Práctica 3

FECHA DE ENTREGA: DEL 19 DE ABRIL AL 23 DE ABRIL
(HORA LÍMITE: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS)

MEMORIA COMPARTIDA

Introducción a la Memoria Compartida

La **memoria convencional** que puede direccionar un proceso a través de su espacio de direcciones virtual es local al proceso, y no puede ser accedida desde otro proceso.

En cambio, 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, como se muestra en la Figura 1.

Funciones para el Manejo de Memoria Compartida

En sistemas Unix se puede trabajar con memoria compartida en C utilizando la API de POSIX (descrita en el manual de `shm_overview`), que proporciona entre otras las siguientes funciones:

- `shm_open`: crea o abre un segmento de memoria compartida, y devuelve un descriptor de fichero que hace referencia al mismo.
- `ftruncate`: cambia el tamaño del fichero o segmento de memoria compartida indicado.
- `mmap`: une lógicamente una región de un fichero o memoria compartida al espacio de direccionamiento virtual de un proceso.
- `munmap`: separa una región de un fichero o memoria compartida del espacio de direccionamiento virtual de un proceso.

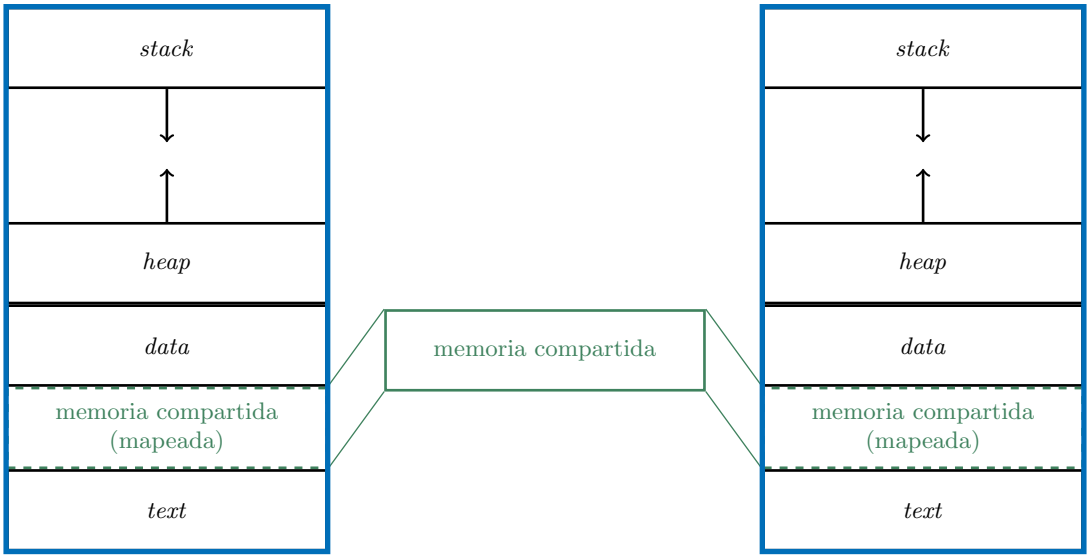

Figura 1: Memoria compartida entre dos procesos.

- `close`: cierra un descriptor de fichero.
- `shm_unlink`: elimina el nombre de un segmento de memoria compartida. El segmento en sí será eliminado cuando ningún proceso tenga un descriptor de fichero que se corresponda con dicho segmento, y ningún proceso tenga una región de memoria asociada al segmento.
- `fstat`: obtiene información sobre el fichero o segmento de memoria asociado a un descriptor de fichero. En particular, se puede usar para obtener el tamaño del segmento de memoria una vez creado.

Estas funciones están incluidas en las ficheros de cabecera `<sys/mman.h>`, `<sys/stat.h>`, `<fcntl.h>`, `<unistd.h>` y `<sys/types.h>`. Además, será necesario enlazar el programa con la biblioteca de tiempo real, es decir, se debe añadir a la llamada al compilador `gcc` el parámetro `-lrt`.

Aunque el uso de estas funciones se encuentra detallado en la página correspondiente del manual, a continuación se describirán brevemente.

Creación y Eliminación de Segmentos de Memoria Compartida

Los segmentos de memoria se crean o abren usando la función `shm_open`, de manera análoga a como se hace con la función `open` para ficheros. El primer argumento, `name`, es el nombre del segmento de memoria, una cadena que debe comenzar por el carácter '`/`' y no tener ningún carácter '`/`' adicional, por ejemplo `"/ejemplo"`. Los parámetros `oflag` y `mode` son similares a los de `open`. En concreto, `oflag` requiere que al menos se especifique `O_RDONLY` o `O_RDWR`. El primero indicará que el descriptor de fichero devuelto solo permitirá leer, mientras que el segundo indica que se permitirá la lectura y la escritura. Adicionalmente, se podrán usar las banderas `O_CREAT`, para crear el segmento si no existe, `O_EXCL`, que en combinación con `O_CREAT` reportará error si el segmento ya existía, y `O_TRUNC`, que reducirá el segmento a tamaño a cero si el segmento ya existía. Si se especifica `O_CREAT` y se crea un nuevo segmento de memoria, entonces el parámetro `mode` especificará los permisos del segmento creado. El valor devuelto por esta función es un descriptor de fichero haciendo referencia al segmento de memoria compartida abierto o creado.

