PID,PID2;
```

```

int shmid1,shmid2,shmid3;
int *proceso1;
int *proceso2;
int *Proceso_favorecido;
//se crea la memoria compartida para los procesos
key_t llave1;
key_t llave2;
key_t llave3;
llave1=ftok("Prueba1",'k');
llave2=ftok("Prueba2",'l');
llave3=ftok("Prueba3",'m');
shmid1=shmget(llave1,sizeof(int),IPC_CREAT|0600);
shmid2=shmget(llave2,sizeof(int),IPC_CREAT|0600);
shmid3=shmget(llave3,sizeof(int),IPC_CREAT|0600);
proceso1=shmat(shmid1,0,0);
proceso2=shmat(shmid2,0,0);
Proceso_favorecido=shmat(shmid3,0,0);
//se inicializan en diferente estado los procesos
*proceso1=1;
*proceso2=0;
*Proceso_favorecido=1;
//se codifica el algortimo de dekker
if(*proceso1==0)
{
    while (1)
    {
        *proceso1=1;
        while(*proceso2)
        {
            if(*Proceso_favorecido==2)
            {
                *proceso1=0;
                while(*Proceso_favorecido==2);
                *proceso1=1;
            }
        }
        PID=getpid();
        regionCritica(PID);
        *Proceso_favorecido=2;
        *proceso1=0;
        regionNoCritica(PID);
    }
}
else
{
    while(1)
    {
        *proceso2=1;
        while(*proceso1)
        {
            if(*Proceso_favorecido==1)

```

```

        {
            *proceso2=0;
            while(*Proceso_favorecido==1);
            *proceso2=1;
        }
    }
    PID2=getpid();
    regionCritica(PID2);
    *Proceso_favorecido=1;
    *proceso2=0;
    regionNoCritica2(PID);
}
}
return 0;
}

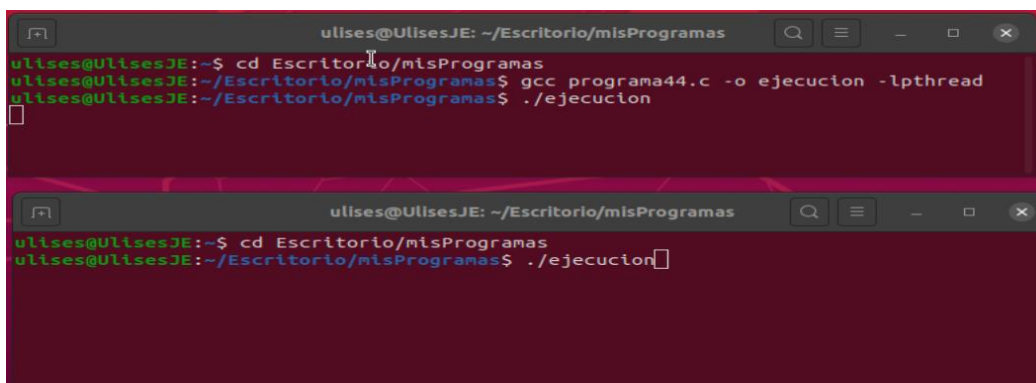
```

Explicación general del código completo.

Básicamente para la creación del programa lo que comenzamos haciendo es la definición de la implementación de las múltiples bibliotecas que requerimos para el trabajo necesario con nuestra implementación de la solución para el algoritmo, lo que debemos considerar son los conceptos básicos de la implementación de la memoria compartida, pero también tenemos que tomar en cuenta toda la parte teórica de las regiones críticas como para lo cual se definirá una región critica la cual será ocupada por algunos los procesos pero también debemos tener en consideración por otra parte que requerimos tener la sección de las regiones no críticas tanto para proceso uno como para el proceso dos, la parte más interesante es la implementación de toda la sección de definiciones necesarias y de declaraciones para poder con esto generar las llaves para la interacción con la memoria compartida, y ya finalmente lo que hacemos en el programa es la implementación del algoritmo lo cual deviene precisamente del análisis del mismo.

Ejecución del código compilado.

Ya explicada la implementación de nuestro código, procedemos al compilar nuestro programa y ejecutarlo directamente la terminal, en donde podemos ver precisamente la parte de la identificación de proceso que se encuentra dentro de nuestra región crítica y como está haciendo uso de nuestro contador como para lo cual hemos mostrado dos capturas de la terminal para poder ejemplificarlo de la mejor forma posible, esto se aprecia en la ilustración 25 y 26.



