INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

SISTEMAS OPERATIVOS

PRÁCTICA 4:

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

ALUMNOS:

Martínez Ramírez Sergi Alberto – 2020630260 - 2CV15
Meza Vargas Brandon David – 2020630288 - 2CM15
Peña Atanasio Alberto – 2020630367 - 2CV15
Sarmiento Gutiérrez Juan Carlos – 2020630386 - 2CV15

PROFESOR:

José Alfredo Jiménez Benítez

Índice de contenido

Índice de figuras

Glosario de términos

IPC: abreviación de Inter Process Communicaction, comprende una serie de mecanismos que permiten la comunicación de procesos.

Memoria compartida: intercambio rápido de datos entre procesos, no hay sincronización.

Semáforos: elementos para sincronizar procesos, se basan en aumentar o decrementar su valor de forma atómica.

Colas de mensajes: utilidad para intercambio de mensajes entre profesos de una máquina. También empleadas para sincronizar procesos.

Sincronización: es la transmisión y recepción de señales que tiene por objetivo llevar a cabo el trabajo de un grupo de procesos cooperativos.

Proceso: programa en ejecución.

Hilo: un hilo es un proceso ligero o subproceso es una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo.

Contenido

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

Es muy común que los procesos se necesitan comunicarse con otros procesos. Se conoce como Inter Processs Communication (IPC)al conjunto de mecanismos clave en los sistemas Unix para llevar a cabo la compartición de datos (intercomunicación) y sincronización entre distintos procesos de forma bastante sencilla.

Actualmente Linux proporciona tres herramientas IPC: colas de mensajes, memoria compartida y semáforos.

Memoria compartida

La herramienta de memoria compartida permite a dos o más procesos compartir un segmento de memoria, y, por consiguiente, los datos que hay en él. Es por ello el método más rápido de comunicación entre procesos.

Los programas que utilizan memoria compartida deben, normalmente, establecer algpun protocolo para el bloqueo. Este protocolo puede ser la utilización de semáforos, que es a su vez otro tipo de comunicación de procesos.

Para hacer uso de la memoria compartida se dispone de 4 llamadas al sistema:

- **shmet():** permite acceder a unza zona de memoria compartida y, opcionalmente, crearla en caso de no existir. Se le pasan tres argumentos: una clave, tamaño del segmento a crear y el flag inicial de operación, y devuelve un entero, denominado shmid, que se utiliza para hacer referencia a dicho segmento
- **shmctl():** proporciona una variedad de operaciones para el control de la memoria compartida
- **shmat():** es la llamada que debe invocar un proceso para adjuntar una zona de memoria compartida dentro de su espacio de direcciones. Recibe tres parámetros: el identificador del segmento, una dirección de memoria y las banderas
- **shmdt():** se desvincula de una zona de memoria compartida. Para ello, deberá precisarse como único parámetro la dirección de memoria de dicha zona

Colas de mensajes

Las colas de mensajes son otra herramienta de IPC que permite a dos procesos, incluso procesos no relacionados, enviarse mensajes y datos entre sí.

Para obtener una cola de mensajes de usa la función msgget(), su sintaxis es: *int msgget(key_t, int msgflg)*

Una cola de mensajes se creará mediante esta función si se cumple alguna de las siguientes condiciones:

- El argumento key toma el valor IPC_PRIVATE

- No existe cola de mensajes en el sistema con clave identificadora igual al valor de key y además se ha especificado el flag IPC_CREAT en el parámetro msgflg

Para enviar una cola de mensajes se usa msgsnd, su sintaxis es: int msgsnd(int msqid, const void *msgp, void *msgp, size_t msgsz, int msgflg)

La función anterior envía un mensaje a la cola asociada al identificador del sistema referenciado por el parámetro *msqid*. El argumento *msgp* es un puntero al comienzo de la zona de memoria de usuario donde esta almacenado el mensaje a enviar, el mensaje debe poseer un campo de comienzo del tipo long en el que sindica el tipo del mensaje, el siguiente campo deberá contener el mensaje propiamente dicho.

Para recibir una cola de mensajes se usa msgrcv, su sintaxis es: int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg)

La función anterior toma un mensaje de la cola especificado por le parámetro *msqid* y lo copia en la zona de memoria apuntada por *msgp*. El mensaje recibido tiene una estructura análoga a la del mensaje enviado por emisor. El argumento *msgtyp* se utiliza para especificar el tipo de mensaje a recibir.

Semáforos

Los semáforos permiten o deniegan el acceso a ciertos recursos. Podríamos, por ejemplo, tener varios procesos concurrentes tratando de modificar un archivo, y para evitar que las escrituras se solapen, podríamos implementar semáforos, que le den el acceso a un proceso, y dejen en espera al reto, hasta que el recurso se desocupe.

Hay tres operaciones fundamentales sobre semáforos:

- semget: crea o toma control de un semáforo
- semctl: operaciones de lectura y escritura del estado del semáforo, también es para destruir el semáforo
- semop: operaciones de incremento o decremento con bloqueo

Desarrollo de la práctica

Programa 41

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. El código completo se encuentra en los anexos.

En la imagen 1 podemos ver las funciones que controlan a los hilos del programa 41.

```c
void *Hilo1(void *segs){ //Inicio de la ejecución del primer hilo

    int i;
    int *seg = (int*)segs; //Parseo de la variable segs
    for(i = 0; i < (*seg); i++){ //for que itera desde 0 a la cantidad dada por el usuario
        sem_wait(&semaforo); //Instrucción que espara una "luz verde del semaforo"
        var++; //Incrementa en uno la variable
        sem_post(&semaforo); //Cambia el valor del semaforo, para indicar que su ejecución
                             //ha finalizado
    }
}

void *Hilo2(void *segs){ //Inicio de la ejecución del segundo hilo

    int i;
    int *seg = (int*)segs;
    for(i = 0; i < (*seg); i++){
        sem_wait(&semaforo);
        var--; //Decrementa en uno la variable
        sem_post(&semaforo);
    }
}
```

Imagen 1. Funciones de los hilos del código 41.

En la imagen anterior podemos observar el código de los hilos que además son independientes, podemos ver que ambos códigos son similares, pero difieren en una cosa.

Primero podemos ver que la función recibe como parámetro una variable llamada segundos, la cual es el dato ingresado por el usuario, que después será parseada a entero, posteriormente hay una declaración de la variable i, variable la cual se utiliza para el for que itera desde 0 hasta el número de segundos ingresados por el usuario, luego se puede apreciar la llamada al sistema “sem_wait(&semaforo)” la cual decrementa el semáforo y si su valor es menor que cero, bloquea el proceso hasta tener un valor mayor o igual que cero. Se utiliza para saber si el recurso está ocupado. Después de esta instrucción hay un incremento en la variable, y aquí es donde las dos funciones se diferencian, ya que el primer hilo incrementa la variable “var” una unidad, mientras el segundo la decrementa en uno. Y por último se encuentra la llamada "sem_pos(&semaforo)” el cuál le indica al semáforo que ha terminado de ocupar el recurso compartido, en este caso la variable var. Hasta que está instrucción no sea ejecutada todos los "sem_wait()” bloquearán a cualquier proceso que quiera ingresar a esta variable. El main del programa lo vemos en la imagen 2.

```c
int main(int argc, char const *argv[]){

    pthread_t id_hilo1, id_hilo2; //Declaracion de las variables hilos
    sem_init(&semaforo, 0, 1); //Inicia el semáforo
    int segs = 0; //Declaración de la variable segs

    printf("Escriba el tiempo a esperar: "); //Impresión en pantalla
    scanf("%d", &segs); //Guarda la entrada del usuario en una variable

    pthread_create(&id_hilo1, NULL, Hilo1, &segs); //Creación del primer hilo
    pthread_create(&id_hilo2, NULL, Hilo2, &segs); //Creación del segundo hilo

    pthread_join(id_hilo1, NULL); //El proceso principal espera a la ejecución
    pthread_join(id_hilo2, NULL); //del primer y segundo hilo, despues continúa

    sleep(segs); //El proceso principal se duerme según la entrada del usuario

    printf("Valor de la variable al final %d \n", var); //Impresión en pantalla
                                                        //del valor final de la variable
    return 0; // Fin del programa
}
```

Imagen 2. Código de la función main() del programa 41.