En caso de error devuelve `-1`.

Para cerrar segmentos de memoria compartida, y dado que `shm_open` devuelve un descriptor de fichero, basta con utilizar la función `close`, que recibe como único argumento `fd` el descriptor del fichero, que será liberado una vez realizado la separación lógica del fichero o segmento de memoria compartida del espacio de direccionamiento virtual del proceso.

Por último, para borrar la ruta al segmento de memoria compartida se usa la función `shm_unlink`, análoga a `unlink`, que recibe como argumento `name` el nombre que se empleó en `shm_open` y que se desea borrar.

0.40 ptos. **Ejercicio 1: Creación de Memoria Compartida.** Dado el siguiente código en C:

shm_open.c

```c
/* This is only a code fragment, not a complete program... */
    fd_shm = shm_open (SHM_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRUSR |
        S_IWUSR);
    if (fd_shm == -1) {
        if (errno == EEXIST) {
            fd_shm = shm_open(SHM_NAME, O_RDWR, 0);
            if (fd_shm == -1) {
                perror("Error opening the shared memory segment");
                exit(EXIT_FAILURE);
            }
        }
        else {
            perror("Error creating the shared memory segment\n");
            exit(EXIT_FAILURE);
        }
    }
    else {
        printf ("Shared memory segment created\n");
    }
```

0.20 ptos. a) Explicar en qué consiste este código, y qué sentido tiene utilizarlo para abrir un objeto de memoria compartida.

0.20 ptos. b) En un momento dado se deseará forzar (en la próxima ejecución del programa) la inicialización del objeto de memoria compartida `SHM_NAME`. Explicar posibles soluciones (en código C o fuera de él) para forzar dicha inicialización.

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Cambiar el Tamaño de Ficheros y Obtener Información

Para cambiar el tamaño de un fichero (o de un segmento de memoria compartida) se usa la función `ftruncate`, que recibe como primer argumento, `fd`, un descriptor de fichero válido (por ejemplo, el descriptor devuelto por `shm_open`). El segundo argumento, `length`, es el tamaño que se le quiera dar a la memoria.

Nota. *Por lo general, cuando se trabaja con memoria compartida solo es necesario llamar a la función* `ftruncate` *una vez, desde el proceso que crea el segmento de memoria.*

Para obtener información acerca de un fichero o segmento de memoria compartida se usa la función `fstat`, que recibe el descriptor del fichero del que se desea obtener información, `fd`, y la información se almacena en la estructura apuntada por el segundo argumento, `buf`. El campo

más interesante de esta estructura es `st_size`, que indica el tamaño del fichero o segmento de memoria.

0.40 ptos. **Ejercicio 2: Tamaño de Ficheros.** Dado el siguiente código en C:

`file_truncate.c`

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#define SHM_NAME "/shm_example"
#define MESSAGE "Test message"


int main(int argc, char *argv[]) {
    int fd;
    struct stat statbuf;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <FILE>\n", argv[0]);
    }

    fd = open(argv[1], O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    dprintf(fd, "%s", MESSAGE);
    /* Get size of the file. */

    /* Truncate the file to size 5. */

    close(fd);
    exit(EXIT_SUCCESS);
}
```

0.20 ptos. a) Completar el código anterior para obtener el tamaño del fichero abierto.

0.20 ptos. b) Completar el código anterior para truncar el tamaño del fichero a 5 B. ¿Qué contiene el fichero resultante?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Mapeado de Ficheros

Para facilitar el uso de ficheros y memoria compartida, existe la posibilidad de unirlos lógicamente al espacio de direccionamiento virtual del proceso. Para ello se utiliza la función `mmap`, que recibe como primer argumento, `addr`, la dirección de memoria donde se enlazará el fichero o segmento de memoria (si se deja este parámetro a `NULL` será el sistema operativo el que escoja una dirección apropiada). El segundo parámetro, `length`, es el tamaño de la memoria que se desea enlazar (cuando se trabaja con memoria compartida, suele ser el tamaño total del segmento). El tercer argumento, `prot`, especifica la protección de la región de memoria, y debe ser compatible con los permisos asociados al descriptor de fichero. Los valores que puede tomar `prot` serán el OR a nivel de bits de una o varias de las siguientes constantes:

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

El parámetro `flags` permite añadir, también usando OR a nivel de bits, banderas que especifican información adicional. Es obligatorio especificar al menos una, y solo una, de las siguientes: `MAP_SHARED`, indicando que los cambios realizados en la memoria por el proceso actual deben ser visibles para los demás procesos, o `MAP_PRIVATE`, indicando que estos cambios no serán visibles. Por lo general, si se quiere que todos los procesos vean la misma memoria, se usará siempre `MAP_SHARED`. El siguiente parámetro, `fd` es el descriptor de fichero cuyo contenido se quiere enlazar a la memoria (por ejemplo, el descriptor devuelto por `shm_open`). El último argumento, `offset`, indica la posición dentro del fichero o segmento de memoria en la que se encuentra la memoria deseada, y debe ser múltiplo del tamaño de página. Para enlazar todo el contenido del segmento de memoria, `offset` tendrá el valor `0`. El valor devuelto por `mmap` será la dirección de memoria dentro del proceso que se ha enlazado a la memoria del fichero o segmento de memoria especificado. El puntero devuelto, por tanto, puede ser diferente para distintos procesos. Entre otras cosas, esto implica que no se debe guardar punteros dentro de la propia memoria compartida, ya que incluso aunque en el proceso original se refieran a direcciones dentro de la memoria compartida, esto no tiene por qué ser cierto para el resto de los procesos. En caso de error, `mmap` devolverá la constante `MAP_FAILED`.

Para separar la memoria enlazada se usa la función `munmap`, que recibe la dirección original devuelta por `mmap`, `addr`, y el tamaño pasado a `mmap`, `length`.

0.30 ptos.

Ejercicio 3: Mapeado de Ficheros. Dado el siguiente código en C:

file_mmap.c