```

ulises@UlisesJE: ~/Escritorio/misProgramas
ulises@UlisesJE:~$ cd Escritorio/misProgramas
ulises@UlisesJE:~/Escritorio/misProgramas$ gcc programa44.c -o ejecucion -pthread
ulises@UlisesJE:~/Escritorio/misProgramas$ ./ejecucion

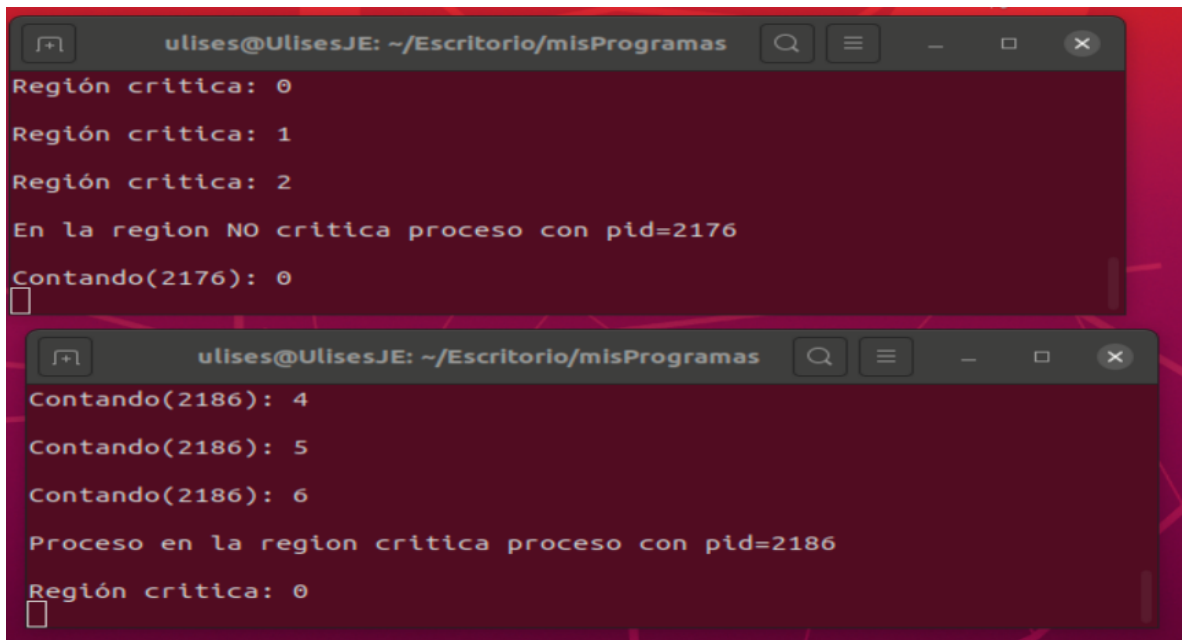
```

```

ulises@UlisesJE:~/Escritorio/misProgramas
ulises@UlisesJE:~$ cd Escritorio/misProgramas
ulises@UlisesJE:~/Escritorio/misProgramas$ ./ejecucion

```

Ilustración 25 Ejecución Programa44.c



```
ulises@UlisesJE: ~/Escritorio/misProgramas
Región critica: 0
Región critica: 1
Región critica: 2
En la region NO critica proceso con pid=2176
Contando(2176): 0
█

ulises@UlisesJE: ~/Escritorio/misProgramas
Contando(2186): 4
Contando(2186): 5
Contando(2186): 6
Proceso en la region critica proceso con pid=2186
Región critica: 0
█
```

Ilustración 26 Ejecución de programa44.c segunda ilustración

Programa45.c

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

Código explicado por partes.

En la siguiente imagen (ilustración 27) observamos los elementos de cabecera del programa, una de ellas es la biblioteca estándar de datos de entrada y salida <stdio.h> que nos permite la manipulación de datos, otra de las bibliotecas es la que le permite al programa utilizar hilos ya que para la simulación de los filósofos utilizamos threads, la última biblioteca utilizada es la que permite el uso de semáforos que en este caso es un factor clave para el programa ya que estamos utilizando regiones críticas.

Otras de las cosas que también podemos ver en la imagen son los macros que empleamos en la práctica, los macros nos permiten sustituir frases por valores o inclusive por otras frases, esto para facilitar la lectura del código, lo mencionado se aprecia en la ilustración 27.



