Sistemas Operativos – Práctica 3

FECHA DE ENTREGA:

SEMANA 16-23 ABRIL (HORA LÍMITE DE ENTREGA: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS).

La tercera práctica se va a desarrollar en tres semanas con el siguiente cronograma:
Semana 1: Memoria Compartida
Semana 2: Colas de Mensajes

Los ejercicios correspondientes a esta segunda práctica se van a clasificar en:

APRENDIZAJE, se refiere a ejercicios altamente recomendable realizar para el correcto seguimiento de los conocimientos teóricos que se presentan en esta unidad didáctica.

ENTREGABLE, estos ejercicios son obligatorios y deben entregarse correctamente implementados y documentados.

SEMANA 1
Memoria Compartida

La memoria convencional que puede direccionar un proceso a través de su espacio de direcciones virtual es local al proceso y cualquier intento de direccionar esa memoria desde otro proceso provocará una violación de segmento.

La memoria compartida es una zona de memoria común gestionada a través del sistema operativo, a la que varios procesos pueden conseguir acceso de forma que lo que un proceso escriba en la memoria sea accesible al resto de procesos.

Los procesos pueden comunicarse directamente entre sí compartiendo partes de su espacio de direccionamiento virtual, por lo que podrán leer y/o escribir datos en la memoria compartida. Para conseguirlo, se crea una región o segmento fuera del espacio de direccionamiento de un proceso y cada proceso que necesite acceder a dicha región, la incluirá como parte de su espacio de direccionamiento virtual.

Para trabajar con memoria compartida en C para Linux se utilizan básicamente las siguientes llamadas al sistema:
· *shmget*, crea una nueva región de memoria compartida o devuelve una existente.
· *shmat*, une lógicamente una región al espacio de direccionamiento virtual de un proceso.
· *shmdt*, separa una región del espacio de direccionamiento virtual de un proceso.
· *shmctl*, manipula varios parámetros asociados con la memoria compartida y realiza diversas operaciones de control, como por ejemplo borrar la memoria.

Estas funciones están incluidas en los ficheros de cabecera <sys/ipc.h>, <sys/shm.h> y <sys/types.h>.

La sintaxis de las funciones anteriores es la siguiente:

- *int shmget (key_t key, int size,int shmflg);*

donde *size* es el número de bytes de la región. El núcleo del sistema busca en la tabla de memoria compartida por la clave (*key*) indicada. La clave será un valor entero único para todos los procesos que compartan la misma región de memoria. Si existe una entrada y los permisos lo permiten, devuelve un identificador a la región (*shmid*). Si no lo encuentra y se ha indicado la opción IPC_CREAT en *shmflg*, el núcleo creará una nueva región. Otras opciones que se pueden indicar en el parámetro *shmflg* son los permisos de la región.

En caso de que la función genere un error, devolverá -1.

La región de memoria quedará reservada y sólo se asignará al espacio de direccionamiento de los procesos (tablas de páginas) cuando éstos se unan a la región mediante la ejecución de la llamada al sistema *shmat()*.

En algunos sistemas Unix, a la hora de utilizar la función *shmget* para crear un segmento de memoria compartida, es necesario usar los *flags* SHM_W y SHM_R además del *flag* IPC_CREAT. Estos *flags* dan permisos a la memoria compartida de escritura y lectura, respectivamente. Sin estos *flags*, el segmento de memoria compartida se crea, pero los procesos no pueden escribir o leer de ella.

- *char *shmat (int shmid, char *addr, int shmflg)*

donde *shmid* (valor que devuelve la función *shmget()*) identifica a la región de memoria compartida, *addr* es la dirección virtual donde el usuario quiere unir la memoria compartida y *shmflg* especifica los permisos de la región.
El valor que devuelve es la dirección virtual donde el núcleo ha unido la región. En caso de error, devuelve –1.