```c
/* READER */
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

#define FILE_NAME "test_file.dat"

int main(void) {
    int created = 0;
    int fd;
    int *mapped = NULL;


    if ((fd = open (FILE_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRUSR |
        S_IWUSR)) == -1) {
        if (errno == EEXIST) {
            if ((fd = open(FILE_NAME, O_RDWR, 0)) == -1) {
                perror("open");
                exit(EXIT_FAILURE);
            }
        }
        else {
            perror("open");
            exit(EXIT_FAILURE);
```

```
28          }
29      }
30      else {
31          if (ftruncate(fd, sizeof(int)) == -1) {
32              perror("ftruncate");
33              close(fd);
34              exit(EXIT_FAILURE);
35          }
36          created = 1;
37      }
38
39      if ((mapped = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, 0)) == MAP_FAILED) {
40          perror("mmap");
41          close(fd);
42      }
43      close(fd);
44
45      if (created) {
46          *mapped = 0;
47      }
48      *mapped += 1;
49      printf("Counter: %d\n", *mapped);
50      munmap(mapped, sizeof(int));
51
52      exit(EXIT_SUCCESS);
53  }
```

0.20 ptos. a) ¿Qué sucede cuando se ejecuta varias veces el programa anterior? ¿Por qué?

0.10 ptos. b) ¿Se puede leer el contenido del fichero "`test_file.dat`" con un editor de texto? ¿Por qué?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Inspeccionar Memoria en Linux

Segmentos de Memoria Compartida

Aunque POSIX no especifica ninguna forma estándar para listar todos los segmentos de memoria compartida creados con `shm_open`, cada sistema Unix suele implementar alguna. En el caso de Linux, la implementación de la memoria compartida se realiza montando un sistema de archivos especial, que usa RAM en lugar de disco, en `/dev/shm`. Por tanto, en `/dev/shm` se puede ver la memoria compartida creada, e incluso manipularla con programas que trabajen con ficheros (por ejemplo, usar el comando `rm` para borrarla; algo útil en el caso de que procesos que usen memoria compartida aborten o sean finalizados sin borrar la misma).

Mapas de Memoria

Los mapas de memoria añadidos al proceso con `mmap` se pueden ver en el fichero `/proc/<pid>/maps`. Además de aquellos mapeos que se hayan realizado explícitamente en el código, también se pueden observar aquellos que pertenecen a la carga inicial del programa y las bibliotecas dinámicas de las que depende (realizados por el enlazador al hacer `exec`), y aquellos correspondientes a la apertura de bibliotecas dinámicas durante la ejecución del código (con `dlopen`).

Es posible identificar el segmento dedicado a la memoria dinámica (*heap*), reservada con `malloc`, el segmento dedicado a la pila del hilo principal (*stack*), y cada uno de los segmentos correspondientes al programa original, las bibliotecas que usa y otros ficheros o memoria compartida mapeados por el proceso, atendiendo al valor de la última columna. Para cada ejecutable y biblioteca, según los permisos de cada segmento, se puede distinguir un segmento que almacena el código (con permisos de lectura y ejecución), otro que almacena variables globales y estáticas (con permisos de lectura y escritura), y otro para constantes, como los literales de cadena (con permiso solo de lectura). Además, si el ejecutable o biblioteca tiene variables globales inicializadas a `0` (o sin inicializar, ya que es el valor por defecto para variables globales), no se guarda el contenido en el propio ejecutable o biblioteca, sino que solo se indica su tamaño y se mapea un segmento anónimo con permisos de lectura y escritura a la hora de cargarlo (segmento BSS).

Uso de Memoria Compartida

Resumen

De forma resumida, para usar la memoria compartida con la API de POSIX se ha de hacer lo siguiente:

1. Obtener un descriptor de fichero a la memoria compartida con `shm_open`. El proceso que cree la memoria debe usar la bandera `O_CREAT` y dar tamaño al segmento con `ftruncate`. Los demás deben averiguar el tamaño del segmento de memoria usando `fstat` en caso de no conocerlo de antemano.
2. Enlazar la memoria al espacio de direcciones del proceso con `mmap`, y cerrar el descriptor de fichero con `close`.
3. Usar la memoria compartida.
4. Cuando el proceso no requiera la memoria compartida, eliminarla de su espacio de direcciones con `munmap`.

→ Cuando no quede ningún proceso que vaya a abrir un descriptor de fichero a ese segmento de memoria, borrar su nombre con `shm_unlink`. Tras esto, la memoria asociada se borrará definitivamente cuando todos los descriptores de fichero asociados al segmento hayan sido cerrados y ningún proceso tenga enlazado el segmento a su espacio de direcciones. La memoria compartida también se borrará automáticamente cuando el ordenador se reinicie.

Nota. *La memoria reservada con* `mmap`*, al igual que otros recursos (como por ejemplo los semáforos abiertos con* `sem_open`*), se hereda por los procesos hijo, así que no es necesario reabrirla en cada uno de ellos.*

Nota. *Los pasos anteriormente descritos (basados en* `mmap` *y* `munmap`*) pueden realizarse con ficheros ordinarios reemplazando* `shm_open` *por* `open` *y* `shm_unlink` *por* `unlink`*. La ventaja de usar memoria compartida es que se garantiza que todo el contenido del segmento está en RAM, y por tanto es más rápido de acceder. En caso de usar ficheros el sistema operativo irá cargando en RAM las páginas que se estén usando.*

Ejemplo

0.80 ptos.

Ejercicio 4: Memoria Compartida. Dado el siguiente código en C:

`shm_writer.c`

