# ÍNDICE

# 1. PROGRAMACIÓN DISTRUIBUIDA UTILIZANDO MEMORIA COMPARTIDA

## 1.1. Conceptos de memoria compartida

Una parte de los sistemas de gestión de memoria tanto software (sistema operativo UNIX) como hardware está enfocada a asegurar que la memoria perteneciente a un proceso no sea accedida por otro proceso sin permiso. En sistemas multi-usuario este tipo de intromisiones puede ir desde la interferencia de un proceso en la operación de otros hasta, peor aún, la caída del sistema debido a la sobreescritura de partes del kernel.

Sin embargo la compartición de memoria puede ser una forma efectiva de comunicación entre procesos cooperativos de una aplicación distribuida. Consecuentemente la mayoría de versiones de UNIX soportan el mecanismo IPC denominado **memoria compartida**.

La idea básica se ilustra en la Figura 1: Un proceso (por ejemplo el A) crea inicialmente un segmento de memoria compartida en el heap. El proceso A deberá indica el tamaño del segmento así como asociarle una clave. Una vez está creado el segmento compartido si un proceso (A o B) desea hacer uso de esa memoria deberá de vincular (attach) el segmento a su espacio de direcciones. La operación de vinculación (attach) devuelve un puntero igual que lo haría la función conocida malloc, con el que se podrá acceder al segmento compartido. Por supuesto que para acceder al segmento compartido el proceso interesado deberá conocer la clave de segmento compartido. Evidentemente los cambios en el segmento desde cualquier proceso son visibles en el resto de procesos que tienen vinculado dicho segmento compartido.

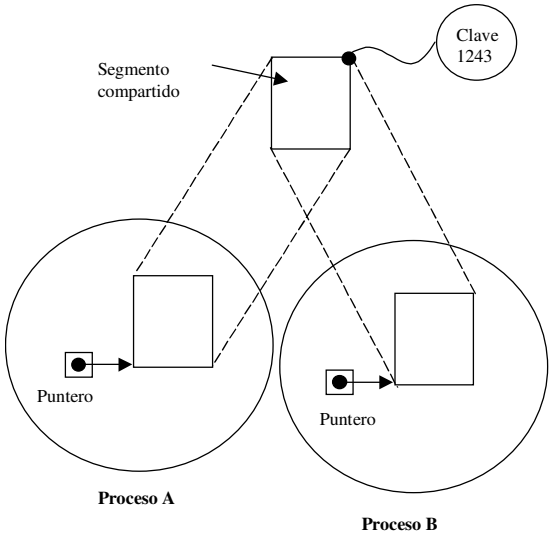


**Figura 1. Ilustración de memoria compartida**

Existe una similitud entre la memoria compartida y los archivos de UNIX. Un segmento de memoria compartida no pertenece a ningún proceso, puede incluso permanecer en memoria aún cuando ningún proceso lo tiene vinculado. Un segmento de memoria compartida también presenta permisos de acceso y hora y fecha de modificación. Con el comando **ipcs** se puede obtener un listado de los segmentos de memoria compartida, como el comando **ls** para archivos.

## 1.2. Operaciones con memoria compartida

Las llamadas al sistema empleadas para operar con memoria compartida son **shmget()** para crear un nuevo segmento de memoria compartida u obtener un manejador a un segmento existente. En esta llamada el segmento se identifica con su clave. La llamada **shmat()** se empleará para vincular un segmento al espacio de direcciones del proceso que invoca la llamada.

```
id = shmget (key, size, flag)

donde:
  id: Manejador para el segmento.
  key: Clave numérica identificativa del segmento.
  size: Tamaño del segmento en bytes.
  flag: Típicamente se usa flag = IPC_CREAT | 0644 para crear un nuevo segmento con acceso
rw-r--r--, flag = 0  para obtener un segmento existente.

ptr = shmat(id, addr, flag)

donde:
  ptr: Puntero para acceder al segmento de memoria compartida desde el proceso que haya
ejecutado shmat.
  id: Manejador para el segmento
  addr: Especifica una dirección en la que mapear el segmento de memoria compartida.
Típicamente el valor 0, se emplea para el que el sistema elija la dirección de mapeo.
  flag: Típicamente 0. Se puede emplear SHM_RDONLY para vincular en solo lectura.
```

## 1.3. Un ejemplo de memoria compartida

Se emplearán los programas cline y pline para presentar un problema de memoria compartida.