```
1
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <stdio.h>
5
6 #define N 5 //num. de filosofos
7 #define IZQ (i-1)%N //vecino izquierdo de i
8 #define DER (i+1)%N //vecino derecho de i
9 #define PENSANDO 0
10 #define CON_HAMBRE 1
11 #define COME 2
12
13
14 pthread_t filos[N]; //hilos que representan a los filósofos
15 sem_t mutex ; //semáforo para la sección crítica
16 sem_t s[N]; //semáforos para los filósofos
17 int estado [N] ; //estado actual de cada filósofo
18
```

Ilustración 27 Bibliotecas y macros

En esta imagen (ilustración 28) observamos la función pensar, la cual simula un estado de análisis del filósofo, este estado no le permitirá al filósofo comer por lo que no tendrá que utilizar los tenedores que se encuentran a sus lados. Aquí hacemos uso de uno de los macros que es pensar cuyo valor que sustituye es el 0, este valor se le asigna al estado del filósofo i.

```

1  /*
2   el filosofo i va a perder el tiempo... (va a pensar)
3  */
4  void pensar (int i)
5  {
6      int t ;
7      t = rand() % 11;
8      printf("Filosofo %d pensando \n", i) ;
9      estado[i] = PENSANDO;
10     sleep (t) ;
11 }

```

Ilustración 28 pensar

Ahora estamos en la función comer, esta función recibe el valor del filósofo que va a comer, su estado cambia a COME y duerme el hilo durante 5 segundos, como lo muestra la ilustración 29.

```

1  /*
2   El filosofo i, va a comer !!!!!!!
3  */
4  void comer (int i)
5  {
6      printf("Filosofo %d esta comiendo un caballo \n", i);
7      estado[i] = COME;
8      sleep (5);
9  }
10

```

Ilustración 29 comer

Llegamos a la función verificar, en esta función lo que hacemos es verificar, valga la redundancia, si el filósofo puede tomar ambos tenedores, recordemos que esta es la principal condición para que el filósofo pueda cambiar al estado comer, si el filósofo tiene acceso a ambos tenedores entonces podrá cambiar al estado comer, de lo contrario tendrá que esperar a que sus dos tenedores compartidos estén disponibles para él, ejemplo claro se aprecia en la ilustración 30.

```

1  /*
2   Verifica que pueda tomar ambos tenedores
3  */
4  void verifica(int i)
5  {
6      if( estado[i]==CON_HAMBRE && estado[IZQ]!=COME && estado[DER]!=COME ){
7
8          estado[i] = COME;
9          printf("Filosofo %d comiendo\n", i) ;
10         sem_post(&s[i]);
11     }
12 }

```

Ilustración 30 verifica

Para la función `toma_tndrs` lo que se simula es que el filósofo `i` tome ambos tenedores, si esto sucede el filósofo tendrá la posibilidad de cambiar al estado comer. Primero entra a la región crítica para tomar los tenedores y después sale para dejar que alguien más entre a la región, lo cual puede ser visto en la ilustración 31.

```

1  /*
2   El filosofo i intenta tomar Los tenedores
3  */
4  void toma_tndrs(int i)
5  {
6
7      sem_wait(&mutex);          //
8      //entra a la sección crítica, hace uso del semaforo
9      estado[i] = CON_HAMBRE;    //dice: tengo mucha hambre
10     verifica(i);               // verifica que pueda tomar Los tenedores
11     sem_post(&mutex);          //
12     //sale de la sección crítica y el sem. puede permitir la entrada a alguien más
13     sem_wait(&s[i]);            //se bloquea si no consiguió Los tenedores
14 }

```

Ilustración 31 tomar tenedores

Para la función `deja_tndr` el filósofo dejará los tenedores y se pondrá a pensar. Al dejar los tenedores les permitirá a los demás filósofos tomar los tenedores y comer si es que pueden tener dos al mismo tiempo, en la imagen 32 se aprecia claramente.