Primero se puede observar en esta función la declaración de dos variables de tipo pthread_t llamadas id_hilo1 e id_hilo2, estas variables serán los identificadores de los hilos, después se inicializa el semáforo (declarado anteriormente como variable global) con la llamada al sistema "sem_init()". Las siguientes 3 líneas de código son la declaración en la variables segundos y se le solicita ingresar un dato al usuario, posteriormente se procede a crear los hilos con la llamada al sistema "pthread_create()" y se le pasa como atributos; el identificador de hilo declarado anteriormente, la función que queremos sea un hilo y la variable segundos. En estas líneas se ejecuta el código de los hilos, después el proceso padre espera a que terminé la ejecución de ambos hilos para continuar, luego el proceso principal se duerme la cantidad de segundos que el usuario haya ingresado. Y por último se imprime el valor de la variable al final. En la imagen 3 vemos la compilación y ejecución del programa 41.

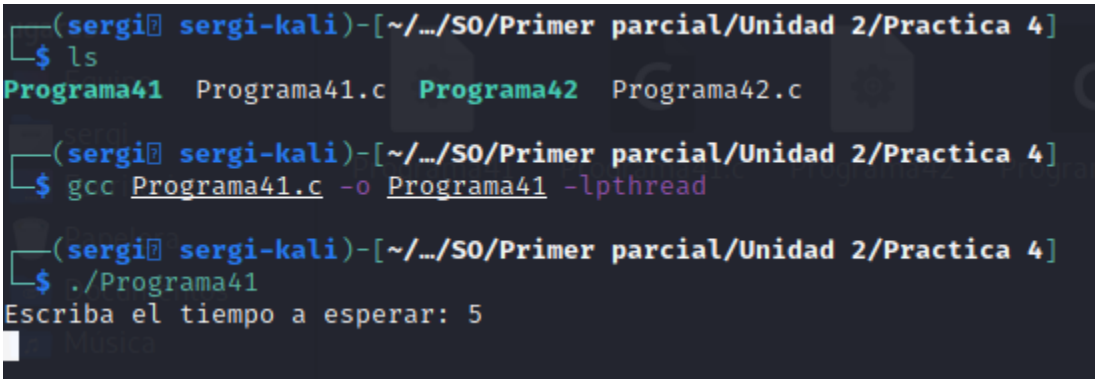

Imagen 3. Compilación y ejecución del programa 41.

En esta imagen se puede observar que al momento de ejecutar se solicita que el usuario ingrese un dato, en este caso el número de segundos, tal vez no se pueda observar en la imagen, sin embargo, el programa espera 5 segundos y después imprime la salida que se ve en la imagen 4.

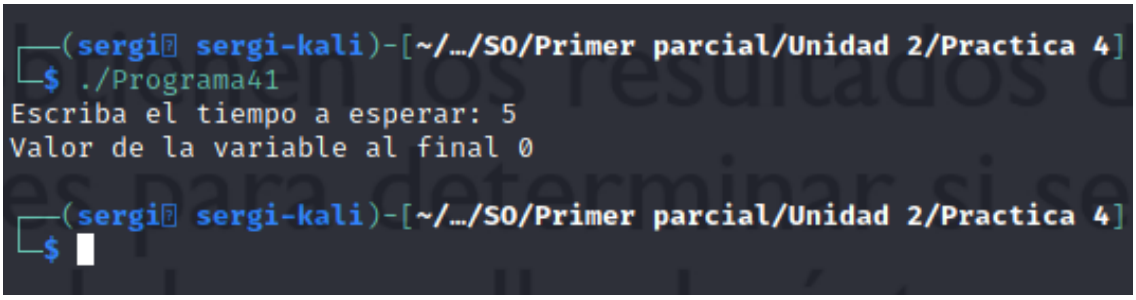

Imagen 4. Salida de la ejecución del programa 41.

Como se puede observar en esta imagen el valor de la variable al final es 0, ya que sin importar que hilo se ejecute primero, nunca podrán acceder a la variable los dos al mismo tiempo, entonces si uno incrementa la variable 500 veces, el otro la decrementará otras 500 veces y viceversa, por eso siempre dará 0.

Programa 42

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. El código completo se encuentra en los anexos. En la imagen 5 vemos una parte del programa 42.

```
key_t key; //Declaración de la variable "key"

key = ftok("Memoria", 'k'); //Convierte una ruta y un proyecto a una llave de IPC del sistema
shmid = shmget(key, (sizeof(float)*10), IPC_CREAT|0600); //Obtiene un segmento de memoria compartida
*random = (float*)shmat(shmid, 0, 0); //Una variable apuntador, apunta la dirección de memoria
                                      //compratida obtenida anteriormente

pid = fork(); //Crea un nuevo proceso
if(pid == -1){ //verifica si el proceso tuvo errores
    perror("Error al crear el proceso"); //En caso de no haberse creado el proceso, manda un error
    exit(-1); //Termina el proceso con mensaje de error
} else if(pid == 0){ //Verifica que el proceso se haya creado correctamente
    for(i = 0; i < 10; i++){ //for que itera desde el 0 hasta el 9
        (*random)[i] = rand() % 100; //Asigna un número aleatorio a la posición actual
    }                                //de la variable random
    exit(0); //Finaliza el proceso
}

for(i = 0; i < 10; i++){ //For que itera desde el 0 hasta el 9
    printf("%0.2f \n", (*random)[i]); //Imprimer la variable compartida
}

pid = wait(&salida); //Espera a que el proceso termine

shmdt(&random); //Desvincula la variable apuntador del segmento de memoria compartida
shmctl(shmid, IPC_RMID, 0); //Elimina el segmento de memoria compartida

return 0; //Fin del programa
```

Imagen 5. Parte del código del programa 42.

En esta imagen se puede observar la declaración de la variable key de tipo key_t, la cuál será la llave para obtener la memoria compartida, posteriormente se le asigna a esta variable una llave de IPC del sistema con la llamada "ftok()", después esa llave es utilizada para obtener el segmento de memoria compartida con la llamada “shmget()” y se guarda en la variable “shmid”, más adelante se le asigna al apuntador "random" la dirección de la memoria compartida que obtuvimos en la línea anterior.

Luego inicia toda la creación del proceso hijo que, mediante un for, asigna a la variable compartida números aleatorios del 0 al 100, una vez el proceso hijo termina de asignarle valores aleatorios a la variable compartida, el proceso padre imprime esa variable y una vez hecho esto, elimina el segmento de memoria compartida para evitar que haya problemas con el sistema operativo posteriormente.

Ahora se muestra la ejecución y compilación del programa 42 en la imagen 6.