El programa pline (print line) imprime una línea de caracteres cada 4 segundos. La longitud de la línea así como el carácter impreso, se obtendrán de una pequeña estructura de tipo **struct info** contenida en el segmento de memoria compartida. La idea es que otro programa pueda acceder al segmento y modificar los datos de la estructura **struct info**.

El programa cline (change line) toma dos argumentos: un carácter y un contador. Esto datos serán escritos en el segmento compartido sobre la estructura info, donde serán vistos por pline, la próxima vez que imprima una línea.

Ambos programas deben conocer la clave del segmento y el tipo de la estructura info. Como esta información será compartida, la situaremos en un fichero cabecera:

```
/* line.h */
struct info {
  char c;
  int length;
};
#define KEY     ( (key_t) (1234))
#define SEGSIZE sizeof(struct info)
```

La elección de la clave 1234 es siempre arbitraria. El único requerimiento es que no se solape con el de otras aplicaciones de la misma máquina.

El programa pline crea un segmento si es posible, lo vincula a su espacio de direccionamiento y escribe los valores por defecto en la estructura info. A continuación procede escribir una línea de texto cada 4 segundos de acorde a los campos de la estructura **info** almacenada en el segmento compartido. Esto bucle finalizará cuando el contador de **info** sea 0.

```
/* pline.c
Este es un ejemplo de uso de memoria compartida. El programa imprime una línea de texto
cada cuatro segundos. La longitud de la línea, y el carácter impreso se controlan mediante
dos valores incluidos en una pequeña estructura de datos en un segmento de memoria
compartida. Cualquier otro programa puede vincular dicho segmento y modificar los campos de
la estructura de datos.
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include “line.h”

main()
{
        int i, id;
        struct info *ctrl;
        struct shmid_ds shmbuf;

        id = shmget(KEY, SEGSIZE, IPC_CREAT | 0666);
        if (id < 0) {
          perror("pline: shmget failed:");
          exit(1);
        }

        ctrl = (struct info *)shmat(id, 0, 0);
        if (ctrl <= (struct info *)(0)) {
                perror("pline: shmat failed:");
                exit(2);
        }
        /*Inicia los parametros por defecto */
        ctrl->c         = 'a';
        ctrl->length    = 10;

        /*Este bucle imprime una linea cada 4 segundos */
        while (ctrl->length > 0) {
          for(i = 0; i < ctrl->length; i++)
            putchar(ctrl->c);
            putchar('\n');
            sleep(4);
        }
        exit(0);
}
```

En el caso del programa cline, se asume que el segmento compartido ya existe, y fallará si no existe. Una vez se ha obtenido una manejador para el segmento compartido y vinculado, el programa cline toma un carácter y un contador de la línea de comandos y los escribe en el segmento compartido.

```
/* cline.c
Este es un ejemplo de uso de memoria compartida. El programa opera en conjunto con el
programa pline. Este programa permite modificar la longitud de la línea y el carácter
escribiendo en un segmento de memoria compartido.
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include “line.h”

main(int argc, char *argv[])
{
        int i, id;
        struct info *ctrl;
        struct shmid_ds shmbuf;

        if (argc!= 3) {
                fprintf(stderr, "uso: cline <char> <length>\n");
                exit(3);
        }
        id = shmget(KEY, SEGSIZE, 0);
        if (id < 0) {
                perror("cline: shmget faild:");
                exit(1);
        }
        ctrl = (struct info *)shmat(id, 0, 0);
        if (ctrl <= (struct info *)(0)) {
                perror("cline: shmat failed:");
                exit(2);
        }

        /*Copia los datos de la linea de comandos en el segmento de memoria compartida */
        ctrl->c                 =  argv[1][0];
        ctrl->length   = atoi(argv[2]);

        exit(0);
}
```

