1. Memoria compartida

La forma más rápida de comunicar dos procesos es hacer que compartan una zona de memoria. Para enviar datos de un proceso a otro, sólo hay que escribir en memoria y automáticamente esos datos están disponibles para que los lea cualquier otro proceso.

La memoria convencional que puede direccionar un proceso a través de sus espacio de direcciones virtuales es local a ese proceso y cualquier intento de direccionar esa memoria desde otro proceso va a provocar una violación de segmento.

Para solucionar este problema, UNIX System V brinda la posibilidad de crear zonas de memoria con la característica de poder ser direccionadas por varios procesos simultáneamente. Esta memoria va a ser virtual, por lo que sus direcciones físicas asociadas podrán variar con el tiempo. Esto no va a plantear ningún problema, ya que los procesos no generan direcciones físicas, sino virtuales, y es el núcleo el encargado de traducir de unas a otras.

Las llamadas para poder manipular la memoria compartida son: *shmget*, para crear una zona de memoria compartida o habilitar el acceso a una ya creada; *shmctl*, para acceder y modificar la información administrativa y de control que el núcleo le asocia a cada zona de memoria compartida; *shmat*, para unir una zona de memoria compartida a un proceso, y *shmdt*, para separar una zona previamente unida.

1.1. Petición de memoria compartida, *shmget*

Mediante *shmget* vamos a obtener un identificador con el que podemos realizar futuras llamadas al sistema para controlar una zona de memoria compartida. Su declaración es:

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

int shmget (key_t key, int size, int shmflg);
```

key es una llave que tiene el mismo significado que vimos para la llamada *semget* (creación de semáforos). Esta es una característica que tienen todas las llamadas para crear mecanismos IPC.

size es el tamaño en bytes de la zona de memoria que queremos crear.

shmflg es una máscara de bits que tiene el mismo significado que vimos para la máscara *semflg* de la llamada *semget*.

Si la llamada se ejecuta correctamente, devolverá el identificador (número entero no negativo) asociado a la zona de memoria. Si falla, devolverá el valor -1 y en *errno* estará el código del tipo de error producido.

El identificador devuelto por *shmget* es heredado por los procesos descendientes del proceso actual.

Las siguientes líneas muestran cómo crear una zona de memoria de tamaño 4,096 bytes, donde sólo el usuario va a tener permisos de lectura y escritura:

```
int shmid;
...
if((shmid = shmget(IPC_PRIVATE, 4096, IPC_CREAT|0600))==-1){
  /*Error en la creación de la memoria compartida.
    Tratamiento del error. */
}
```

1.2. Control de una zona de memoria compartida, *shmctl*

Con *shmctl* podemos realizar operaciones de control sobre una zona de memoria previamente creada por una llamada a *shmget*. Su declaración es:

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

shmid es un identificador válido devuelto por una llamada previa a *shmget*.

cmd indica el tipo de operación de control a realizar. Sus posibles valores son:

IPC_STAT Lee el estado de la estructura de control de la memoria y lo devuelve a través de la zona de memoria apuntada por *buf*.

IPC_SET Inicializa algunos de los campos de la estructura de control de la memoria compartida. El valor de estos campos los toma de la estructura apuntada por *buf*.

IPC_RMID Borra del sistema la zona de memoria compartida identificada por *shmid*. Si el segmento de memoria está unido a varios procesos, el borrado no se hace efectivo hasta que todos los procesos liberen la memoria.

SHM_LOCK Bloquea en memoria el segmento identificado por *shmid*. Esto quiere decir que no se va a realizar intercambio sobre él. Sólo los procesos cuyo identificador de usuario efectivo (EUID) sea igual al del superusuario van a poder realizar esta operación.

SHM_UNLOCK Desbloquea el segmento de memoria compartida, con lo que los mecanismos de intercambio van a poder trasladarlo de la memoria principal a la secundaria, y viceversa, cada vez que sea necesario. Sólo los procesos cuyo identificador de usuario efectivo (EUID) sea igual al del superusuario van a poder realizar esta operación.

La estructura *shmid_ds* se define como sigue:

```
struct shmid_ds{

  struct ipc_per shm_per;     /* Estructura de permisos. */
  int            sh_segsz:    /* Tamaño del área de memoria compartida. */
  int            pad1;        /* Usado por el sistema. */
  unshort        shm_lpid;    /* PID del proceso que hizo la última
                                 operación con este segmento de memoria. */
  unshort        shm_cpid;    /* PID del proceso creador del segmento. */
  unshort        shm_nattach; /* Número de procesos unidos al segmento
                                 de memoria. */
  short          pad2;        /* Usado por el sistema. */
  time_t         shm_atime;   /* Fecha de la última unión al segmento
                                 de memoria. */
  time_t         shm_dtime    /* Fecha de la última separación del
                                 segmento de memoria. */
  time_t         shm_ctime;   /* Fecha del último cambio en el segmento
                                 de memoria. */
}
```

La siguiente línea, muestra cómo borrar del sistema, una zona de memoria compartida:

```
shmctl(shmid, IPC_RMID,0);
```

1.3. Operaciones con la memoria compartida, *shmat* y *shmdt*

Antes de usar una zona de memoria compartida, tenemos que asignarle un espacio de direcciones virtuales de nuestro proceso. Esto es lo que se conoce como unirse o atarse al segmento de memoria compartida. Una vez que dejamos de usar un segmento de memoria, tenemos que desatarnos del él. Al realizar esta operación, el segmento deja de estar accesible para el proceso.

Las llamadas al sistema para realizar estas operaciones con *shmat* (para atar) y *shmdt* (para desatar), y sus declaraciones son:

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

char *shmat(int shmid, char *shmaddr, int shmflg);
int shmdt(char  *shmaddr);
```

shmid es el identificador de una zona de memoria creada mediante una llamada previa a *shmget*.

shmaddr es la dirección virtual donde queremos que empiece la zona de memoria compartida. Si la llamada a *shmat* funciona correctamente, devolverá un puntero a la dirección virtual a la que está unido el segmento de memoria compartida. Esta dirección puede coincidir o no con

shmaddr, dependiendo de la decisión que tome el núcleo. Lo normal es que *shmaddr* valga 0, con lo cual se deja en manos del núcleo la elección de la dirección de inicio. Si *shmaddr* no vale 0, el núcleo intentará satisfacer la petición del usuario, pero no siempre se consigue. En el caso de *shmdt*, *shmaddr* es la dirección virtual del segmento de memoria compartida que queremos separar del proceso.

shmflg es una máscara de bits que indica la forma de acceso a la memoria. Si el bit SHM_RDONLY está activo, la memoria será accesible para leer, pero no para escribir.

Si las llamadas funcionan correctamente, *shmat* devuelve la dirección a la que está unido el segmento de memoria compartida y *shmdt* devuelve 0. Si algo falla, ambas llamadas devuelven -1 y en *error* estará el código del tipo de error producido.

Una vez que la memoria está unidad al espacio de direcciones virtuales del proceso, el acceso a ella se realiza a través de punteros, como con cualquier otra memoria de datos asignada al programa.

A continuación mostramos la forma de crear una zona de memoria compartida en la que se va a almacenar un arreglo unidireccional de números reales:

```
#define MAX 10

int shmid, i;
float *array;
key_t llave;
...

/* Creación de una llave. */
llave = ftok("prueba", 'k');

/* Petición de una zona de memoria compartida. */
shmid = shmget(llave, MAX*sizeof(float), IPC_CREAT|0600);

/* Unión de la zona de memoria compartida a nuestro espacio de
   direcciones virtuales. */
array = shmat(shmid,0,0);

/* Manipulación de la zona de memoria compartida. */
for(i=0; i<MAX; i++)
   array[i]= i*i;
...

/* Separación de la zona de memoria compartida de nuestro
   espacio de direcciones virtuales. */
shmdt(array);

/* Borrado de la zona de memoria compartida. */
shmctl(shmid, IPC_RMID,0);
```

1.4. Ejemplo - Multiplicación en paralelo de matrices

Como ejemplo, vamos a ver la implementación de un algoritmo para multiplicar dos matrices en paralelo.