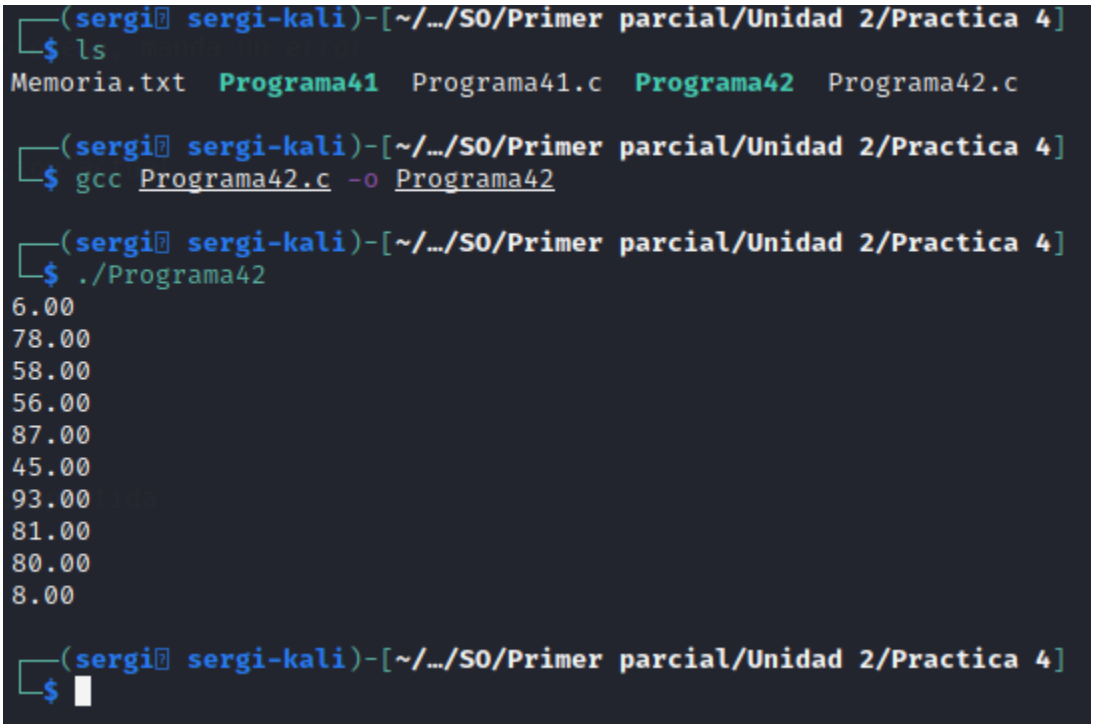

Imagen 6. Compilación y ejecución del programa 42.

Como se puede apreciar, el programa imprime efectivamente un arreglo de 10 números flotantes.

Programa 43

Aquí se solicita lo siguiente:

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

Aquí tendremos dos programas, uno será el proceso A y el otro el proceso B, los programas permiten enviar y recibir colas de mensajes para que se realice su comunicación. El código completo de ambos programas se encuentra incluido en los anexos.

Los programas tendrán varias cosas en común, en la imagen 7 se ve la creación de una estructura.

```
struct msg{
    long MID; //almacena el id del mensaje
    char ms[40];
    pid_t PID;
};
```

Imagen 7. Creación de una estructura.

Como se ve, esta estructura se encargará de guardar el mensaje que se enviara, así mismo servirá para guardar el mensaje que se reciba. Esta estructura está presente en ambos programas.

Posteriormente, creamos la llave que es necesaria para la comunicación, de igual forma creamos la cola de mensajes, esto se ve en la imagen 8.

```
llave=ftok("prueba", 'k');//se crea la llave, los procesos cona cceso al archivo y al entero
                          //pueden acceder a los recursos compartidos

idcmsg=msgget(llave, 0600 | IPC_CREAT);//se crea la cola de mensajes
```

Imagen 8. Creación de la llave y la cola de mensajes.

Como vemos, dentro de msgget hay algunos parámetros, en donde el IPC_CREAT hace que se cree la cola de mensajes, también ponemos la llave que se creó y el 0600 representa permisos de lectura y escritura para el usuario que lance los procesos.

Después vamos a enviar el mensaje, para esto usamos msgsnd, donde los parámetros son el id de la cola de mensajes, la dirección del mensaje, el tamaño total del mensaje, en este caso es una cadena y un entero, y el IPC_NOWAIT indica que, si el mensaje no se puede enviar, que no espere y marque error. Para recibir el mensaje usamos msgrcv, donde los parámetros son el id de la cola de mensajes, la dirección de donde queremos guardar ese mensaje recibido y el tipo de mensaje que queremos recibir, en este caso será tipo 2, esto lo podemos ver en la imagen 9.

```
msgsnd(idcmsg, (struct msgbuf *)&mensaje, sizeof(mensaje.PID)+sizeof(mensaje.ms), IPC_NOWAIT);//se manda un mensaje
printf("He enviado el mensaje\n");

msgrcv(idcmsg, (struct msgbuf *)&mensaje, sizeof(mensaje.PID)+sizeof(mensaje.ms), 2, 0);   //se recibe la cola de mensajes
printf("He recibido un mensaje:\n");
printf("%s, mi PID es %d\n", mensaje.ms, mensaje.PID);//imprime el mensaje recibido
```

Imagen 9. Envió y recibimiento de una cola de mensajes

El código de la imagen anterior es igual al del programa B, solo que en este primero se hace el recibimiento de la cola de mensajes y después se recibe.

Por último, se cierra y borra la cola de mensajes, esto se hace solo en el programa A, lo vemos en la imagen 10.

```
msgctl(idcmsg, IPC_RMID, (struct msqid_ds*)NULL);//se cierra la cola de mensajes
```

Imagen 10. Se cierra la cola de mensajes.

Como se ve, para cerrar una cola de mensajes usamos msgctl, IPC_RMID indica que se quiere borrar.

Ahora bien, en la imagen 11 podemos ver la compilación y ejecución de estos programas.

```
brandon@BDMV:~/programas so/practica 4$ gcc programa43p1.c -o p431
brandon@BDMV:~/programas so/practica 4$ ./p431
He enviado el mensaje
He recibido un mensaje:
Hola soy el proceso 2, mi PID es 5167
```

```
brandon@BDMV:~/programas so/practica 4$ gcc programa43p2.c -o p432
brandon@BDMV:~/programas so/practica 4$ ./p432
He recibido un mensaje:
Hola soy el proceso 1, mi PID es 5166
He enviado el mensaje
```

Imagen 11. Ejecución de los dos programas.

Podemos ver que el programa43p1 es nuestro proceso A y el programa43p2 es el B, por medio de colas de mensajes establecen una comunicación, donde el proceso A manda un mensaje indicando que el es el proceso 1 y su PID. Lo mismo sucede con el proceso B, recibe el mensaje del proceso A y manda de igual forma un mensaje diciendo que él es el proceso 2 y su PID. De esta forma se establece una comunicación bidireccional.

Programa 44

“Programe el Algoritmo de Dekker en dos procesos no emparentados, utilizando memoria compartida ejecutándose en terminales diferentes.” El código completo de este programa se encuentra en los anexos.