Veamos un ejemplo si ejecutamos el programa pline y a continuación ejecutamos sucesivas veces cline con ciertos parámetros:

```
$ pline &                               #Lanzamos pline en background
[1] 29799                               #El shell devuelve el ID del proceso
aaaaaaaaaa
$ aaaaaaaaaa
aaaaaaaaaa                              #pline está ocupado imprimiendo las 10 a’s
aaaaaaaaaa
cline x 20                              #Modificamos los valores del segmento
$ xxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxx                     #Ahora pline imprime 20 x’s
xxxxxxxxxxxxxxxxxxx
cline f 0                               #Se pone el contador a 0
$                                       #Ahora pline finalizará
[2]  Done                     pline     #El shell notifica la finalización
```

Una vez cline y pline finalizan, el segmento compartido permanecerá, y podrá verse con el comando **ipcs**. La clave 0x4d2 mostrada abajo es el valor hexadecimal correspondiente a nuestra clave 1234, definida en line.h.

```
bash$ ipcs -bm

Shared Memory:
T       ID        KEY    MODE          OWNER    GROUP      SEGSZ
m        0 0x3253bc5c --rw-rw-rw-       root   system        648
m        1  0x1b5ed92 --rw-------       root   system   21291008
m        2   0x280267 --rw-r--r--       root   system    1048576
m      131      0x4d2 --rw-rw-rw-   delacal   profes          8
```

Por último, podríamos eliminar el segmento mediante el comando **ipcrm** y a continuación comprobar que el segmento ya no está:

```
bash$ ipcrm -m 131
bash$ ipcs -bm

Shared Memory:
T       ID        KEY    MODE          OWNER    GROUP      SEGSZ
m        0 0x3253bc5c --rw-rw-rw-       root   system        648
m        1  0x1b5ed92 --rw-------       root   system   21291008
m        2   0x280267 --rw-r--r--       root   system    1048576
```

## 1.4. Características de la memoria compartida

Puede parecer que el mecanismo de memoria compartida es excesivamente complejo de emplear, aunque no es tan diferente del uso que hacemos de un archivo. Por ejemplo, los permisos de control de acceso deben existir ya que cualquier proceso que conozca la clave puede acceder a un segmento compartido,…

Si comparamos la memoria compartida con los pipes como mecanismo de comunicación, podemos ver que la eficiencia y el acceso aleatorio son las principales ventajas de la memoria compartida frente a los pipes. En cuanto a la eficiencia, es debido a que:

- Los datos no necesitan ser copiados.
- Un pipe es un buffer en el espacio de direcciones del kernel, por lo que los datos del proceso que da entrada al pipe deben ser copiados al espacio del kernel y otra vez del kernel al espacio de direcciones del proceso que da salida al pipe.

En cuanto al acceso aleatorio, está claro ya que un pipe es un dispositivo de acceso FIFO y por lo tanto secuencial.

Además, la memoria compartida provee de un mecanismo de comunicación muchos a muchos (es decir muchos procesos puede vincularse a un mismo segmento), y no existe una relación clara productor-consumidor. Por otro lado los pipes sí poseen esta relación productor-consumidor de forma nítida.

Una de las ventajas de los pipes es que automáticamente imponen una ligera sincronización entre los procesos de entrada y salida del pipe. La memoria compartida no impone ningún tipo de sincronización. Por ejemplo si queremos implementar un proceso consumidor que se bloquee hasta que aparezcan nuevos datos en el segmento compartido, se deberá emplear algún tipo mecanismo para lograr esa sincronización, semáforos, señales o incluso un pipe.

Como la memoria compartida provee de acceso aleatorio, podría intentar emplearse para compartir grandes estructuras no lineales del estilo de un grafo. Este tipo de estructuras pueden dar muchos

problemas si se emplean listas enlazadas implementadas con punteros, en C por ejemplo. Y esto es debido a lo siguiente:

*Los punteros creados por un proceso sólo son válidos en el espacio de direcciones de dicho proceso. Estos punteros no son válidos en el espacio de direcciones de otro proceso a menos que tal proceso pueda garantizar la vinculación del segmento de memoria compartida a la misma dirección que el primer proceso.*

La posible solución pasaría por el uso de punteros base. Esto es, punteros relativos a una dirección de memoria base (en nuestro caso la dirección a la que ha sido vinculado el segmento de memoria compartida). Desafortunadamente C, no contempla los punteros base aunque puede simularse por aritmética de punteros.