```c
#include <fcntl.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define SHM_NAME "/shm_example"
#define INT_LIST_SIZE 10
#define MSG_MAX 100
#define MESSAGE "Hello world shared memory!"

typedef struct{
    int integer_list[INT_LIST_SIZE];
    char message[MSG_MAX];
} ShmExampleStruct;

int main(void) {
    int i, fd_shm;
    ShmExampleStruct *shm_struct = NULL;

    /* Creation of the shared memory. */
    if ((fd_shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_EXCL,  S_IRUSR | S_IWUSR)) == -1) {
        perror("shm_open");
        exit(EXIT_FAILURE);
    }

    /* Resize of the memory segment. */
    if (ftruncate(fd_shm, sizeof(ShmExampleStruct)) == -1) {
        perror("ftruncate");
        shm_unlink(SHM_NAME);
        exit(EXIT_FAILURE);
    }

    /* Mapping of the memory segment. */
    shm_struct = mmap(NULL, sizeof(ShmExampleStruct), PROT_READ | PROT_WRITE, MAP_SHARED, fd_shm, 0);
    close(fd_shm);
    if (shm_struct == MAP_FAILED) {
        perror("mmap");
        shm_unlink(SHM_NAME);
        exit(EXIT_FAILURE);
    }
    printf("Pointer to shared memory segment: %p\n", (void*)shm_struct);

    /* Initialization of the memory. */
    memcpy(shm_struct->message, MESSAGE, sizeof(MESSAGE));
    for (i = 0; i < INT_LIST_SIZE; i++) {
        shm_struct->integer_list[i] = i;
    }

    /* The daemon executes until some character is pressed. */
    getchar();

    /* Unmapping and freeing of the shared memory */
    munmap(shm_struct, sizeof(ShmExampleStruct));
    shm_unlink(SHM_NAME);

```

```
    exit(EXIT_SUCCESS);
}
```

shm_reader.c

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

#define SHM_NAME "/shm_example"
#define INT_LIST_SIZE 10
#define MSG_MAX 100

typedef struct{
    int integer_list[INT_LIST_SIZE];
    char message[MSG_MAX];
} ShmExampleStruct;

int main(void) {
    int i, fd_shm;
    ShmExampleStruct *shm_struct = NULL;

    /* Open of the shared memory. */
    if ((fd_shm = shm_open(SHM_NAME, O_RDONLY, 0)) == -1) {
        perror("shm_open");
        exit(EXIT_FAILURE);
    }

    /* Mapping of the memory segment. */
    shm_struct = mmap(NULL, sizeof(ShmExampleStruct), PROT_READ,
        MAP_SHARED, fd_shm, 0);
    close(fd_shm);
    if (shm_struct == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    printf("Pointer to shared memory segment: %p\n", (void*)shm_struct);

    /* Reading of the memory. */
    printf("%s\n" , shm_struct->message);
    for (i = 0; i < INT_LIST_SIZE; i++) {
        printf("%d\n", shm_struct->integer_list[i]);
    }

    /* Unmapping of the shared memory */
    munmap(shm_struct, sizeof(ShmExampleStruct));

    exit(EXIT_SUCCESS);
}
```

0.20 ptos. a) ¿Tendría sentido incluir `shm_unlink` en el lector? ¿Por qué?