Para el desarrollo de esta práctica, se tomó como base el Programa12.c proporcionado por el profesor, posteriormente se crearon dos archivos, para poder simular los procesos no emparentados, estos fueron Programa44a.c y Programa44b.c , en donde cada uno contenía el mismo código, a excepción de algunos detalles. En la imagen 12 vemos el código de las regiones criticas y no críticas.

```
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);
        }
}

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\n",i);
                sleep(1);
        }
}
```

Imagen 12. Código de las regiones crítica y no critica

Posteriormente, estos archivos solo hay una forma de comunicarnos y ese es el uso de la memoria compartida, para ello se crean 3 archivos para las variables “turno” por así decirles, para que la comunicación entre estos procesos sea de forma óptima. Esto lo vemos en la imagen 13.

```
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);
Hijo_desea_entrar=shmat(shmid1,0,0);
Padre_desea_entrar=shmat(shmid2,0,0);
Proceso_favorecido=shmat(shmid3,0,0);
```

Imagen 13. Implementación de memoria compartida

Después se dividieron los procesos del algoritmo de Dekker en donde un padre que es el proceso principal, el cual es un bucle infinito, en el que se está esperando a que un hijo en este caso otro proceso se comunique, para poder activar la siguiente parte que es apagar al padre para que el otro

proceso sea capaz de entrar en la región crítica y finalizar actualizando el archivo de memoria compartida a continuación el código del padre o proceso b mostrado en la imagen 14.

```
while(1){
        *Padre_desea_entrar=1;
        while(*Hijo_desea_entrar){
                if(*Proceso_favorecido==1){
                        *Padre_desea_entrar=0;
                        while(*Proceso_favorecido==1);
                                *Padre_desea_entrar=1;
                }
        }
        regionCritica();
        *Proceso_favorecido=1;
        *Padre_desea_entrar=0;
        regionNoCritica();
}
```

Imagen 14. Código proceso b o padre

De forma similar ocurre en el proceso a solo que en este caso cambia la variable hijo desea entrar, a continuación, la imagen 15 que muestra su código.

```
while (1){
        *Hijo_desea_entrar=1;
        while(*Padre_desea_entrar){
                if(*Proceso_favorecido==2){
                        *Hijo_desea_entrar=0;
                        while(*Proceso_favorecido==2);
                                *Hijo_desea_entrar=1;
                }
        }
        regionCritica();
        *Proceso_favorecido=2;
        *Hijo_desea_entrar=0;
        regionNoCritica();
```

Imagen 15. Código proceso a o hijo

Finalmente, solo hay que inicializar en un archivo las variables y omitir este paso en el otro ya que mediante el código estas estarán alternando su valor a continuación las variables en la imagen 16.

```
*Hijo_desea_entrar=1;
*Padre_desea_entrar=0;
*Proceso_favorecido=1;
```

Imagen 16. Variables inicializadas en proceso a

A continuación, la compilación de los archivos en la imagen 17.

```
juan@juanpc:~/Escritorio/PracticasSO/P4/P44$ gcc Practica44a.c -o P44A
juan@juanpc:~/Escritorio/PracticasSO/P4/P44$ gcc Practica44b.c -o P44B
juan@juanpc:~/Escritorio/PracticasSO/P4/P44$
```

Imagen 17. Compilación éxitos archivos Practica44a.c y Practica44b.c

Finalmente, ambos programas funcionando en terminales distintas como se muestra en la imagen 18.

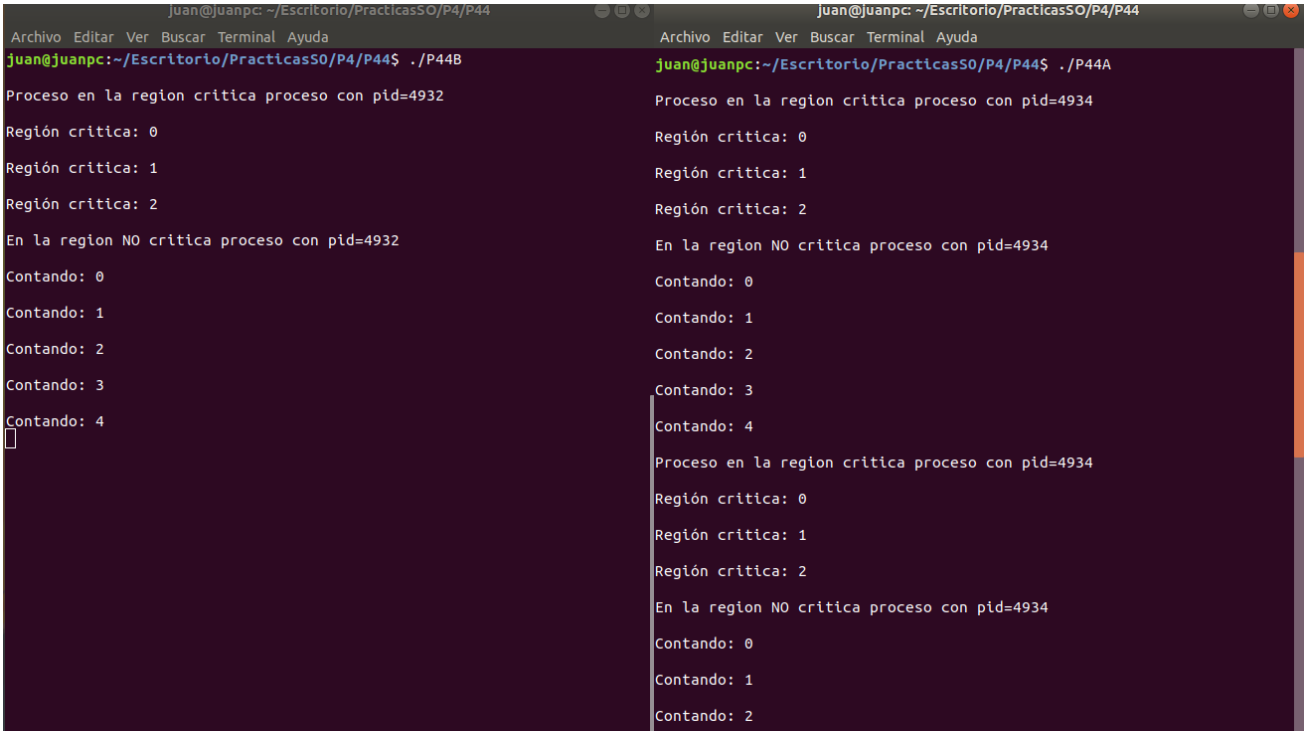

Imagen 18. Programas P44A y P44B funcionando en terminales distintas

Programa 45

Programe el problema de los filósofos utilizando semáforos y memoria compartida. Para que se pueda entender el programa, primero se explicará qué es el problema de los filósofos.

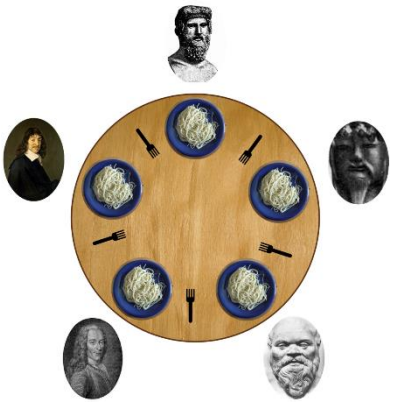

Imagen 19. Ilustración del problema

Cinco filósofos se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato, como se puede observar en la imagen 19. Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer. Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer. Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces los filósofos se morirán de hambre, este bloqueo mutuo se denomina interbloqueo o deadlock. El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.