## 1.5. Resumen de llamadas al sistema relacionadas con el mecanismo memoria compartida

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

int shmget(key_t key, size_t size, int flags);
```

Esta rutina crea u obtiene si ya existe el ID de segmento de memoria compartida con clave **key**, de tamaño **size** y con los permisos **flags**.

Donde:

- **Key**: Representa la clave compartida para acceder al segmento de memoria compartida desde cualquier proceso. Debe ser un valor distinto de 0 y si el valor es **IPC_PRIVATE** aseguramos que sea un valor distinto de cualquier otro valor empleado como clave de memoria compartida.
- **Size**: Especifica el mínimo tamaño en bytes del segmento que deseamos crear o acceder.
- **Flags**: Especifica los flags de creación, con sus permisos al estilo ficheros. Los posibles valores son:
  - √ **IPC_CREAT**: Si la clave **key** no existe se crea un nuevo segmento de memoria compartida y se devuelve un nuevo ID. Si la clave ya existe y no se ha activado el flag **IPC_EXCL** se devuelve el ID del segmento ya existente. Si la clave ya existe y se ha activado el flag **IPC_EXCL**, **shmget()** falla y devuelve una notificación de error.
  - √ **IPC_EXCL**: Si la clave ya existe **shmget()** falla y devuelve una notificación de error.
- **Devuelve**: el ID del segmento compartido creado o ya existente, y un error en caso contrario.

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

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Esta rutina vincula un segmento de memoria compartida ya existente al espacio de direcciones que la invoque y devuelve un puntero en el espacio de direcciones del proceso que la invoca.

Donde:

- **shmid:** Especifica el ID una región de memoria compartida. Este ID será habitualmente devuelto por una llamada a **shmget()**.
- **shmaddr:** Especifica la dirección virtual en la que el proceso desea vincular la región de memoria compartida. Se puede indica 0 para que sea el propio kernel el que seleccione la dirección apropiada.
- **shmflg:** Especifica los flags de vinculación, que serán los. Los posibles valores son:
  - √ **SHM_RND**: Si el parámetro **addr** no es 0, el kernel redondea hacia abajo la dirección, si es necesario.
  - √ **SHM_RDONLY**: Si el proceso que invoca a **shmat()** tiene permisos de lectura sobre el segmento compartido , el kernel asocia la región sólo para lectura.
- **Devuelve**: un puntero con el segmento compartido obtenido.

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

int shmdt(const void *addr);
```

Esta llamada desvincula el puntero addr de un proceso de un segmento de memoria compartida.
Donde:
**Addr**: especifica una dirección de memoria virtual para la región compartida a ser desvinculada.
**Devuelve**: 0 si no hubo problemas y –1 en caso contrario.

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

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

Permite una gran variedad de operaciones de control de memoria compartida, indicas estas operaciones en el parámetro **cmd**.

Donde:

- **shmid**: Especifica el ID del segmento de memoria compartida.
- **cmd**: Especifica el tipo de operación a realizar sobre el segmento de memoria. Los valores posibles son:
  - √ **IPC_STAT**: Accede al segmento de memoria compartida ID copiando el contenido de estructura **shmid_ds** asociada a la estructura **buf**.
  - √ **IPC_SET**: Configura el segmento de memoria ID, copiando los valores de **buf** sobre la estructura **shmid_ds** del segmento. Los campos de esta estructura tomaran los siguientes valores:
    - **shm_perm.uid** se modifica con el ID del usuario propietario.
    - **shm_perm.gid** se modifica con el ID del grupo propietario.
    - **shm_perm.mode** se modifica con los nuevos modos de acceso y solamente se tocan los 9 bits más bajos (los correspondientes a los permisos).
  - √ **IPC_RMID**: Elimina el segmento compartido ID y deshace la asociación con la estructura **shmid_ds.**
  - √ **SHM_LOCK**: Bloquea el segmento de memoria ID.
  - √ **SHM_UNLOCK**: Desbloquea el segmento de memoria ID.
- **buf**: Especifica la dirección de la estructura **shmid_ds**.
- **Devuelve**: 0 si finalizó con éxito y –1 en caso de acabar con error, devolviendo en **errno** el error correspondiente.