0.20 ptos. b) ¿Tendría sentido incluir `ftruncate` en el lector? ¿Por qué?

0.20 ptos. c) ¿Cuál es la diferencia entre `shm_open` y `mmap`? ¿Qué sentido tiene que existan dos funciones diferentes?

0.20 ptos. d) ¿Se podría haber usado la memoria compartida sin enlazarla con `mmap`? Si es así, explicar cómo.

> ***Nota.*** *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Concurrencia en Memoria Compartida

Al compartir memoria es muy típico que surjan problemas de condiciones de carrera. Una manera de solucionar este tipo de problemas, aprovechando el uso de memoria compartida, es la utilización de semáforos anónimos creados con la función `sem_init`. Al estar visible la variable semáforo a todos los elementos que hacen uso de la misma (en este caso procesos), no haría falta identificar al semáforo con un nombre. Una vez terminado el uso del semáforo, y antes de liberar la memoria compartida, es necesario destruirlo usando `sem_destroy`.

Nota. *Con lo estudiado hasta el momento, ya se puede realizar el Ejercicio 7a) y el Ejercicio 7b).*

COLAS DE MENSAJES

Introducción a las Colas de Mensajes

Las **colas de mensajes** son otro de los recursos compartidos que pone Unix a disposición de los programas para que puedan intercambiarse información.

En la primera práctica se vio que dos procesos podían intercambiar información en forma de flujo continuo de caracteres a través de una tubería. Además de esto, Unix permite el intercambio de fragmentos discretos de información, o mensajes. Uno de los mecanismos para conseguirlo es la cola de mensajes. Los procesos introducen mensajes en la cola, que se van almacenando. Cuando un proceso extrae un mensaje de la cola, extrae el primer mensaje que se introdujo (*first in, first out*), 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 tres elementos:

1. Mensajes: son similares a las cartas que se envían por correo, y por tanto contienen la información que se desea transmitir entre los procesos.
2. 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.

En el caso de colas de prioridad, cada mensaje lleva asociada una prioridad, de modo que los mensajes de mayor prioridad se leerán antes que los de menor prioridad, incluso aunque sean posteriores.

Funciones para el Manejo de Colas de Mensajes

En sistemas Unix se puede trabajar con colas de mensajes en C utilizando la API de POSIX (descrita en el manual de `mq_overview`), que proporciona entre otras las siguientes funciones:

- `mq_open`: crea o abre una cola de mensajes, y devuelve un descriptor de cola de mensajes que hace referencia a la misma.
- `mq_send`: envía un mensaje a la cola.

- `mq_receive`: recibe un mensaje de la cola.
- `mq_close`: cierra un descriptor de cola de mensajes.
- `mq_unlink`: elimina el nombre de una cola de mensajes. La cola en sí será eliminada cuando ningún proceso tenga un descriptor de cola de mensajes que se corresponda con dicha cola.

Estas funciones están incluidas en los ficheros de cabecera `<mqueue.h>`, `<sys/stat.h>` y `<fcntl.h>`. Además, será necesario enlazar el programa con la biblioteca de tiempo real, es decir, se debe añadir a la llamada al compilador `gcc` el parámetro `-lrt`.

Aunque el uso de estas funciones se encuentra detallado en la página correspondiente del manual, a continuación se describirán brevemente.

Creación y Eliminación de Colas de Mensajes

Para crear o abrir una cola de mensajes se utiliza la función `mq_open`, análoga a `open` pero para colas de mensajes. El primer parámetro, `name`, es una cadena que debe comenzar por el carácter '`/`' y no tener ningún carácter '`/`' adicional, por ejemplo `"/ejemplo"`. Los parámetros `oflag` y `mode` son similares a los de `open`. En concreto, `oflag` requiere que al menos se especifique `O_RDONLY`, `O_WRONLY` o `O_RDWR`. El primero indicará que la cola de mensajes solo permitirá recibir mensajes, el segundo que solo podrá enviarlos y el tercero indica que ambas operaciones están permitidas. Adicionalmente, se podrán usar las banderas `O_CREAT`, para crear la cola de mensajes si no existe, `O_EXCL`, que en combinación con `O_CREAT` reportará error si la cola ya existía, y `O_NONBLOCK`, que hace que las operaciones de envío o recepción de mensajes que fueran a bloquearse por falta de espacio en cola o de mensajes a recibir, devuelvan error en lugar de bloquearse. Si se especifica `O_CREAT` y se crea una nueva cola, entonces el parámetro `mode` especificará los permisos de la cola de mensajes creada, de manera similar a `open`. En ese caso, el parámetro `attr` especificará los atributos de la cola a crear. Este parámetro debe ser un puntero a una estructura `mq_attr`, que tiene la siguiente definición:

```
struct mq_attr {
    long mq_flags;   /* Flags (ignored for mq_open()) */
    long mq_maxmsg;  /* Max. # of messages on queue */
    long mq_msgsize; /* Max. message size (bytes) */
    long mq_curmsgs; /* # of messages currently in queue
                        (ignored for mq_open()) */
};
```