Cuando se ejecuta *shmat*(), el núcleo verifica que el proceso tiene los permisos necesarios para acceder a la región. Si la dirección especificada por el proceso es 0, el núcleo elige una dirección virtual conveniente.

- *int shmdt (char *addr);*

Para separar una región de memoria de su espacio de direccionamiento virtual, los procesos utilizan la llamada al sistema *shmdt()*, donde *addr* es la dirección que devuelve la llamada *shmat()*.

- *shmctl (int shmid, int cmd, struct shmid_ds shmstatbuf);*

La llamada al sistema *shmctl()* permite a un proceso buscar el estado y establecer los parámetros de la región de memoria compartida.

El parámetro *shmid* identifica a la entrada correspondiente en la tabla de memoria compartida, *cmd* indica el tipo de operación a realizar y *shmstatbuf* indica la dirección de una estructura de datos a nivel de usuario que contiene la información de estado de la entrada de la tabla de memoria compartida. La sintaxis de esta estructura es:

```
struct shmid_ds {
        struct ipc_perm shm_perm; /* permisos */
        int shm_segsz; /* Tamaño del segmento */
        ushort shm_cpid; /* PID del creador */
        ...
};
```

Algunas operaciones que se pueden indicar en el parámetro *cmd* de la llamada *shmctl()* son:

- IPC_STAT sitúa el valor actual de cada elemento de la estructura de datos asociada con shmid en la estructura apuntada por *shmstatbuf*.
- IPC_SET establece el valor de los elementos de la estructura de datos asociada con shmid, obtenidos a partir de la estructura de datos apuntada por shmstatbuf.
- IPC_RMID elimina del sistema el identificador de memoria compartida especificado en *shmid*.

Resumiendo, para usar memoria compartida en Linux es necesario seguir una serie de pasos que luego se traducen a llamadas al sistema.

1. Necesitamos obtener un identificador de IPC (Inter Process Communication). Para ello, una posibilidad es convertir una ruta (path) del sistema en un identificador IPC. Este identificador es necesario para crear la crear la zona de memoria virtual. Esto es muy sencillo de hacer con la llamada al sistema *ftok()*.
2. Crear el segmento de memoria compartida con la llamada al sistema *shmget()*.
3. Operar con la memoria compartida. Indicamos lo que queremos compartir con la llamada al sistema *shmat()*.
4. Destruimos el segmento de memoria compartida con la llamada al sistema *shmdt()* y *shmctl()*. Cuando un segmento de memoria compartida es borrado (mediante *shmctl())* en realidad sólo se está marcando para ser borrado. La eliminación definitiva del segmento la realizará el sistema operativo cuando todos los procesos enganchados a él se desenganchen.

Ejemplo de uso:

Proceso 1

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

#define FILEKEY "/bin/cat" /*Util para ftok */

#define KEY 1300
#define MAXBUF 10

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

int *buffer; /* shared buffer */
int key, id_zone;
int i;

char c;

/* Key to shared memory */
   int key = ftok(FILEKEY, KEY);
   if (key == -1) {
      fprintf (stderr, "Error with key \n");
      return -1;
   }

   /* We create the shared memory */
   id_zone = shmget (key, sizeof(int)*MAXBUF, IPC_CREAT | IPC_EXCL
                     |SHM_R | SHM_W);
   if (id_zone == -1) {
      fprintf (stderr, "Error with id_zone \n");
      return -1;
   }
   printf ("ID zone shared memory: %i\n", id_zone);


   /* we declared to zone to share */
   buffer = shmat (id_zone, (char *)0, 0);
   if (buffer == NULL) {
      fprintf (stderr, "Error reserve shared memory \n");
```

```
      return -1;
   }

   printf ("Pointer buffer shared memory: %p\n", buffer);

   for (i = 0; i < MAXBUF; i++)
      buffer[i] = i;


   /* The daemon executes until press some character */

   c = getchar();