El programa principal se va a encargar de leer dos matrices y comprobar si se pueden multiplicar. Acto seguido van a arrancar tantos procesos como le hayamos indicado en la línea de órdenes para multiplicar las matrices. Cada proceso se va a ocupar de generar una fila de la matriz producto mientras queden filas por generar. Naturalmente, las matrices que intervienen en la operación deben estar en memoria compartida. Para controlar la fila que debe generar cada proceso, vamos a utilizar un semáforo que se inicializa con el total de filas de la matriz producto y se va decrementando por cada fila generada. El proceso principal se queda esperando a que todos los demás terminen para presentar el resultado.

Aunque en sistemas monoprocesador este método no resulta eficiente, es un buen ejercicio de sincronismo y comunicación entre procesos.

El código de este programa es el siguiente:

```
1  /***
   * PROGRAMA: matrices.c
3  * DESCRIPCIÓN: Programa para multiplicar matrices en paralelo.
   * FORMA DE USO: matrices n_de_procesos
5  ***/

7  #include <stdio.h>
   #include <stdlib.h>
9  #include <sys/types.h>
   #include <sys/ipc.h>
11 #include <sys/sem.h>
   #include <sys/shm.h>
13 #include <unistd.h>
   #include <sys/wait.h>
15
   /* Definición de matriz. */
17
   typedef struct{
19   int shmid; /* Identificador de la zona de memoria compartida
                 donde va a estar la matriz. */
21   int filas;
     int columnas;
23   float **coef; /* Coeficientes de la matriz. */
   }matriz;
25
   /***
27 * FUNCIÓN: Crear_matriz
   * DESCRIPCIÓN: Función que crea la memoria compartida donde va
29 *              a esta la matriz.
   ***/
31
   matriz *crear_matriz(int filas, int columnas){
33 int shmid;
   int i;
35 matriz *m;
```

```
37 /* Petición de memoria compartida. */
   shmid = shmget (IPC_PRIVATE,
39             sizeof (matriz) + filas*sizeof(float *) +
          filas*columnas*sizeof(float),
41         IPC_CREAT | 0600);

43 if(shmid == -1){
     perror("crear_matriz(shmget)");
45   exit(-1);
   }
47
   /* Nos atamos a la memoria. */
49
   if((m = (matriz *) shmat(shmid,0,0)) == (matriz *)-1){
51   perror("crear_matriz(shmdt)");
     exit(-1);
53 }

55 /* Inicialización de la matriz. */

57 m->shmid = shmid;
   m->filas = filas;
59 m->columnas = columnas;

61 /* Le damos formato a la memoria para poder direccionar los
      coeficientes de la matriz. */
63
   m->coef = (float **) & m->coef + sizeof(float **);
65
   for(i = 0; i < filas; i++)
67   m->coef[i] = (float *) & m->coef[filas] +
                  i*columnas*sizeof(float);
69 return m;
   }
71
   /***
73  * FUNCIÓN: Leer_matriz
    * DESCRIPCIÓN: Lee una matriz de un arreglo.
75  ***/

77 matriz *leer_matriz(int filas, int columnas, float* mm){
     int i;
79   int j;
     matriz *m;
81
       m = crear_matriz(filas,columnas);
83     for(i = 0; i < filas; i++)
         for(j = 0; j < columnas; j++){
85          m->coef[i][j] = *mm;
       mm++;
87       }
       return m;
89 }
```

```
91 /***
   * FUNCIÓN: multiplicar_matriz
93 * DESCRIPCIÓN: Multiplica dos matrices y crea una nueva para el
   *              resultado. El trabajo se distribuye entre el total de
95 *              procesos que indique "numproc".
   ***/
97
matriz *multiplicar_matrices(matriz *a, matriz *b, int numproc){
99
    int p;
101 int semid;
    int estado;
103 matriz *c;

105  if (a->columnas != b->filas)
        return NULL;
107
     c = crear_matriz(a->filas , b->columnas);
109
     /* Creación de dos semáforos. Uno de ellos se inicializa con
111     el total de filas de la matriz producto. */

113  semid = semget(IPC_PRIVATE, 2, IPC_CREAT | 0600);