Para iniciar la resolución del problema, se tiene que identificar el proceso y el recurso necesario. En este caso el platillo será interpretado como un semáforo y el filósofo será visualizado como un proceso. Se definen ciertas constantes que harán que los procesos vayan siendo controlados, la interpretación de todo esto, está plasmado en la Imagen 2, de abajo.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/ipc.h>
6  #include <sys/shm.h>
7  #include <sys/sem.h>
8  #include <time.h>
9
10 #define FALSE   0
11 #define TRUE    1
12 #define NF      5           //Constante que indica el Numero de filósofos
13 #define RIGHT   (i+1)%NF    //Constante de la derecha
14 #define LEFT    (i+NF-1)%NF //Constante de la izquierda
15 #define COMER       2       //El filosofo está comiendo
16 #define HAMBRE      1       //Estado hambriento del filosofo
17 #define PENSAR  0       //Estado pensativo del filosofo
18
```

Imagen 20. Constantes del programa 45.

Posteriormente se muestran en la Imagen 21 los prototipos de las llamadas al sistema que serán utilizadas, para la ejecución del programa.

```
int estado[NF];     //Arreglo donde está el estado del filósofo

void pensar(void);  //Funcion que indica que el filosofo piensa
void comer (void);  //Funcion que indica que el filosofo come
void up(int semid); //Funcion de control del semaforo
void down(int semid);   //Funcion del control de semaforo
int crea_semaforo(key_t key, int valor_ini);
```

Imagen 21. Prototipos del programa 45

A continuación, en la Imagen 22 se declaran las llamadas al sistema fundamentales para las operaciones con los semáforos, la de Up y Down. Todas ellas recibirán una llave y crea_semáforo() un valor inicial.

```
31  int crea_semaforo(key_t key, int valor_ini)
32  {
33      int semid;  //ID del semáforo
34
35      semid = semget(key,1,IPC_CREAT | 0644 );
36
37      if(semid==-1)
38      {
39          printf("\nNo se pudo crear el semaforo");
40          exit(-1);
41
42      }
43
44      semctl(semid,0,SETVAL,valor_ini);
45
46  }
47
48
49  void down(int semid)
50  {
51      struct sembuf op_v[]={0,-1,0};
52      semop(semid,op_v,1);
53
54
55  }
56
57
58  void up(int semid)
59  {
60      struct sembuf op_p[]={0,+1,0};
61      semop(semid,op_p,1);
62
63  }
64
```

Imagen 22. Funciones fundamentales del semáforo para el programa 45.

La Imagen 23 muestra la simulación del filósofo, a la hora del recorrido, permite reconocer si está comiendo o pensando.

```
69  void comer (void)
70  {
71      int j;
72
73      for(j=0;j<NF;j++)
74      {
75          if(estado[j]==COMER)
76          {
77              printf("\nEl filosofo %d esta comiendo",j+1);
78
79          }
80          else if(estado[j]==PENSAR)
81          {
82              printf("\nEl filosofo %d esta pensando",j+1);
83          }
84
85      }
86
87
88  }
```

Imagen 23. Llamada al sistema comer.

De igual manera, la Imagen 24 muestra la llamada al sistema “pensar”, que permite saber, en la simulación, si el filósofo piensa, sí es así, entonces continuamos con el ciclo, sino decimos que está comiendo

```
92  //Función que ayuda a saber si el filosofo está pensando o comiendo
93  void pensar(void)
94  {
95      int j;
96      for(j=0;j<NF;j++)
97      {
98          if(estado[j] == PENSAR)
99          {
100             //Continua
101         }
102         else if(estado[j]==COMER)
103         {
104             printf("\nEl filosofo %d esta comiendo",j+1);
105
106         }
107
108
109     }
110
111
112 }
```

Imagen 24. Llamada al sistema pensar.

Se declaran las variables correspondientes para que funcione el programa, se crean los semáforos con la llamada al sistema crea_semaforo(). Para la llamada ftok() es necesario tener archivos en el mismo directorio y asignarles una llave entera. Esto lo vemos en la imagen 25.

```
115  void main()
116  {
117      int s[NF],i=0, almacena=0,no_filosofo=5,sema_mutex=0;
118      key_t key1=0,key2=0,key3=0,key4=0,key5=0,key6=0;
119
120      printf("1");
121
122      //Creando los semáforos de los filósofos
123      //Creando las llaves e invocando a la funcion
124      key1=ftok("file1",'a');
125      s[0] = crea_semaforo(key1,1);
126
127      key2=ftok("file2",'b');
128      s[1] = crea_semaforo(key2,1);
129
130      key3=ftok("file3",'c');
131      s[2] = crea_semaforo(key3,1);
132
133      key4=ftok("file4",'d');
134      s[3] = crea_semaforo(key4,1);
135
136      key5=ftok("file5",'e');
137          s[4] = crea_semaforo(key5,1);
138
139      key6=ftok("file6",'f');
140      sema_mutex=crea_semaforo(key6,1);
```

Imagen 25. Primera parte del main del programa 45.

En la Imagen 26 se puede apreciar como inicia la simulación con un ciclo y básicamente lo que se hace es cambiar de estados, gestionar los procesos, prender y encender los semáforos y comparando el estado en el cual se encuentre al arreglo estado [i].

```
142      while(p<NF)
143      {
144
145          almacena=i;
146          pensar();
147
148          down(sema_mutex);
149          printf("\nTomando el tenedor");
150          estado[i]=HAMBRE;
151
152          //Verificando condiciones de los filosofos
153          if(estado[i]==HAMBRE && estado[LEFT] != COMER && estado[RIGHT] != COMER)
154          {
155                  estado[i]=COMER;
156                  up(s[i]);
157
158          }
159
160
161          //Se termina de probar
162
163          up(sema_mutex); //Se despierta el semaforo
164          down (s[i]);    //Se duerme el semaforo
165
166
167          //termina la funcion tomar tenedor
168          comer();
```

Imagen 26. Segunda parte del main del programa 45.

En la última parte del main, en la Imagen 27, se puede ver la continuación de la simulación de las demás condiciones que se pudieran llegar a representar con los filósofos.

```
        down(sema_mutex);
        estado[i]=PENSAR;
        i=LEFT;

        //Se verifican condiciones
        if(estado[i]==HAMBRE && estado[LEFT]!=COMER && estado[RIGHT] != COMER)
        {
            estado[i]=HAMBRE;
            up(s[i]);

        }
        //Se actualiza el contador i
        i=almacena;
        i=RIGHT;

        //Se verifican condiciones
        if(estado[i]==HAMBRE && estado[LEFT]!=COMER && estado[RIGHT] != COMER)
        {
            estado[i]=HAMBRE;
            up(s[i]);

        }

        //Se despierta el semaforo
        up(sema_mutex);
        printf("\n");
        p++;