   /* Free the shared memory */
   shmdt ((char *)buffer);
   shmctl (id_zone, IPC_RMID, (struct shmid_ds *)NULL);
   return 0;
}
```

Proceso 2

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <string.h>
#include <errno.h>
#include <sys/shm.h> /* shm*  */

#define FILEKEY "/bin/cat"
#define KEY 1300
#define MAXBUF 10

int main () {

int *buffer; /* shared buffer */
int key, id_zone;
int i;

char c;

   /* Key to shared memory */
   key = ftok(FILEKEY, KEY);
   if (key == -1) {
      fprintf (stderr, "Error with key \n");
      return -1;
   }

   /* we create the shared memory */
   id_zone = shmget (key, sizeof(int)*MAXBUF, IPC_CREAT | IPC_EXCL
                     |SHM_R | SHM_W);
   if (id_zone == -1) {
      fprintf (stderr, "Error with id_zone \n");
      return -1;
   }

   printf ("ID zone shared memory: %i\n", id_zone);

   /* we declared to zone to share */
   buffer = shmat (id_zone, (char *)0, 0);
   if (buffer == NULL) {
      fprintf (stderr, "Error reserve shared memory \n");
```

```
      return -1;
   }

   printf ("Pointer buffer shared memory: %p\n", buffer);

   /* Write the values of shared memory */

   for (i = 0; i < MAXBUF; i++)
      printf ("%i\n", buffer[i]);
   return 0;
}
```

Ejercicio 1. **(APRENDIZAJE)** Estudia qué hace el siguiente fragmento de código

```
if((shmid=shmget(clave,TAM_SEG,IPC_CREAT|IPC_EXCL|0660))==-1){
     printf("El segmento de memoria compartida ya existe\n");
     printf("    Abriendo como cliente\n");
     if((shmid=shmget(clave,TAM_SEG,0))==-1)
         printf("Error al abrir el segmento\n");
   }
  else
   printf("Nuevo segmento creado\n");
```

Facilidades de IPC desde la línea de comandos

Los programas estándar ipcs e ipcrm permiten controlar los recursos IPC que gestiona el sistema, y nos pueden ser de gran ayuda a la hora de depurar programas que utilizan estos mecanismos.

- ipcs, se utiliza para ver qué mecanismos están asignados y a quién. Si no se indica ninguna opción ipcs muestra un resumen de la información de control que se almacena para la memoria compartida, los semáforos y mensajes que hay asignados. Las principales opciones son:

  -m muestra información de los segmentos de memoria compartida que hay activos.
  -s muestra información de los semáforos que hay activos.
  -q muestra información de las colas de mensajes que hay activas.
  -b muestra información completa sobre los tipos de mecanismos IPC que hay activos.

- ipcrm, se utiliza para liberar un mecanismo asignado. A continuación se explican las opciones más comunes:
  -m shmid borra la zona de memoria compartida cuyo identificador coincide con shmid.
  -s semid borra el semáforo cuyo identificador coincide con semid.
  -q msqid borra la cola de mensajes cuyo identificador coincide con msqid.

Si interrumpimos un proceso con la señal de interrupción Ctrl-C o simplemente el proceso termina de forma anormal, el recurso (la memoria compartida, en este caso) no se libera y queda

en el sistema. La forma de borrarla sería con este comando. Es bastante normal mientras desarrollamos y depuramos nuestro programa que no liberemos, abortemos el proceso, etc. El número de memorias compartidas que podemos crear está limitado, por tanto en las pruebas podríamos empezar a obtener errores debido a que no se pueden crear las memorias, ipcrm nos permite eliminar las memorias que se han quedado "pendientes" en nuestras pruebas.