```

1  /*
2   el filosofo i dejar los tenedores
3  */
4  void deja_tndrs(int i)
5  {
6
7      sem_wait(&mutex);    // de nuevo entra a la sección crítica
8      estado[i] = PENSANDO; //deja de comer y se pone a pensar
9      verifica(IZQ);
10     verifica(DER);
11     sem_post(&mutex);
12 }

```

Ilustración 32 dejar tenedores

En esta parte del programa (ilustración 33) establecemos las capacidades de los filósofos, las cuales son: pensar, comer, tomar tenedores y dejar los tenedores, para el cambio de estados se deben cumplir ciertas condiciones, la principal es que para cambiar al estado comer, el filósofo necesita tener ambos tenedores.

```

1  /*
2  void * filosofos (int i)
3  {
4      int j ;
5
6
7      for (; ; )
8      {
9          pensar(i) ;
10         toma_tndrs(i) ;
11         comer(i) ;
12         deja_tndrs(i) ;
13     }
14 }

```

Ilustración 33 filósofos

Para terminar la función main como es mostrada en la ilustración 34, creara a los hilos que representaran a los filósofos, cada filosofo cuenta con los estados establecimos previamente y el cambio de estado será únicamente si cumplen las condiciones para el cambio.

```

1  main()
2  {
3      int i ;
4
5
6      for(i = 0; i < 5; i++){
7          sem_init (&s[i], 0, 1);
8
9
10         estado[i] = PENSANDO ;
11     }
12
13
14     sem_init (&mutex, 0, 1);
15
16     //creamos un hilo de ejecucion para cada filosofo, que ejecuta filosofos()
17     for (i=0; i<N; i++){
18         pthread_create(&filos[i], NULL, (void *)filosofos,(void *) i);
19
20     //cada hilo espera a que terminen Los dem@s y Libera Los recursos
21     for (i=0; i<N; i++){
22         pthread_join(filos[i],NULL);
23     }
24
25 }

```

Ilustración 34 main filósofos

Código completo.

Este código es todo lo que se implementa y cuenta con su sección de descripción de todo el conjunto en la siguiente sección.

```

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5                //num. de filósofos
#define IZQ (i-1)%N        //vecino izquierdo de i
#define DER (i+1)%N        //vecino derecho de i
#define PENSANDO 0
#define CON_HAMBRE 1
#define COME 2

pthread_t filos[N];        //hilos que representan a los filósofos
sem_t mutex ;              //semáforo para la sección crítica
sem_t s[N];                //semáforos para los filósofos
int estado [N] ;           //estado actual de cada filósofo

/*
    el filosofo i va a perder el tiempo... (va a pensar)
*/
void pensar (int i)

```

```

{
    int t ;
    t = rand() % 11;
    printf("Filosofo %d pensando \n", i) ;
    estado[i] = PENSANDO;
    sleep (t) ;
}

/*
    El filosofo i, va a comer !!!!!!!
*/
void comer (int i)
{
    printf("Filosofo %d esta comiendo un caballo \n", i);
    estado[i] = COME;
    sleep (5);
}

/*
    Verifica que pueda tomar ambos tenedores
*/
void verifica(int i)
{
    if( estado[i]==CON_HAMBRE && estado[IZQ]!=COME && estado[DER]!=COME ){
        estado[i] = COME;
        printf("Filosofo %d comiendo\n", i) ;
        sem_post(&s[i]);
    }
}

/*
    El filosofo i intenta tomar Los tenedores
*/
void toma_tndrs(int i)
{
    sem_wait(&mutex);           //entra a la seccion critica, hace uso del
    semaforo                    //semaforo
    estado[i] = CON_HAMBRE;      //dice: tengo mucha hambre
    verifica(i);                 // verifica que pueda tomar los tenedores
    sem_post(&mutex);            //sale de la seccion critica y el sem. puede
    permitir la entrada a alguien mas
    sem_wait(&s[i]);              //se bloquea si no consigue los tenedores
}

/*
    el filosofo i dejar los tenedores
*/

```

```

void deja_tndrs(int i)
{
    sem_wait(&mutex);      // de nuevo entra a la seccion critica
    estado[i] = PENSANDO; //deja de comer y se pone a pensar
    verifica(IZQ);
    verifica(DER);
    sem_post(&mutex);
}

/*

*/
void * filosofos (int i)
{
    int j ;

    for (; ; )
    {
        pensar(i) ;
        toma_tndrs(i) ;
        comer(i) ;
        deja_tndrs(i) ;
    }
}

main()
{
    int i ;

    for(i = 0; i < 5; i++){
        sem_init (&s[i], 0, 1);

        estado[i] = PENSANDO ;
    }

    sem_init (&mutex, 0, 1);