    }
```

Imagen 27. Última parte del main.

Como se puede notar, el código es un poco extenso, pero la verdad es practico para resolver el problema en cuestión, se utilizaron semáforos, memoria compartida y procesos.

Para comprobar que el código funciona se procede a compilarlo y posteriormente prepararlo para su ejecución. La Imagen 28 muestra el anterior proceso mencionado, a través de la consola.

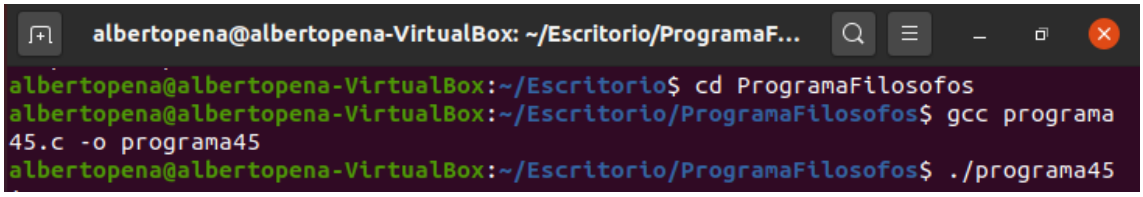

Imagen 28. Compilación del programa de los filósofos.

Después, se ejecutó el programa y se resuelve el problema de los filósofos. A continuación, la Imagen 29 presenta el resultado arrojado.

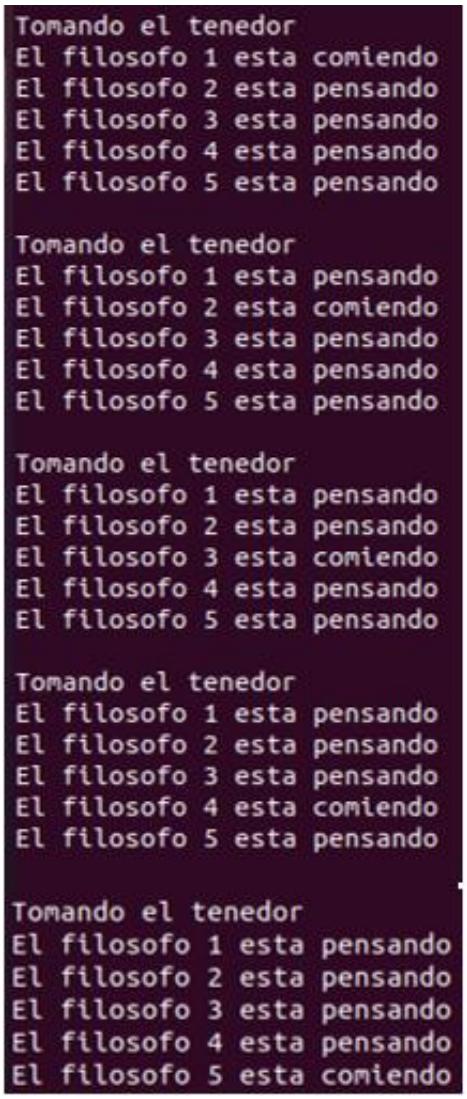

Imagen 29. Ejecución del programa que resuelve el problema de los filósofos.

Cómo se puede notar, se administran de buena forma los procesos y por lo tanto queda resuelto el problema de los filósofos a partir de este algoritmo.

Conclusiones

Martínez Ramírez Sergi Alberto

En esta práctica pudimos observar y quizás comprender el uso y funcionamiento de las herramientas del Inter-Process Cominucation, las cuales son los semáforos y la memoria compartida. En lo personal pude comprender un poco más a fondo la implementación de ambos ya que al momento de realizar los códigos tenía errores a la hora de ejecutarlos. En el primer programa, el programa 41 aunque no tenía errores de compilación, al momento de ejecutar el programa, me saltaba un error que decía "segmentation fault", no sabía a qué se debía ese error, así que lo busqué en internet y descubrí que este error ocurría porque alguna parte del programa ingreseba a un segmento de memoria a la cuál no tenía permiso de entrar. Después de media hora de analizar mi código y como 30 "printf('Hola')" para saber en qué parte ocurría el error, pude darme cuenta que el error saltaba cuando algún hilo quería manipular la variable, esto sucedía ya que no había programado el semáforo para sincronizar a ambos hilos, entonces lo que se me viene a la mente es que mientras un hilo estaba manipulando la variable, el otro hilo comenzó su ejecución y quería manipular la variable al mismo tiempo, y los semáforos internos del sistema operativo restringían el acceso a esa variable para algún otro proceso y por eso saltaba ese error.

En el programa 42 ocurría algo similar, ya que cuando ejecutaba por primera vez el código, todos los números eran "0.00", si lo ejecutaba una segunda vez el primero salía "0.00" y los demás eran números aleatorios, sin embargo, si lo seguía ejecutando repetidas veces, el primer digito siempre resultaba ser "0.00", hasta ahora que estaba haciendo el reporte me di cuenta, que esto sucedía porque el proceso padre imprimía los números antes de que el proceso hijo terminará, esto se resume en que la impresión de la variable la coloqué antes de la llamada al sistema "wait()" (Véase la imagen 5). Logré corregir estos errores simplemente cambiando el orden del código (Véase la imagen 31).

Meza Vargas Brandon David

Al realizar esta práctica algunos conceptos de la comunicación entre procesos que me habían confundido se resolvieron, pues con los distintos programas se vio de una mejor manera como es que sucede esta comunicación.

Como vimos, tenemos distintas herramientas para la comunicación de procesos como lo son las colas de mensajes, la memoria compartida y los semáforos. Estos los pudimos implementar en los distintos programas realizados en esta práctica. Con estos programas podemos ver que podemos hacer varias cosas al comunicar procesos, como compartir valores de ciertas variables y realizar operaciones sobre dichas variables. Al igual que podemos modificar archivos con la comunicación de procesos, también mandar mensajes entre distintos procesos a partir de colas de mensajes como se hizo en el programa 43.

Algo importante es que podemos comunicar procesos emparentados, pero sobre todo procesos que no están emparentados, de esta forma la comunicación de procesos tiene mayor utilidad. Al igual con la realización de los distintos programas me di cuenta de que es posible usar más de una herramienta del IPC, por ejemplo, en el programa 45 se hizo uso de memoria compartida y semáforos para poder resolver el problema de los filósofos.

Es importante mencionar que gracias a estas herramientas que nos proporciona el sistema operativo, podemos solucionar problemas de regiones críticas y de sincronización entre programas como fue el caso del programa 44 y 45.

Peña Atanasio Alberto

El Inter Process Comunication (IPC) permite que los distintos procesos se comuniquen y se sincronizan con otros y esto es realmente importante para una correcta administración de los recursos del Sistema Operativo.

También se aplicaron los conocimientos adquiridos en los videos, principalmente los de memoria compartida y semáforos, estos últimos son importantes debido a que evitan que los procesos se estorben entre sí y que entren en regiones críticas. Con los programas se pudo comprender, más precisamente, el uso de las variables globales y el bloqueo de las áreas críticas correspondientes. Los semáforos gestión la entrada de memoria compartida, de igual forma.

Se pudo comprender de donde viene la analogía de los semáforos. En la vida real, simplemente un semáforo da ciertas indicaciones para guiar a los automóviles y que estos no choquen, en la informática pasa exactamente lo mismo, los semáforos informáticos guían y administran a los procesos para que no se provoque un caos.

Fue una práctica un tanto complicada, debido al nuevo conocimiento que se tuvo que adquirir en cuanto a procesos y nuevas llamadas al sistema jamás se habían manejado, de manera personal, en alguna otra materia de programación, de manera explícita, claramente.

Sarmiento Gutiérrez Juan Carlos

Esta práctica fue muy educativa ya que en ella entendí mucho mejor el cómo se comunican los procesos del sistema operativo, que para este caso necesitan manejar sus regiones Crítica y No Critica lo cual al principio puede ser confuso de entender ya que mientras uno está en esta otro no puede está en la misma región sin embargo en otros algoritmos sí que lo pueden estar, aunque según los videos y programas lo mejor es que estén en buena comunicación con una prefase de los semáforos tal es el caso del algoritmo dekker, sin duda una muy buena práctica.

Bibliografía

[1] W. Stallings, Sistemas operativos, aspectos internos y principios de diseño. Madrid, España: Pearson Educación, 2005.

[2] D. Córdoba, “IPC: Comunicación entre procesos en Linux”, 2014. [En línea]. Disponible: https://juncotic.com/ipc-comunicacion-entre-procesos-en-nix/

Anexos

Código programa 41

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h> //Librería necesaria para manejar hilos
#include <semaphore.h> //Librería necesaria para manejar semáforos
#include <unistd.h>

sem_t semaforo; //Declaración de la variable "semaforo"
int var = 0; //Declaración de la variable var que manipularan los hilos

void *Hilo1(void *segs); //Declaración de la función del primer hilo
void *Hilo2(void *segs); //Declaración de la función del segundo hilo

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

    pthread_t id_hilo1, id_hilo2; //Declaracion de las variables hilos
    sem_init(&semaforo, 0, 1); //Inicia el semáforo
    int segs = 0; //Declaración de la variable segs

    printf("Escriba el tiempo a esperar: "); //Impresión en pantalla
    scanf("%d", &segs); //Guarda la entrada del usuario en una variable

    pthread_create(&id_hilo1, NULL, Hilo1, &segs); //Creación del primer hilo
    pthread_create(&id_hilo2, NULL, Hilo2, &segs); //Creación del segundo hilo

    pthread_join(id_hilo1, NULL); //El proceso principal espera a la ejecución
    pthread_join(id_hilo2, NULL); //del primer y segundo hilo, despues continúa

    sleep(segs); //El proceso principal se duerme según la entrada del usuario

    printf("Valor de la variable al final %d \n", var); //Impresión en pantalla
                                                        //del valor final de la variable
    return 0; // Fin del programa
}

void *Hilo1(void *segs){ //Inicio de la ejecución del primer hilo

    int i;
    int *seg = (int*)segs; //Parseo de la variable segs
    for(i = 0; i < (*seg); i++){ //for que itera desde 0 a la cantidad dada por el usuario
        sem_wait(&semaforo); //Instrucción que espara una "luz verde del semaforo"
        var++; //Incrementa en uno la variable
        sem_post(&semaforo); //Cambia el valor del semaforo, para indicar que su ejecución
                             //ha finalizado
    }
}

void *Hilo2(void *segs){ //Inicio de la ejecución del segundo hilo

    int i;
    int *seg = (int*)segs;
    for(i = 0; i < (*seg); i++){
        sem_wait(&semaforo);
        var--; //Decrementa en uno la variable
        sem_post(&semaforo);
    }
}
```