Ejercicio 2. **(ENTREGABLE) (2.5 ptos) Condición de carrera** En este ejercicio se pide un programa escrito en lenguaje C, *ejercicio2.c.* El programa generará n procesos hijos (n es un argumento de entrada al programa). El proceso padre reservará un bloque de memoria, que compartirá con los procesos hijo, suficiente para una estructura del tipo

```
struct info{
    char nombre[80];
    int id;
}
```

Cuando el proceso padre reciba la señal SIG_USR1 leerá de la zona de memoria compartida e imprimirá su contenido, nombre del usuario e identificador.

Cada proceso hijo realizará los siguientes pasos:

- dormirá un tiempo aleatorio y
- solicitará al usuario que dé de alta un cliente (recoger el nombre del cliente)
- verificará de la zona de memoria compartida cuál fue el último id y lo incrementará en una unidad.
- enviará la señal SIG_USR1 al proceso padre
- terminará correctamente.

El proceso padre terminará cuando todos los proceso hijo hayan terminado.

1. Explica en qué falla el planteamiento de este ejercicio.
2. Implementa un mecanismo (ejercicio2_solved.c) para solucionar este problema basado en tu conocimiento de la asignatura.

Ejercicio 3. **(ENTREGABLE) (4 ptos)** Realizar un programa que implemente el problema del productor-consumidor. Para ello, se debe crear una región de memoria compartida que alojará el búfer en el que el productor escribirá los elementos que produzca, letras en orden alfabético y posteriormente números del 0 al 9, y del que el consumidor extraerá dichas letras y números mostrándolos en pantalla. Los procesos productor y consumidor se ejecutarán concurrentemente y la sincronización entre los procesos debe seguir realizándose a través de semáforos, prestando especial atención para evitar el interbloqueo entre los procesos. Para implementarlo se debe hacer uso de la biblioteca de semáforos realizada en la práctica anterior.

Mapeo de ficheros en memoria: Función mmap():

Mmap() es una alternativa para gestionar espacios de memoria específicamente diseñada para establecer un mapeo entre el espacio de direccionamiento de un proceso y un fichero, una memoria compartida u otros recursos. El prototipo de la función es el siguiente:

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

```
void *mmap(void *addr, size_t len, int prot, int flags,
        int fildes, off_t off);
```

Siguiendo este prototipo, un posible uso de esta llamada y su interpretación sería el siguiente:

```
pa=mmap(addr, len, prot, flags, fildes, off);
```

Esta llamada establecería un mapeo entre el espacio de direccionamiento del proceso invocador en la dirección de memoria *pa* de *len* bytes de longitud con respecto del objeto referenciado por el descriptor de fichero *fildes* con un desplazamiento de `off` bytes. El valor de `pa` es una función del parámetro `addr` y los valores especificados en `flags`, explicados a continuación. El espacio otorgado por tanto comenzará en `pa` y se extenderá por `len` bytes mas.

El parámetro `prot` establece los permisos de lectura, escritura, ejecución o cualquier combinación de los anteriores accesos. `prot` tomará el valor de una o las siguientes constantes, concatenadas por el operador OR y definidas en la librería `<sys/mman.h>`.

Constante	Descripción
PROT_READ	Permiso de lectura.
PROT_WRITE	Permiso de escritura.
PROT_EXEC	Permiso de ejecución.
PROT_NONE	Sin permisos.

El parámetro `flags` nos brinda información adicional para la gestión del mapeo de memoria. Al igual que en el caso de `prot`, se especifican los flags mediante el operador OR y definidas en la librería `<sys/mman.h>`.

Constante	Descripción
MAP_SHARED	Cambios compartidos.
MAP_PRIVATE	Cambios privados.
MAP_FIXED	Interpretar *addr* literal.

Si la función mmap() retorna error, devuelve la constante MAP_FAILED, útil para control de errores. La función munmap libera el mapeo, devolviendo -1 en caso de error.

```
int munmap(void *addr, size_t length);
```

El fichero deberá ser cerrado por close() una vez que el mapeo termine, munmap() no cierra el fichero, solo libera el mapeo.