    //creamos un hilo de ejecucion para cada filosofo, que ejecuta filosofos()
    for (i=0; i<N; i++)
        pthread_create(&filos[i], NULL, (void *)filosofos,(void *) i);

    //cada hilo espera a que terminen los demas y libera los recursos
    for (i=0; i<N; i++){
        pthread_join(filos[i],NULL);
    }
}

```

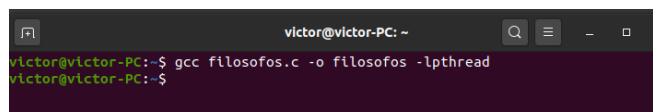
```
}  
  
}
```

Código general del código por partes.

Este programa es una simulación de 5 filósofos que se encuentran en una mesa redonda, cada uno tiene 2 estados, comer y pensar, además cada uno cuenta con un par de tenedores que se encuentra a sus costados, el detalle es que estos tenedores son compartidos entre los filósofos, pues el tenedor derecho de uno es el tenedor izquierdo de otro, y la condición principal para que al menos uno de ellos pueda comer es que tenga ambos tenedores en su poder. La implementación del programa es por medio de hilos y de memoria compartida, que a su vez esta administrada por la variable semáforo.

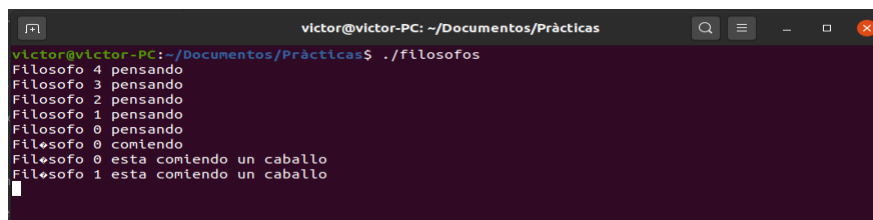
Ejecución del código compilado.

El programa simula el problema de los filósofos en donde únicamente pueden comer si y solo si tienen ambos tenedores en su poder, de lo contrario tendrán que esperar a que los tenedores estén disponibles. El programa fue compilado mediante el comando `gcc filósofos.c -o filósofos -lpthread`, apreciamos su compilación en la ilustración 35 y en la 36 y 37 la ejecución del mismo.



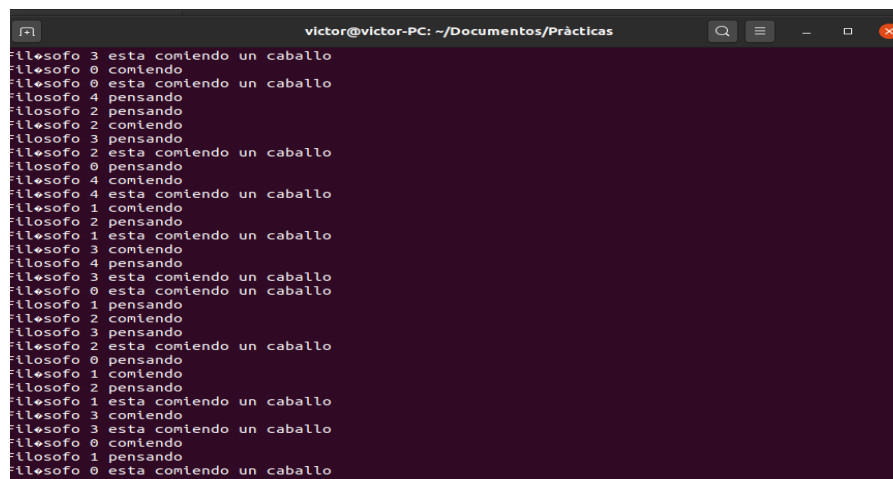
```
victor@victor-PC: ~  
victor@victor-PC:~$ gcc filósofos.c -o filósofos -lpthread  
victor@victor-PC:~$
```

Ilustración 35 Compilación



```
victor@victor-PC: ~/Documentos/Prácticas  
victor@victor-PC:~/Documentos/Prácticas$ ./filósofos  
Filósofo 4 pensando  
Filósofo 3 pensando  
Filósofo 2 pensando  
Filósofo 1 pensando  
Filósofo 0 pensando  
Filósofo 0 comiendo  
Filósofo 0 esta comiendo un caballo  
Filósofo 1 esta comiendo un caballo
```

Ilustración 36 ejecución