     if(semid == -1){
115     perror("multiplicar_matrices(semget)");
  exit(-1);
117  }

119  semctl(semid, 0, SETVAL,1);
     semctl(semid, 1, SETVAL, c->filas +1);
121
     /* Creación de tantos procesos como indique "numproc". */
123  for(p = 0; p < numproc; p++){

125      if(fork() == 0){
         /*Código para los procesos hijo. */
127      int i;
         int j;
129      int k;
         struct sembuf operacion;
131
         operacion.sem_flg == SEM_UNDO;
133
         while(1){
135       /*Cada proceso hijo se encarga de generar una fila
            de la matriz producto. Para saber que columna
137         tiene que generar, consulta el valor del semaforo. */

139       /* Operación "P" sobre el semaforo "0". */

141       operacion.sem_num = 0;
          operacion.sem_op = -1;
143       semop(semid, &operacion,1);
```

```
145           /* Concultamos el valor del semaforo. */

147            i = semctl(semid, 1,GETVAL,0);

149            if(i > 0){
                 /* Decremenmtamos el valor del semaforo "1" en una unidad. */
151             semctl(semid, 1, SETVAL, --i);
                /* Operación "V"  sobre el semaforo "0". */
153             operacion.sem_num = 0;
                operacion.sem_op = 1;
155             semop(semid, &operacion,1);
               } else
157                exit(0);

159            /* Calculo de la fila i-ésima de la matriz producto. */
               for(j = 0; j < c->columnas; j++){
161               c->coef[i][j] = 0;
                  for(k = 0; k < a->columnas; k++)
163                  c->coef[i][j] += a->coef[i][k]* b->coef[k][j];
               }
165          } /* while */
           } /* if */
167      } /* for */

169      /* Esperamos a que termien todos los procesos. */
         for(p = 0; p < numproc; p++)
171          wait(&estado);

173      /* Borramos el semáforo. */
         semctl(semid, 0, IPC_RMID, 0);
175      return c;
}
177
/***
179  * FUNCIÓN: destruir_matriz
     * DESCRIPCIÓN: Función encargada de liberar una zona de memoria
181  *              compartida.
     ***/
183

185 void destruir_matriz(matriz *m){
      shmctl(m->shmid, IPC_RMID,0);
187 }

189 /***
     * FUNCIÓN: imprimir_matriz
191  * DESCRIPCIÓN: Esta función presenta una matriz en el fichero
     *              estandar de salida.
193  ***/

195 void imprimir_matriz(matriz *m){
      int i;
197   int j;
```

```
199   for(i = 0; i < m->filas; i++){
        for(j = 0; j < m->columnas; j++)
201       printf("%g ", m->coef[i][j]);
          printf("\n");
203     }
}
205
/****
207  * Función principal.
 ****/
209
int main (int argc, char *argv[]){
211
  int numproc;
213   matriz *a;
  matriz *b;
215   matriz *c;

217         float MA[3][4] = { 0 ,1 ,1,1,
                              6, -1, 2, 0,
219                           3, -2, -1, 6};

221         float MB[4][4] = { 1 ,2 ,3, 6,
                              6, -1, 2, 0,
223                           3, -2, -1, 6,
                              6, 2, 0, 2};
225
  /* Análisis de los parámetros de la línea de ordenes. */
227   if(argc != 2)
      numproc = 2;
229   else
      numproc = atoi(argv[1]) + 1;
231
  /* Lectura de las matrices. */
233   a = leer_matriz(3, 4, (float *)&MA);
  b = leer_matriz(4, 4, (float *)&MB);
235
  /* Procesamiento  de las matrices. */
237   c = multiplicar_matrices(a,b,numproc);

239   if (c != NULL)
        imprimir_matriz(c);
241   else
        fprintf(stderr, "Las matrices no se pueden multiplicar.");
243
  destruir_matriz(a);
245   destruir_matriz(b);
  destruir_matriz(c);
247   return 0;
}
```

Código 1: Multiplicación de una matriz en paralelo.