Mas información sobre la función mmap() puede encontrarse en el manual de la misma. *man mmap*.

Un buen ejemplo para entender la funcionalidad de esta alternativa es el siguiente. En vez de usar las funciones read() y write() se puede usar equivalentemente la función mmap() del siguiente modo:

fildes = open(...)

```
lseek(fildes, some_offset)

read(fildes, buf, len)

/* Usar los datos en buf. */
```

Se convierte en:

fildes = open(...)

```
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes,
some_offset)

/* Usar los datos en address. */
```

Para extraer el tamaño del fichero podemos hacer uso de la función fstat() de `<sys/stat.h>` con el siguiente prototipo y que vuelca la información en una estructura stat importada con la librería de la que podemos extraer el tamaño con el campo size:

```
int fstat(int fd, struct stat *buf);
```

Ejercicio 4. (ENTREGABLE) (1 pto.) Ejecuta un programa que invoque a un hilo que escriba en un fichero un número n de números aleatorios donde n es un número aleatorio del 1000 al 2000 y los números aleatorios de este fichero tendrán un valor entre el 100 y el 1000 separados por comas. El programa deberá esperar a que este hilo termine para invocar a otro hilo que lea este fichero empleando mmap(). El tamaño del fichero deberá ser extraído mediante la función fstat y el descriptor del fichero será pasado por parámetro. Una vez mapeado el contenido del fichero en memoria, el hilo deberá, a partir de la dirección de memoria donde está mapeado el contenido, modificar las comas por espacios e imprimir esta información por pantalla.

SEMANA 2

Colas de Mensajes

Las colas de mensajes, junto con los semáforos y la memoria compartida son los recursos compartidos que pone Unix a disposición de los programas para que puedan intercambiarse información.

En C para Unix es posible hacer que dos procesos sean capaces de enviarse mensajes y de esta forma intercambiar información. El mecanismo para conseguirlo es el de una cola de mensajes. Los procesos introducen mensajes en la cola y se va almacenando en ella. Cuando un proceso extrae un mensaje de la cola, extrae el primer mensaje que se introdujo y dicho

mensaje se borra de la cola. Las colas de mensajes son un recurso global que gestiona el sistema operativo.

El sistema de colas de mensajes es análogo a un sistema de correos, y en él se pueden distinguir dos tipos de elementos:

- *Mensajes*: son similares a las cartas que se envían por correo, y por tanto contienen la información que se desea transmitir entre los procesos.

- *Remitente* y *destinatario*: es el proceso que envía o recibe los mensajes, respectivamente. Ambos deberán solicitar al sistema operativo acceso a la cola de mensajes que los comunica antes de poder utilizarla. Desde ese momento, el proceso remitente puede componer un mensaje y enviarlo a la cola de mensajes, y el proceso destinatario puede acudir en cualquier momento a recuperar un mensaje de la cola.

Es posible hacer "tipos" de mensajes distintos, de forma que cada tipo de mensaje contiene una información distinta y va identificado por un entero. Por ejemplo, los mensajes de tipo 1 pueden contener el saldo de una cuenta de banco y el número de dicha cuenta, los de tipo 2 pueden contener el nombre de una sucursal bancaria y su calle, etc. Los procesos luego pueden retirar mensajes de la cola selectivamente por su tipo. Si un proceso sólo está interesado en saldos de cuentas, extraería únicamente mensajes de tipo 1, etc.

Los procesos acceden secuencialmente a la cola, leyendo los mensajes en orden cronológico (desde el más antiguo al más reciente), pero selectivamente, esto es, considerando sólo los mensajes de un cierto tipo: esta última característica nos da un tipo de control de la prioridad sobre los mensajes que leemos.