```
victor@victor-PC: ~/Documentos/Prácticas  
Filósofo 3 esta comiendo un caballo  
Filósofo 0 comiendo  
Filósofo 0 esta comiendo un caballo  
Filósofo 4 pensando  
Filósofo 2 pensando  
Filósofo 2 comiendo  
Filósofo 3 pensando  
Filósofo 2 esta comiendo un caballo  
Filósofo 0 pensando  
Filósofo 4 comiendo  
Filósofo 4 esta comiendo un caballo  
Filósofo 1 comiendo  
Filósofo 2 pensando  
Filósofo 1 esta comiendo un caballo  
Filósofo 3 comiendo  
Filósofo 4 pensando  
Filósofo 3 esta comiendo un caballo  
Filósofo 0 esta comiendo un caballo  
Filósofo 1 pensando  
Filósofo 2 comiendo  
Filósofo 3 pensando  
Filósofo 2 esta comiendo un caballo  
Filósofo 0 pensando  
Filósofo 1 comiendo  
Filósofo 2 pensando  
Filósofo 1 esta comiendo un caballo  
Filósofo 3 comiendo  
Filósofo 3 esta comiendo un caballo  
Filósofo 0 comiendo  
Filósofo 1 pensando  
Filósofo 0 esta comiendo un caballo
```

Ilustración 37 ejecución

Conclusiones.

Chavarría Vázquez Luis Enrique.

Primero que nada, cabe destacar, que en esta práctica se logró 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 con el flujo 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 estudiante. La parte que más disfruté es que con esta práctica conseguí la oportunidad de entender cómo funcionan los hilos, trabajar directamente en implementación, también trabajar en un proyecto con el uso de memoria compartida.

La parte del desarrollo del chat, una de las cosas que más me enseñó 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 formas de implementarlo a mayor escala.

Para ser honesto, la información de la comunicación entre procesos y las múltiples herramientas que existen es bastante amplia, con lo que veo muchas posibilidades y por supuesto facilidades para llevar aplicaciones a la práctica.

Ya finalmente si nos remontamos a problemas más clásicos, como lo es el problema de los filósofos ,notamos pues que también puede 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 ficticios pero que trasladándolos 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 es que he aprendido mucho sobre como implementarlo en términos de la comunicación que hay entre ellos.

Juárez Espinoza Ulises.

La práctica me ayudo a comprender como funcionan las diferentes herramientas del IPC y como ayudan a la comunicación entre procesos dentro del sistema operativo LINUX-UNIX, IPC, es decir como desde un proceso enviar un dato a otro, y de esta forma, comunicarlos. Son variados estos mecanismos y dependiendo de la complejidad de la tarea es más optimo utilizar uno u otro. Siendo en un principio me costó comprender algunos códigos relacionados con los procesos, ya que no había tenido contacto con nada relacionado antes. Sin embargo, documentándome un poco hizo que todo se volviera fácil de entender, sobre todo porque de eso va el sistema operativo, se la pasa creando procesos, comunicándolos sin que nosotros lo sepamos.

Las herramientas del IPC entonces nos ayudan realizar programas que realizan operaciones a nivel sistema operativo y así comprender mejor de que va y como funciona el sistema operativo.

Primero quiero comenzar hablando de la memoria compartida cada proceso en Linux mantiene un rango de direcciones de memoria virtual para poder trabajar, un proceso no puede modificar información del mapa de memoria de otro proceso, pero si se puede definir segmentos de memoria compartidos que le permitan a otro proceso acceder a sus datos, y modificarlos, en el caso de que sea necesario. Claro está que se puede ocupar para resolver algunos problemas como el de la alternancia estricta. Otro IPC a destacar son las colas de mensajes son importantes porque permite a dos procesos, incluso procesos no relacionados, a enviarse mensajes y datos entre sí, un ejemplo de esto fue el Programa43.c.

Por ultimo y no menos importante están los semáforos muy adecuado ese nombre ya que realizan exactamente esa función, permiten o deniegan el acceso a ciertos recursos, a un proceso. Lo interesante de esto es que se puede tener varios procesos concurrentes tratando de acceder o modificar algún archivo, y con ayuda de los semáforos podríamos dar en acceso en determinado momento a un proceso y dejar en espera al resto, hasta que el recurso se desocupe, tal es el caso de la alternancia estricta o con el problema de los filósofos.