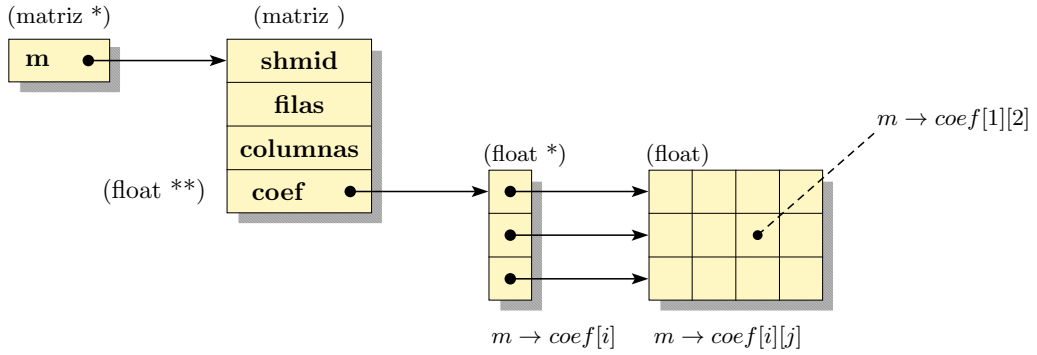
Es importante aclarar la forma de organizar la memoria asignada a una matriz, ya que puede haber quedado algo oscura en el código.

Recordemos que una matriz se define mediante la estructura:

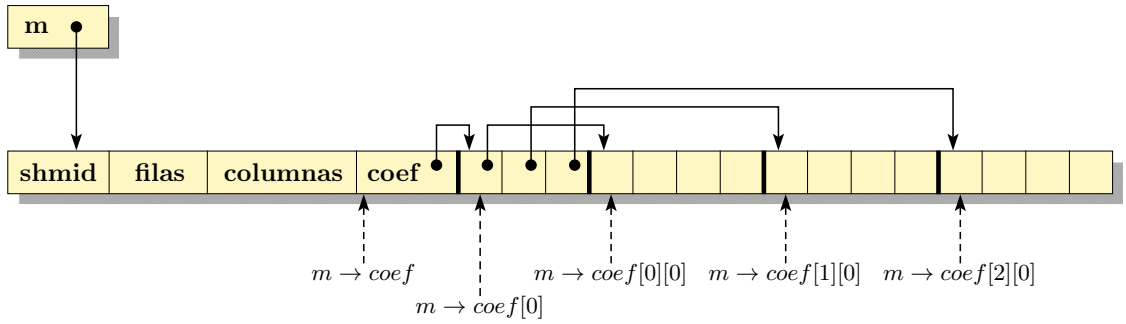
```
typedef struct{
   int shmid;
   int filas;
   int columnas;
   float **coef;
}matriz;
```

Las matrices se almacenan en zonas de memoria compartida para que puedan ser manejadas por varios procesos. Estas zonas se crean mediante una llamada a *shmget*, y mediante *shmat* podemos unirlas a un puntero del programa. La memoria creada por *shmget* sólo cumple el requisito de estar alineada, por lo que tenemos que darle formato de matriz.

Los campos *shmid*, *filas* y *columnas* no plantean ningún problema, ya que son accesibles a través de un puntero del tipo *matriz*. El problema lo van a plantear los coeficientes de la matriz, que se van a almacenar en el campo *coef*. Este campo es un doble puntero y para que pueda utilizarse para indexar una matriz, debe responder a una organización como la reflejada en la figura 1(a). Esta es una representación gráfica para comprender el sentido de la doble indirección, pero la verdadera estructura de la memoria se muestra en la figura 1(b).



a) Representación gráfica del uso de la doble indirección para indexar matrices.



b) Estructura real de una matriz m de 3×4.

Figura 1: Representación gráfica del tipo matriz.