El parámetro `mq_maxmsg` será el número máximo de mensajes en la cola. En una instalación por defecto de Linux, este valor solo puede establecerse como máximo a `10`, así que este será el valor que se usará. El parámetro `mq_msgsize` es el tamaño de mensaje máximo permitido en la cola. Las colas de mensajes permiten mensajes de tamaño variable, pero se producirá un error si se intenta enviar un mensaje de mayor tamaño que el especificado aquí, o si el buffer usado en la recepción es de menor tamaño al especificado por este parámetro. Los otros dos parámetros no se usan en la creación de la cola, y pueden dejarse a `0`. El valor retornado por esta función es un descriptor de cola de mensajes haciendo referencia a la cola abierta o creada. En caso de error devuelve `(mqd_t)-1`.

Para cerrar una cola de mensajes se utiliza la función `mq_close`, que recibe como argumento `mqdes` el descriptor de la cola de mensajes a liberar.

Por último, para borrar el nombre de la cola de mensajes se usa la función `mq_unlink`, análoga a `unlink`, que recibe como argumento `name` el nombre que se empleó en `mq_open` y que se desea borrar.

Envío y Recepción de Mensajes

Las dos operaciones fundamentales de las colas de mensajes son el envío y la recepción de mensajes.

Para enviar mensajes se usa la función `mq_send`, cuyo primer argumento es el descriptor de la cola a la que se va a enviar. El segundo parámetro, `msg_ptr`, apunta al mensaje, cuya longitud está dada por el tercer argumento, `msg_len` (debe ser menor o igual al tamaño de mensaje máximo especificado en la creación de la cola). Finalmente, la prioridad se indica en `msg_prio`. Si el mensaje no fuera una cadena de caracteres deberá hacerse casting para enviarlo.

Para recibir mensajes se usa la función `mq_receive`, que recibe como primer parámetro el descriptor de la cola de la que se va a recibir, `mqdes`. El mensaje se escribirá en el buffer `msg_ptr`, que tiene longitud `msg_len` (debe ser mayor o igual al tamaño de mensaje máximo especificado en la creación de la cola). Si `msg_prio` no es `NULL` la prioridad del mensaje recibido se escribirá en la dirección de memoria a la que apunte. Si el buffer no fuera una cadena de caracteres, deberá hacerse casting.

Nota. *Dado que* `mq_send` *y* `mq_receive` *son llamadas bloqueantes (salvo que se indique lo contrario en la creación de la cola), uno de sus posibles errores es* `EINTR`*, que se produce en el caso de que haya un manejador de señal instalado y se reciba la señal armada mientras el proceso está bloqueado esperando a que haya espacio en cola o mensajes que recibir.*

0.30 ptos.

Ejercicio 5: Envío y Recepción de Mensajes en Colas. Dado el siguiente código en C:

mq_send_receive.c

```c
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_MESSAGE 1024
#define MQ_NAME "/mq_example"
#define N_MESSAGES 6

int main(void) {
    char aux[MAX_MESSAGE];
    int i;
    struct mq_attr attributes;
    mqd_t mq;
    unsigned int priors[N_MESSAGES] = {1, 1, 1, 2, 1, 3}, prior;

    attributes.mq_maxmsg = 10;
    attributes.mq_msgsize = MAX_MESSAGE;

    if ((mq = mq_open(MQ_NAME, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR, &
        attributes)) == (mqd_t)-1) {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }
    mq_unlink(MQ_NAME);

    for (i = 0; i < N_MESSAGES; i++) {+
        sprintf(aux, "Message %d", i + 1);
        printf("Sending:  %s\n", aux);
        if (mq_send(mq, aux, strlen(aux) + 1, priors[i]) == -1) {
```

```
            perror("mq_send");
            mq_close(mq);
            exit(EXIT_FAILURE);
        }
    }
    printf("\n");
    for (i = 0; i < N_MESSAGES; i++) {
        if (mq_receive(mq, aux, MAX_MESSAGE, &prior) == -1) {
            perror("mq_receive");
            mq_close(mq);
            exit(EXIT_FAILURE);
        }
        printf("Received: %s (Priority: %u)\n", aux, prior);
    }

    mq_close(mq);

    exit(EXIT_SUCCESS);
}
```

0.10 ptos. a) ¿En qué orden se envían los mensajes y en qué orden se reciben? ¿Por qué?

0.20 ptos. b) ¿Qué sucede si se cambia `O_RDWR` por `O_RDONLY`? ¿Y si se cambia por `O_WRONLY`?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Inspeccionar Colas de Mensajes en Linux

Aunque POSIX no especifica ninguna forma estándar para listar las colas creadas utilizando `mq_open`, cada sistema Unix suele implementar alguna. En el caso de Linux, se realiza montando un sistema de archivos virtual en `/dev/mqueue`. Por tanto, en `/dev/mqueue` se pueden ver las colas creadas, y usar el comando `rm` para borrarlas.