Las funciones para trabajar con colas de mensajes en Unix en C están incluidas en las librerías

<sys/types.h>, *<sys/ipc.h>* y *<sys/msg.h>*. En particular, las funciones que se van a emplear son *msgget()*, *msgsnd()*, *msgrcv()* y *msgctl()*. La sintaxis de las funciones anteriores es la siguiente:

Creación de colas de mensajes: *int msgget (key_t key, int msgflg);*

Recibe como argumento una clave IPC y los flags similares a los que ya hemos visto con memoria compartida y semáforos, ejemplo: IPC_CREAT | 0660 que crea la cola, si no existe, y da acceso al propietario y grupo de usuarios.
La función devuelve el identificador de la cola. En caso de que la función genere un error, devolverá -1.

Enviar datos a una cola de mensajes: *int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);*

- *msqid* (valor que devuelve la función *msgget()*) es el identificador de la cola,
- *msgp* es un puntero al mensaje que tenemos que enviar. La estructura *msgbuf* tenemos que definirla en ella hay indicar forzosamente el

primer campo de la estructura como un long que representará el tipo de mensaje y el resto de los campos de la estructura es el mensaje que enviamos.

Por defecto, la estructura base del sistema que describe un mensaje se llama

msgbuf y está declarada en `linux/msg.h`

```
/* message buffer for msgsnd and msgrcv calls */

struct msgbuf {
    long mtype;         /* type of message */
    char mtext[1];      /* message text */
};
```

El campo *mtype* representa el tipo de mensaje y es un número estrictamente

Positivo (>0). El segundo campo representa el contenido del mensaje.

La estructura msgbuf puede ser redefinida y contener datos complejos; por ejemplo:

```
struct message {
    long mtype;         /* message type */
    long sender;        /* sender id */
    long receiver;      /* receiver id */
    struct info         /* message content
    ...
};
```

- *msgsz* es la dimensión del mensaje, excluyendo la longitud del tipo mtype que tiene la longitud de un long, que es normalmente de 4 bytes. Sobre la estructura anterior *message* la longitud del mensaje sería:

```
length = sizeof(struct message) - sizeof(long);
```

- *msgflg* es un flag relativo a la política de espera referente a cuando la cola está llena. Por defecto, el flag es IPC_WAIT, es decir, en el caso de que la cola esté completa, el proceso se queda bloqueado esperando a que se libere algún hueco. Si *msgflg* es puesta a IPC_NOWAIT y no hubiera espacio disponible, el proceso emisor no esperará y saldrá con el código de error EAGAIN.

Recibir datos de una cola de mensajes: *int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg);*

La llamada a sistema *msgrcv* lee un mensaje de la cola de mensajes especificada por *msqid* y de tipo *mtype.* El mensaje leído se guarda en *msgp*, eliminándolo de la cola de mensajes. El argumento *msgsz* especifica el tamaño máximo en bytes de la zona de memoria apuntada por *msgp*. En el argumento *msgflg* hay diversas opciones:

- MSG_NOERROR: si el tamaño del mensaje es superior al tamaño especificado en el campo *msgsz*, y si la opción MSG_NOERROR está posicionada, el mensaje se truncará. La parte sobrante se pierde. En el caso en que no esté esta opción, el mensaje no se retira de la cola y la llamada fracasa devolviendo como error E2BIG.

- IPC_NOWAIT: esta opción permite evitar la espera activa. Si la cola no está nunca vacía, se devuelve el error ENOMSG. Si esta opción no está activa, la llamada se suspende hasta que un dato del tipo solicitado entre en la cola de mensajes.

El tipo de mensaje a leer debe especificarse en el campo *mtype*:

- Si *mtype* es igual a 0, se lee el primer mensaje de la cola, es decir, el mensaje más antiguo, sea cual sea su tipo.
- Si *mtype* es negativo, entonces se devuelve el primer mensaje de la cola con el tipo menor, inferior o igual al valor absoluto de *mtype*.
- Si *mtype* es positivo, se devuelve el primer mensaje de la cola con un tipo estrictamente igual a *mtype.* En el caso de que esté presente la opción MSG_EXCPT, se devolverá el primer mensaje con un tipo diferente.