Imagen 30. Código del programa 41.

Código programa 42

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <time.h> //Libreria para manipular la fecha y hora del sistema

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

    int shmid, i, salida; //Declaración de la variable de memoria compartida, entre otras
    float *random[10]; //Declaración de la variable que va a ser compartida
    pid_t pid; //Declaración de la variable "pid"
    srand(time(NULL)); //Genera la semilla para la llamada al sistema "rand" en base a la
                       //hora del sistema al momento de la ejecución
    key_t key; //Declaración de la variable "key"

    key = ftok("Memoria", 'k'); //Convierte una ruta y un proyecto a una llave de IPC del sistema
    shmid = shmget(key, (sizeof(float)*10), IPC_CREAT|0600); //Obtiene un segmento de memoria compartida
    *random = (float*)shmat(shmid, 0, 0); //Una variable apuntador, apunta la dirección de memoria
                                          //compratida obtenida anteriormente

    pid = fork(); //Crea un nuevo proceso
    if(pid == -1){ //verifica si el proceso tuvo errores
        perror("Error al crear el proceso"); //En caso de no haberse creado el proceso, manda un error
        exit(-1); //Termina el proceso con mensaje de error
    } else if(pid == 0){ //Verifica que el proceso se haya creado correctamente
        for(i = 0; i < 10; i++){ //for que itera desde el 0 hasta el 9
            (*random)[i] = rand() % 100; //Asigna un número aleatorio a la posición actual
        }                                //de la variable random
        exit(0); //Finaliza el proceso
    }

    pid = wait(&salida); //Espera a que el proceso termine

    for(i = 0; i < 10; i++){ //For que itera desde el 0 hasta el 9
        printf("%0.2f \n", (*random)[i]); //Imprimer la variable compartida
    }

    shmdt(&random); //Desvincula la variable apuntador del segmento de memoria compartida
    shmctl(shmid, IPC_RMID, 0); //Elimina el segmento de memoria compartida

    return 0; //Fin del programa
}
```

Imagen 31. Código del programa 42.

Código programa 43

Programa43p1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h> //permite utilizar la llamada al sistema fork()
#include <sys/types.h> //tipos de datos, permite usar pid_t
#include <wait.h> //define las declaraciones de espera

struct msg{
    long MID; //almacena el id del mensaje
    char ms[40];
    pid_t PID;
};

int main()
{
    key_t llave;//redefinicion de un entero
    int idcmsg; //almacena el id de la cola de mensajes
    struct msg mensaje; //mensaje que se enviará

    llave=ftok("prueba", 'k');//se crea la llave, los procesos cona cceso al archivo y al entero
                              //pueden acceder a los recursos compartidos

    idcmsg=msgget(llave, 0600 | IPC_CREAT);//se crea la cola de mensajes

    mensaje.MID=1;
    mensaje.PID=getpid();
    strcpy(mensaje.ms, "Hola soy el proceso 1");

    msgsnd(idcmsg, (struct msgbuf *)&mensaje, sizeof(mensaje.PID)+sizeof(mensaje.ms), IPC_NOWAIT);//se manda un mensaje
    printf("He enviado el mensaje\n");

    msgrcv(idcmsg, (struct msgbuf *)&mensaje, sizeof(mensaje.PID)+sizeof(mensaje.ms), 2, 0);   //se recibe la cola de mensajes
    printf("He recibido un mensaje:\n");
    printf("%s, mi PID es %d\n", mensaje.ms, mensaje.PID);//imprime el mensaje recibido

    msgctl(idcmsg, IPC_RMID, (struct msqid_ds*)NULL);//se cierra la cola de mensajes
    return 0;
}
```

Programa43p2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/msg.h>
#include <string.h>
#include <sys/types.h> //tipos de datos, permite usar pid_t
#include <unistd.h> //permite utilizar la llamada al sistema fork()
#include <wait.h> //define las declaraciones de espera

struct msg{
    long MID; //almacena el id del mensaje
    char ms[40]; //almacena el mensaje que será enviado
    pid_t PID; //almacena el PID del proceso
};

int main()
{
    key_t llave; //redefinicion de un entero
    int idcmsg; //almacena el id de la cola de mensajes
    struct msg mensaje; //mensaje que se enviará

    llave=ftok("prueba", 'k');//se crea la llave, los procesos cona cceso al archivo y al entero
                              //pueden acceder a los recursos compartidos

    idcmsg=msgget(llave, 0600 | IPC_CREAT); //se crea la cola de mensajes
    msgrcv(idcmsg, (struct msgbuf *)&mensaje, sizeof(mensaje.PID)+sizeof(mensaje.ms), 1, 0); //se recibe la cola de mensajes
    printf("He recibido un mensaje:\n");
    printf("%s, mi PID es %d \n", mensaje.ms, mensaje.PID); //imprime el mensaje recibido
    mensaje.MID=2;
    mensaje.PID=getpid();

    strcpy(mensaje.ms, "Hola soy el proceso 2");
    msgsnd(idcmsg, (struct msgbuf *)&mensaje, sizeof(mensaje.PID)+sizeof(mensaje.ms), IPC_NOWAIT);//se manda un mensaje
    printf("He enviado el mensaje\n");

    return 0;
}
```

Imagen 32. Código programa 43.

Código programa 44

Practica44a.c

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

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);
    }
}

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\n",i);
        sleep(1);
    }
}


int main(){
    int PID;
    int shmid1;
    int shmid2;
    int shmid3;
    int *Hijo_desea_entrar;
    int *Padre_desea_entrar;
    int *Proceso_favorecido;
    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);
     Hijo_desea_entrar=shmat(shmid1,0,0);
     Padre_desea_entrar=shmat(shmid2,0,0);
     Proceso_favorecido=shmat(shmid3,0,0);
     *Hijo_desea_entrar=1;
     *Padre_desea_entrar=0;
     *Proceso_favorecido=1;

     while (1){
         *Hijo_desea_entrar=1;
         while(*Padre_desea_entrar){
             if(*Proceso_favorecido==2){
                 *Hijo_desea_entrar=0;
                 while(*Proceso_favorecido==2);
                     *Hijo_desea_entrar=1;
             }
         }
         regionCritica();
             *Proceso_favorecido=2;
             *Hijo_desea_entrar=0;
             regionNoCritica();
       }

    return 0;
}
```