Uso de Colas de Mensajes

Ejemplo

0.80 ptos. **Ejercicio 6: Colas de Mensajes.** Dado el siguiente código en C:

`mq_sender.c`

```
/* SENDER */
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#define MQ_NAME "/mq_example"
#define N 33

typedef struct {
    int valor;
    char aviso[80];
} Mensaje;
```

```

int main(void) {
    struct mq_attr attributes = {
        .mq_flags = 0,
        .mq_maxmsg = 10,
        .mq_curmsgs = 0,
        .mq_msgsize = sizeof(Mensaje)
    };

    mqd_t queue = mq_open(MQ_NAME,
        O_CREAT | O_WRONLY, /* This process is only going to send messages
            */
        S_IRUSR | S_IWUSR, /* The user can read and write */
        &attributes);

    if (queue == (mqd_t)-1) {
        fprintf(stderr, "Error opening the queue\n");
        return EXIT_FAILURE;
    }

    Mensaje msg;
    msg.valor = 29;
    strcpy(msg.aviso, "Hola a todos");

    if (mq_send(queue, (char *)&msg, sizeof(msg), 1) == -1) {
        fprintf(stderr, "Error sending message\n");
        return EXIT_FAILURE;
    }

    /* Wait for input to end the program */
    fprintf(stdout, "Press any key to finish\n");
    getchar();

    mq_close(queue);
    mq_unlink(MQ_NAME);

    return EXIT_SUCCESS;
}
```

mq_receptor.c

```
/* RECECPTOR */
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

#define MQ_NAME "/mq_example"
#define N 33

typedef struct {
    int valor;
    char aviso[80];
} Mensaje;

int main(void) {
    struct mq_attr attributes = {
        .mq_flags = 0,
```

```
19          .mq_maxmsg = 10,
20          .mq_curmsgs = 0,
21          .mq_msgsize = sizeof(Mensaje)
22      };
23
24      mqd_t queue = mq_open(MQ_NAME,
25          O_CREAT | O_RDONLY, /* This process is only going to send messages
                */
26          S_IRUSR | S_IWUSR, /* The user can read and write */
27          &attributes);
28
29      if(queue == (mqd_t)-1) {
30          fprintf(stderr, "Error opening the queue\n");
31          return EXIT_FAILURE;
32      }
33
34      Mensaje msg;
35
36      if (mq_receive(queue, (char *)&msg, sizeof(msg), NULL) == -1) {
37          fprintf(stderr, "Error receiving message\n");
38          return EXIT_FAILURE;
39      }
40
41      printf("%d: %s\n", msg.valor, msg.aviso);
42
43      /* Wait for input to end the program */
44      fprintf(stdout, "Press any key to finish\n");
45      getchar();
46
47      mq_close(queue);
48
49      return EXIT_SUCCESS;
50  }
```

0.20 ptos. a) Ejecutar el código del emisor, y después el del receptor. ¿Qué sucede? ¿Por qué?

0.20 ptos. b) Ejecutar el código del receptor, y después el del emisor. ¿Qué sucede? ¿Por qué?

0.20 ptos. c) Repetir las pruebas anteriores creando la cola de mensajes como no bloqueante. ¿Qué sucede ahora?

0.20 ptos. d) Si hubiera más de un receptor en el sistema, ¿sería adecuado sincronizar los accesos a la cola usando semáforos? ¿Por qué?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Nota. *Con lo estudiado hasta el momento, ya se puede realizar el Ejercicio 7c).*

EJERCICIO DE CODIFICACIÓN

7.00 ptos. **Ejercicio 7: Streaming.** Se pretende diseñar un *streaming* de datos distribuido en procesos que haga uso de: (a) memoria compartida para el almacenamiento de los datos, y (b) colas de mensajes para el paso de instrucciones entre procesos. Uno de los procesos (*stream-server*) se encargará de obtener datos desde un fichero y almacenar estos datos en memoria compartida. El otro (*stream-client*) se encargará de obtener los datos desde memoria compartida e imprimirlos a un archivo. Estos procesos serán iniciados por un

tercer programa (*stream-ui*), que además se encargará de recibir las instrucciones del usuario por consola y transmitirá estas instrucciones a los procesos *stream-server* y *stream-client* utilizando colas de mensajes.

Las instrucciones que recibirá *stream-ui* por consola son las siguientes:

- post: Realizará el flujo de datos desde un archivo hacia memoria compartida.
- get: Realizará la impresión de datos desde memoria compartida a un fichero de salida.
- exit: Terminará la ejecución de todos los procesos.

En concreto, se deberán satisfacer los siguientes requisitos.

2.50 ptos.

a) Memoria compartida:

- Escribir un programa en C ("`stream-ui.c`") que:
  - Cree un segmento de memoria compartida al comienzo del programa. El segmento deberá almacenar una estructura con:
    - Un buffer de tamaño igual a 5 caracteres.
    - Dos enteros, `post_pos` y `get_pos`.