Operaciones de control sobre una cola de mensajes: *int msgctl (int msqid, int cmd, struct msqid_ds *buf);*

Donde *msqid* es el identificador de la cola, *cmd* es la operación que se quiere realizar y *buf* es una estructura donde se guarda la información asociada a la cola en caso de que la operación sea IPC_STAT o IPC_SET. Las operaciones que se pueden realizar son:

IPC_STAT: guarda la información asociada a la cola de mensajes en la estructura apuntada por *buf*. Esta información es por ejemplo el tamaño de la cola, el identificador del proceso que la ha creado, los permisos, etc.

IPC_SET: Establece los permisos de la cola de mensajes a los de la estructura *buf*. IPC_RMID: Marca la cola para borrado. No se borra hasta que no haya ningún proceso que esté asociada a él.

Devuelve 0 si éxito y –1 en caso de error.Ejemplo de uso:

Programa 1

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

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

#define N 33
```

```
typedef struct _Mensaje{
      long id;  /*Campo obligatorio a long que identifica el tipo de mensaje*/

      /*Informacion a transmitir en el mensaje*/

     int valor;

     char aviso[80];
}mensaje;

int main(void){

 key_t clave;

 int msqid;

mensaje msg;


/*

 *  Se obtiene una clave a partir de un fichero existente cualquiera
 *  y de un entero cualquiera. Todos los procesos que quieran compartir la

 * cola de mensaje deben usar el mismo fichero y el mismo entero.
 */

 clave = ftok ("/bin/ls", N);

 if (clave == (key_t) -1)

 {
          perror("Error al obtener clave para cola mensajes\n");
          exit(EXIT_FAILURE);
 }

/*
 * Se crea la cola de mensajes y se obtiene un identificador para ella.
 * El IPC_CREAT indica que cree la cola de mensajes si no lo está.

 * 0600 son permisos de lectura y escritura para el usuario que lance
 * los procesos. Es importante el 0 delante para que se interprete en octal.

 */
 msqid = msgget (clave, 0600 | IPC_CREAT);
 if (msqid == -1)

 {
           perror("Error al obtener identificador para cola mensajes");
          return(0);
 }
 msg.id = 1;  /*Tipo de mensaje*/
 msg.valor= 29;

 strcpy (msg.aviso, "Hola a todos");

/*

 * Se envia el mensaje. Los parámetros son:
```

* Id de la cola de mensajes.

* Dirección al mensaje, convirtiéndola en puntero a (struct msgbuf *)
* Tamaño total de los campos de datos de nuestro mensaje (parte del envio)

* flags. IPC_NOWAIT indica que si el mensaje no se puede enviar
*(habitualmente porque la cola de mensajes esta llena), que no espere
* y de un error. Si no se pone este flag, el programa queda bloqueado

* hasta que se pueda enviar el mensaje.

```
 */

 msgsnd (msqid, (struct msgbuf *) &msg,  sizeof(mensaje) - sizeof(long),   IPC_NOWAIT);

/*

 * Se recibe un mensaje del otro proceso. Los parámetros son:

 * Id de la cola de mensajes.
 * Dirección del sitio en el que queremos recibir el mensaje, convirtiendolo en puntero a
(struct msgbuf *).

 * Tamaño máximo de nuestros campos de datos.
 * Identificador del tipo de mensaje que queremos recibir.
 * flags. En este caso se quiere que el programa quede bloqueado hasta

 *  que llegue un mensaje de tipo 2. Si se pone IPC_NOWAIT, se devolvería
 *   un error en caso de que no haya mensaje de tipo 2 y el proceso continuaría
 ejecutándose.