# ANEXO A

```
/* El fichero cabecera line.h */


struct info {
char c;
int length;
};
#define KEY     ( (key_t) (1234))
#define SEGSIZE sizeof(struct info)
```

```
/*
  Fichero cline.c
  Este es un ejemplo de uso de memoria compartida.
  El programa opera en conjunto con el programa pline.
  Este programa permite modificar la longitud de la línea
  y el carácter escribiendo en un segmento de memoria compartido.
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"

main(int argc, char *argv[])
{
  int i, id;
  struct info *ctrl;
  struct shmid_ds shmbuf;

  if (argc!= 3) {
         fprintf(stderr, "uso: cline <char> <length>\n");
         exit(3);
  }
  id = shmget(KEY, SEGSIZE, 0);
  if (id < 0) {
         perror("cline: shmget faild:");
         exit(1);
  }
  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
         perror("cline: shmat failed:");
         exit(2);
  }

  /*Copia los datos de la linea de comandos en el segmento de memoria compartida */
  ctrl->c        = argv[1][0];
  ctrl->length   = atoi(argv[2]);

  exit(0);
}
```

```
/*
  Fichero: pline.c
  Este es un ejemplo de uso de memoria compartida.
  El programa imprime una linea de texto cada cuatro segundos.
  La longitud de la línea, y el carácter impreso se controlan
  mediante dos valores incluidos en una pequeña estructura de datos
  en un segmento de memoria compartida. Cualquier otro programa puede
  vincular dicho segmento y modificar los campos de la estructura de datos.
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"

main()
{
  int i, id;
  struct info *ctrl;
```

```
  /*struct shmid_ds shmbuf;*/

  id = shmget(KEY, SEGSIZE, IPC_CREAT | 0666);
  if (id < 0) {
         perror("pline: shmget failed:");
         exit(1);
  }

  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
         perror("pline: shmat failed:");
         exit(2);
  }
  /*Inicia los parámetros por defecto */
  ctrl->c        = 'a';
  ctrl->length   = 10;

  /*Este bucle imprime una linea cada 4 segundos */
  while (ctrl->length > 0) {
         for(i = 0; i < ctrl->length; i++)
                putchar(ctrl->c);
         putchar('\n');
         sleep(4);
  }
if (shmctl(id,IPC_RMID,(struct msqid_ds *) NULL)<0) {
perror("pline: shmctl failed:");
                exit(3);
}
           exit(0);
}
```

```
# Escuela Tecnica Superior de Ingenieria Informatica (Gijon)
# Sistemas de Computacion (Curso 5)
# Aplicaciones Distribuidas en Unix con memoria compartida

#*************************************************
# Variables
#*************************************************


CC=cc
CFLAGS=
CLIENTE = pline
SERVIDOR= cline

todo : $(CLIENTE) $(SERVIDOR)


#*************************************************
# Generacion del cliente y del servidor
#*************************************************
$(CLIENTE) : pline.c line.h
        $(CC) $(CFLAGS) -o $(CLIENTE) pline.c

$(SERVIDOR) : cline.c line.h
        $(CC) $(CFLAGS) -o $(SERVIDOR) cline.c


#*************************************************
# Utilidades
#*************************************************

limpiaobjs :
        rm -f *.o
```

# BIBLIOGRAFÍA

## Unix. Distributed Programming

Autor: Chris Brown
Editorial: Prentice-Hall
ISBN: 0-13-075896-5

## Advanced Programming in the Unix Environment

Autor: W. Richard Stevens
Editorial: Addison-Wesley
ISBN: 0-201-56317-1

## Unix Network Programming

Autor: W. Richard Stevens
Editorial: Prentice-Hall
ISBN: 0-13-949876-1

## Unix. Manual Pages