Imagen 33. Código practica44a.c

Practica44b.c

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

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);
    }
}
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\n",i);
        sleep(1);
    }
}
int main(){
    int PID;
    int shmid1;
    int shmid2;
    int shmid3;
    int *Hijo_desea_entrar;
    int *Padre_desea_entrar;
    int *Proceso_favorecido;
    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);
    Hijo_desea_entrar=shmat(shmid1,0,0);
    Padre_desea_entrar=shmat(shmid2,0,0);
    Proceso_favorecido=shmat(shmid3,0,0);
```

```
33      key_t llave1;
34      key_t llave2;
35      key_t llave3;
36      llave1=ftok("Prueba1",'k');
37      llave2=ftok("Prueba2",'l');
38      llave3=ftok("Prueba3",'m');
39      shmid1=shmget(llave1,sizeof(int),IPC_CREAT|0600);
40      shmid2=shmget(llave2,sizeof(int),IPC_CREAT|0600);
41      shmid3=shmget(llave3,sizeof(int),IPC_CREAT|0600);
42      Hijo_desea_entrar=shmat(shmid1,0,0);
43      Padre_desea_entrar=shmat(shmid2,0,0);
44      Proceso_favorecido=shmat(shmid3,0,0);
45          while(1){
46              *Padre_desea_entrar=1;
47              while(*Hijo_desea_entrar){
48                  if(*Proceso_favorecido==1){
49                      *Padre_desea_entrar=0;
50                      while(*Proceso_favorecido==1);
51                          *Padre_desea_entrar=1;
52                  }
53              }
54              regionCritica();
55              *Proceso_favorecido=1;
56              *Padre_desea_entrar=0;
57              regionNoCritica();
58          }
59      return 0;
60  }
61
```

Imagen 34. Código practica44b.c

Código programa 45

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <time.h>

#define FALSE   0
#define TRUE    1
#define NF      5           //Constante que indica el Numero de filósofos
#define RIGHT   (i+1)%NF    //Constante de la derecha
#define LEFT    (i+NF-1)%NF //Constante de la izquierda
#define COMER       2       //El filosofo está comiendo
#define HAMBRE      1       //Estado hambriento del filosofo
#define PENSAR  0       //Estado pensativo del filosofo


int estado[NF];     //Arreglo donde está el estado del filósofo

void pensar(void);  //Funcion que indica que el filosofo piensa
void comer (void);  //Funcion que indica que el filosofo come
void up(int semid); //Funcion de control del semaforo
void down(int semid);   //Funcion del control de semaforo
int crea_semaforo(key_t key, int valor_ini);


//Funcion que crea el semáforo con su llave y valor inicial, asignado permisos

int crea_semaforo(key_t key, int valor_ini)
{
    int semid;  //ID del semáforo

    semid = semget(key,1,IPC_CREAT | 0644 );

    if(semid==-1)
    {
        printf("\nNo se pudo crear el semaforo");
        exit(-1);

    }

```

```
44      semctl(semid,0,SETVAL,valor_ini);
45
46  }
47
48
49  void down(int semid)
50  {
51      struct sembuf op_v[]={0,-1,0};
52      semop(semid,op_v,1);
53
54
55  }
56
57
58  void up(int semid)
59  {
60      struct sembuf op_p[]={0,+1,0};
61      semop(semid,op_p,1);
62
63  }
64
65
66
67
68  //Función que simula que un filosofo está comiendo o está pensando
69  void comer (void)
70  {
71      int j;
72
73      for(j=0;j<NF;j++)
74      {
75          if(estado[j]==COMER)
76          {
77              printf("\nEl filosofo %d esta comiendo",j+1);
78
79          }
80          else if(estado[j]==PENSAR)
81          {
82              printf("\nEl filosofo %d esta pensando",j+1);
83          }
84
```

```
    }


}



//Función que ayuda a saber si el filosofo está pensando o comiendo
void pensar(void)
{
    int j;
    for(j=0;j<NF;j++)
    {
        if(estado[j] == PENSAR)
        {
            //Continua
        }
        else if(estado[j]==COMER)
        {
            printf("\nEl filosofo %d esta comiendo",j+1);

        }


    }


}


void main()
{
    int s[NF],i=0, almacena=0,no_filosofo=5,sema_mutex=0;
    key_t key1=0,key2=0,key3=0,key4=0,key5=0,key6=0;

    printf("1");

    //Creando los semáforos de los filósofos
    //Creando las llaves e invocando a la funcion
```

```
    key1=ftok("file1",'a');
    s[0] = crea_semaforo(key1,1);

    key2=ftok("file2",'b');
    s[1] = crea_semaforo(key2,1);

    key3=ftok("file3",'c');
    s[2] = crea_semaforo(key3,1);

    key4=ftok("file4",'d');
    s[3] = crea_semaforo(key4,1);

    key5=ftok("file5",'e');
        s[4] = crea_semaforo(key5,1);

    key6=ftok("file6",'f');
    sema_mutex=crea_semaforo(key6,1);
    int p=0;
    while(p<NF)
    {

        almacena=i;
        pensar();

        down(sema_mutex);
        printf("\nTomando el tenedor");
        estado[i]=HAMBRE;

        //Verificando condiciones de los filosofos
        if(estado[i]==HAMBRE && estado[LEFT] != COMER && estado[RIGHT] != COMER)
        {
                estado[i]=COMER;
                up(s[i]);

        }


        //Se termina de probar

        up(sema_mutex); //Se despierta el semaforo
        down (s[i]);    //Se duerme el semaforo
```

```
165
166
167         //termina la funcion tomar tenedor
168         comer();
169
170
171         down(sema_mutex);
172         estado[i]=PENSAR;
173         i=LEFT;
174
175         //Se verifican condiciones
176         if(estado[i]==HAMBRE && estado[LEFT]!=COMER && estado[RIGHT] != COMER)
177         {
178             estado[i]=HAMBRE;
179             up(s[i]);
180
181         }
182         //Se actualiza el contador i
183         i=almacena;
184         i=RIGHT;
185
186         //Se verifican condiciones
187         if(estado[i]==HAMBRE && estado[LEFT]!=COMER && estado[RIGHT] != COMER)
188         {
189             estado[i]=HAMBRE;
190             up(s[i]);
191
192         }
193
194         //Se despierta el semaforo
195         up(sema_mutex);
196         printf("\n");
197         p++;
198
199
200     }
201
202
203 }
```

Imagen 35. Código del programa 45.