    Los índices `post_pos` y `get_pos` se inicializarán a `0` y servirán para almacenar respectivamente el último byte escrito y leído dentro del buffer.
  - Lance dos procesos (*stream-server* y *stream-client*) haciendo uso de `fork` y `exec`.
  - Espere a la finalización de los procesos *stream-server* y *stream-client*.
- Escribir dos programas en C ("`stream-server.c`" y "`stream-client.c`") que utilicen el segmento de memoria compartida creado por *stream-ui* como un buffer circular de almacenamiento y lectura de datos.
  - El programa *stream-server* recibirá como argumento el archivo desde el que leer para hacer *stream* de datos a memoria compartida.
  - El programa *stream-client* recibirá como argumento el archivo de salida en el que imprimirá los caracteres que lea de memoria compartida.
  - El programa *stream-ui* recibirá ambos archivos como argumento para utilizarlos en la llamada a `exec` de los procesos *stream-server* y *stream-client*.
  - El proceso *stream-server* deberá leer uno a uno los caracteres desde el archivo recibido como argumento, escribirlos en el buffer de memoria compartida en la posición `post_pos`, y avanzar el valor de `post_pos` convenientemente. En caso de que llegue al final del buffer de datos, deberá seguir almacenando caracteres por el principio del buffer, como si se tratara de un buffer circular. Una vez alcanzado el final del fichero, deberá añadir al buffer `'\0'` y terminar su ejecución.
  - El proceso *stream-client* deberá leer uno a uno los caracteres de la posición `get_pos` desde el segmento de memoria compartida, e imprimir, en el fichero recibido como primer argumento, el carácter leído. En caso de que llegue al final del buffer de datos, deberá seguir leyendo caracteres por el principio del buffer, como si se tratara de un buffer circular. En caso de leer del buffer el carácter `'\0'`, terminará su ejecución.

Nota. *Se puede ver cómo se va rellenando el contenido del fichero de salida mediante el comando* `tail -f <ruta-al-fichero>`.

2.00 ptos.

b) Semáforos:

- Con el fin de evitar accesos concurrentes a memoria, y sincronizar convenientemente ambos procesos (*stream-server* y *stream-client*), se deben controlar los accesos al buffer mediante semáforos almacenados en memoria compartida.

Para ello, se usará el algoritmo de productor–consumidor:

```
Productor {
    Down(sem_empty);
    Down(sem_mutex);
    AñadirElemento();
    Up(sem_mutex);
    Up(sem_fill);
}

Consumidor {
    Down(sem_fill);
    Down(sem_mutex);
    ExtraerElemento();
    Up(sem_mutex);
    Up(sem_empty);
}
```

- Se deberán incluir en la estructura de memoria compartida tres semáforos sin nombre para implementar el algoritmo de productor–consumidor.
- Se usará la función `sem_timedwait` en cada operación con semáforos, de manera que *stream-server* y *stream-client* queden bloqueado un máximo de 2 s cuando la cola esté llena o vacía, respectivamente. Tras esta espera, desecharán la operación indicándolo por pantalla.

2.50 ptos.

c) Colas de instrucciones:

- Modificar el programa *stream-ui* de manera que:
  - Cree dos colas de mensajes, una destinada para *stream-server* y otra para *stream-client*, que permitan enviar mensajes con las instrucciones descritas anteriormente.
  - Quede a la espera de instrucciones por la entrada estándar y las redirija a los procesos destinatarios (*stream-server* y *stream-client*) a través de la cola de mensajes correspondiente.
  - Libere recursos y finalice correctamente la ejecución de todos los procesos cuando reciba la instrucción exit.
- Modificar los programas *stream-server* y *stream-client* de manera que:
  - Reciban instrucciones de la cola de mensajes.
  - El proceso *stream-server* deberá recibir las instrucciones post y exit. Cuando reciba post deberá leer el siguiente carácter del fichero de entrada, escribirlo en el buffer, y volver a leer instrucciones de la cola. En el caso de alcanzar el final del fichero de entrada, escribirá en el buffer `'\0'`, e ignorará las siguientes instrucciones post.
  - El proceso *stream-client* deberá recibir las instrucciones get y exit. Cuando reciba get deberá leer el siguiente carácter del buffer, escribirlo en el fichero de salida, y volver a leer instrucciones de la cola. En caso de leer del buffer `'\0'`, ignorará las siguientes instrucciones get.
  - Ambos procesos, cuando reciban la instrucción exit, deberán terminar correctamente su ejecución.

Nota. *Para comprobar que el sistema sigue siendo capaz de funcionar a pleno rendimiento con las colas implementadas, se puede redirigir la entrada con un* pipeline *del estilo* `cat commands.txt | ./stream-ui input.txt ouptut.txt`*, donde el fichero* `commands.txt` *contenga líneas alternas con las instrucciones* `post` *y* `get`*.*