 */

 msgrcv (msqid, (struct msgbuf *)&msg, sizeof(mensaje) - sizeof(long),  2, 0);

 printf("Recibido mensaje tipo %d \n", msg.id);
 printf("Dato_Numerico = %d \n", msg.valor);
 printf("Mensaje = %s \n", msg.aviso);

/*

 * Se borra y cierra la cola de mensajes.
 * IPC_RMID indica que se quiere borrar. El puntero del final son datos

 * que se quieran pasar para otros comandos. IPC_RMID no necesita datos,
 * así que se pasa un puntero a NULL.

 */

  msgctl (msqid, IPC_RMID, (struct msqid_ds *)NULL);

 exit(EXIT_SUCCESS);

}
```

Programa 2

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


#define N 33

typedef struct _Mensaje{

 long id; /* Identificador del mensaje*/
```

```
/* Informacion que se quiere transmitir*/

 int valor;

 char aviso[80];

}mensaje;

int main(void)
{

 key_t clave;

 int msqid;

mensaje msg;

 clave = ftok ("/bin/ls", N); /*Misma clave que el proceso cooperante*/

 if (clave == (key_t)-1)
 {
          perror("Error al obtener clave para cola mensajes \n");

          exit(EXIT_FAILURE);

 }
/*
* Se crea la cola de mensajes y se obtiene un identificador para ella.
 * El IPC_CREAT indica que cree la cola de mensajes si no lo está.
 *  0600 son permisos de lectura y escritura para el usuario que lance

 * los procesos. Es importante el 0 delante para que se interprete en octal.
 */

 msqid = msgget (clave, 0600 | IPC_CREAT);
 if (msqid == -1)
 {

          perror ("Error al obtener identificador para cola mensajes \n");
          exit(EXIT_FAILURE);

 }
/*
 * Recepcion de un mensaje
 */

 msgrcv (msqid, (struct msgbuf *) &msg, sizeof(mensaje) - sizeof(long),   1, 0);

 printf("Recibido mensaje tipo %d \n", msg.id);
 printf("Dato_Numerico = %d \n", msg.valor);
 printf("Mensaje = %s \n", msg.aviso);

 /*

  * Envio de un mensaje
   */

 msg.id = 2;
 msg.valor = 13;
```

```
 strcpy (msg.aviso, "Adios");

 msgsnd (msqid, (struct msgbuf *) &msg, sizeof(mensaje)-sizeof(long), IPC_NOWAIT);

  msgctl (msqid, IPC_RMID, (struct msqid_ds *)NULL);

   exit(EXIT_SUCCESS);
}
```

Ejercicio 5. (ENTREGABLE) (2.5 ptos) Se pretende diseñar e implementar una cadena de montaje usando colas de mensajes de UNIX. La cadena de montaje está compuesta por tres procesos (A, B y C), cada uno especializado en una función. La comunicación entre cada par de procesos (es decir el proceso i y el proceso i+1) se realiza a través de una cola de mensajes de UNIX. En esta cadena de montaje, cada proceso realiza una función bien diferenciada:

- El primer proceso A lee de un fichero entrada.txt y escribe en la primera cola de mensajes trozos del fichero de longitud máxima 2KB.
- El proceso intermedio B lee de la cola de mensajes cada trozo del fichero y realiza una simple función de conversión, consistente en reemplazar las letras por su siguiente en el abecedario, convirtiendo la 'z' en 'a'. Una vez realizada esta transformación, escribe el contenido en la cola de mensajes.
- El último proceso lee de la cola el trozo de memoria y lo vuelca al fichero f2.

El programa principal acepta dos argumentos de entrada, correspondientes al nombre del fichero origen (f1) y destino (f2). Debe ejecutarse de la siguiente manera:

```
$ cadena_montaje <f1> <f2>
```

Se pide:
a)Decisiones de diseño necesarias para la implementación del programa cadena_montaje.

Se entrega un archivo cadena_montaje.doc.

b)Código fuente en el lenguaje de programación C del programa cadena_montaje.c