Me impresiona la flexibilidad que ofrece el sistema operativo, y también la flexibilidad que muestra c para programar mecanismos que se ejecutan a nivel operativo, sobra decir que se pueden combinar varios de estos mecanismos dependiendo de la necesidad, tal es el caso de la memoria compartida y los semáforos para resolver el problema de la alternancia estricta, también me sorprende la comunicación entre procesos, sobre todo porque es algo que no se ve, pero que ahora sé que existe y que puedo jugar un poco con ella, con ayuda de C y los IPC las opciones son infinitas y estoy ansioso de conocer más de este sistema operativo.

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 molestar. 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 cómo se pueden solucionar.

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

La memoria compartida es una de las formas con las cuales el sistema Linux permite la comunicación entre procesos, aunque este es uno de los métodos con más riesgos de utilizar pues si no establecemos una correcta conexión podemos llegar a bloquear el sistema completo. El tema de memoria compartida me pareció sencillo de comprender, pero la implementación fue otra cosa, en esta ocasión si me costó trabajo poder lograr una correcta comunicación entre procesos.

Durante la realización de la practica también aplicamos nuevos conceptos como semáforos que llego a ser la solución a un problema que se trató de resolver con diferentes métodos, algunos de ellos fueron la mejor opción durante algún tiempo, pero conforme paso el tiempo y las necesidades cambiaron, se dieron cuenta que esa solución no era la mejor. Los semáforos son la mejor opción porque eliminan el problema de dependencia entre procesos y no se desperdicia tiempo de procesamiento.

EL programa de los filósofos fue uno de los más complicados para realizar porque pone en juego todos los conceptos, memoria compartida, semáforos e hilos. El primer contacto que tuve con hilos lo hice en programación Orientada a Objetos, pero en ese momento únicamente los utilicé y no le puse demasiada atención a la teoría detrás, gracias a esta práctica comprendí la diferencia entre hilos y procesos.

Esta práctica me hizo reflexionar con respecto a algunas cosas. Los conceptos trabajados en esta ocasión fueron creados hace mucho tiempo cuando las características de los ordenadores eran menores a las actuales, antes únicamente se contaba con un solo núcleo y actualmente podemos tener hasta 8 núcleos en una sola computadora, además sabemos que todo proceso requiere del núcleo para poder ejecutarse, nosotros en nuestra actual época de formación como programadores, estamos trabajando con técnicas que estas orientadas a un sistema mono núcleo, no nos ponemos a pensar que las actuales computadoras pueden realizar múltiples cosas de manera simultánea, eso es algo que se conoce como programación concurrente. Este paradigma tiene como objetivo tener la capacidad de “decirle” al SO que los procesos a realizar se pueden separar en diferentes procesos cada uno ejecutándose en un núcleo, actualmente esto ya sucede gracias a los sistemas operativos, pero únicamente sucede cuando el sistema mismo se da cuenta que puede dividir los procesos, pero si no lo nota, no sucede y todo se realiza en un solo núcleo. Si nosotros como programadores comenzamos a desarrollar programas pensados en la concurrencia, creo que los alcances de la computación cambiaran y aprovecharemos mejor los recursos disponibles.

Bibliografía

- [1] profesoreselo, «profesoreselo,» profesoreselo, 2020. [En línea]. Available: <http://profesores.elo.utfsm.cl/~agv/elo330/2s08/lectures/IPC.html>. [Último acceso: 2020].
- [2] ulpgc, «ulpgc,» ulpgc, 2020. [En línea]. Available: http://sopa.dis.ulpgc.es/progsis/material-didactico-teorico/tema5_1transporpagina.pdf. [Último acceso: 2020].
- [3] UBUNTU, «UBUNTU MANUALS,» 2020. [En línea]. Available: <http://manpages.ubuntu.com/manpages/bionic/es/man2/msgget.2.html>. [Último acceso: 2020].
- [4] UBUNTU, «UBUNTU MANUALS,» UBUNTU, 2020. [En línea]. Available: <http://manpages.ubuntu.com/manpages/bionic/es/man2/msgctl.2.html>. [Último acceso: 2020].