

# Práctica 3

**FECHA DE ENTREGA: DEL 7 DE ABRIL AL 11 DE ABRIL**  
**(HORA LÍMITE: ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS)**

## Memoria compartida

- Introducción a la memoria compartida
- Funciones para el manejo de memoria compartida
- Inspeccionar memoria en Linux
- Uso de memoria compartida

## Colas de mensajes

- Introducción a las colas de mensajes
- Funciones para el manejo de colas de mensajes
- Inspeccionar colas de mensajes en Linux
- Uso de colas de mensajes

## Ejercicio de codificación

***Nota.** La práctica consta de dos tipos de ejercicios diferentes. Los ejercicios cortos son ejercicios de aprendizaje sin puntuación asociada, que no es necesario entregar porque se evaluarán a través del examen correspondiente de evaluación continua. La calificación de la entrega corresponderá únicamente al ejercicio de codificación, incluyendo tanto el código como la documentación.*

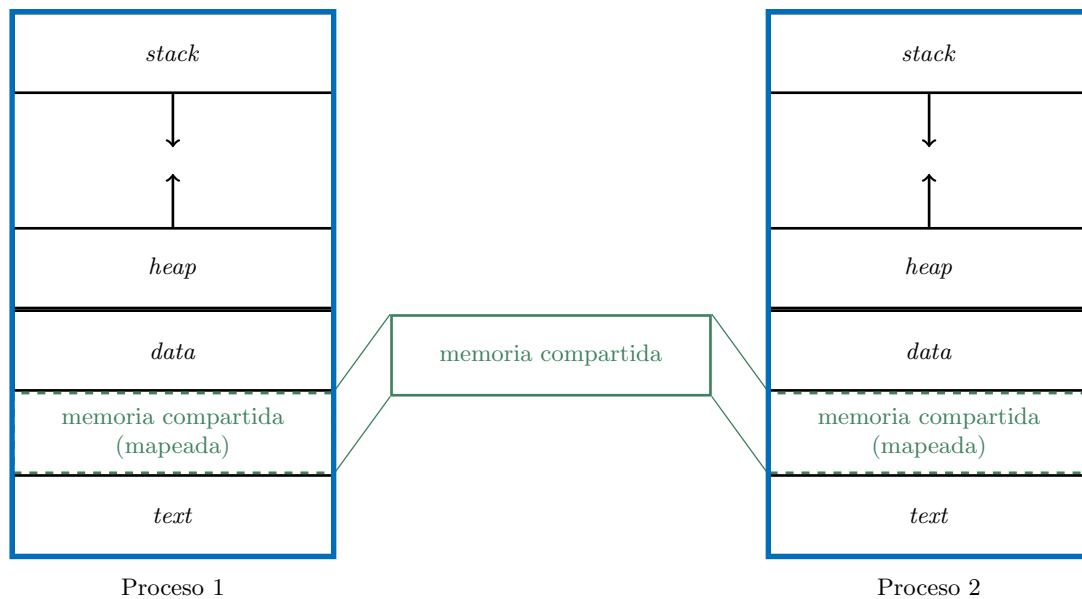
## Memoria compartida

### 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.



**Figura 1:** Memoria compartida entre dos procesos.

## 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.
- `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 los 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 `"/shm_example"`. 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 indica 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 tamaño del segmento a cero si 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.

### Ejercicio 1: Creación de memoria compartida. Dado el siguiente código en C:

shm\_open.c

```
1 /* This is only a code fragment, not a complete program... */
2 fd_shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
3 if (fd_shm == -1) {
4     if (errno == EEXIST) {
5         fd_shm = shm_open(SHM_NAME, O_RDWR, 0);
6         if (fd_shm == -1) {
7             perror("Error opening the shared memory segment");
8             exit(EXIT_FAILURE);
9         } else {
10            printf("Shared memory segment open\n");
11        }
12    } else {
13        perror("Error creating the shared memory segment\n");
14        exit(EXIT_FAILURE);
15    }
16 } else {
17     printf("Shared memory segment created\n");
18 }
```

- ¿En qué consiste este código? ¿Qué sentido tiene utilizarlo para abrir un objeto de memoria compartida?
- En un momento dado se deseará forzar (en la próxima ejecución del programa) la inicialización del objeto de memoria compartida `SHM_NAME`. ¿Qué posibles soluciones, en código C o fuera de él, hay para forzar dicha inicialización?

## 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.

### Ejercicio 2: Tamaño de ficheros. Dado el siguiente código en C:

file\_truncate.c

```
1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/stat.h>
5 #include <unistd.h>
6
7 #define SHM_NAME "/shm_example"
8 #define MESSAGE "Test message"
9
10 int main(int argc, char *argv[]) {
11     int fd;
12     struct stat statbuf;
13
14     if (argc != 2) {
15         fprintf(stderr, "Usage: %s <FILE>\n", argv[0]);
16         exit(EXIT_FAILURE);
17     }
18
19     fd = open(argv[1], O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
20     if (fd == -1) {
21         perror("open");
22         exit(EXIT_FAILURE);
23     }
24     dprintf(fd, "%s", MESSAGE);
25     /* Get size of the file. */
26
27     /* Truncate the file to size 5. */
28
29     close(fd);
30     exit(EXIT_SUCCESS);
31 }
```

- Completar el código anterior para obtener el tamaño del fichero abierto.
- Completar el código anterior para truncar el tamaño del fichero a 5 B. ¿Qué contiene el fichero resultante?

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

## 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
<code>PROT_READ</code>	Permiso de lectura
<code>PROT_WRITE</code>	Permiso de escritura
<code>PROT_EXEC</code>	Permiso de ejecución
<code>PROT_NONE</code>	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 al 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 deben 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`.

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

file\_mmap.c

```

1  /* READER */
2  #include <errno.h>
3  #include <fcntl.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <sys/mman.h>
7  #include <sys/stat.h>
8  #include <unistd.h>
9
10 #define FILE_NAME "test_file.dat"
11
12 int main(void) {
13     int created = 0;
14     int fd;
```

```

15  int *mapped = NULL;
16
17  if ((fd = open(FILE_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR))
    == -1) {
18      if (errno == EEXIST) {
19          if ((fd = open(FILE_NAME, O_RDWR, 0)) == -1) {
20              perror("open");
21              exit(EXIT_FAILURE);
22          }
23      } else {
24          perror("open");
25          exit(EXIT_FAILURE);
26      }
27  } else {
28      if (ftruncate(fd, sizeof(int)) == -1) {
29          perror("ftruncate");
30          close(fd);
31          exit(EXIT_FAILURE);
32      }
33      created = 1;
34  }
35
36  if ((mapped = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED
    , fd, 0)) == MAP_FAILED) {
37      perror("mmap");
38      close(fd);
39      exit(EXIT_FAILURE);
40  }
41  close(fd);
42
43  if (created) {
44      *mapped = 0;
45  }
46  *mapped += 1;
47  printf("Counter: %d\n", *mapped);
48  munmap(mapped, sizeof(int));
49
50  exit(EXIT_SUCCESS);
51 }

```

- ¿Qué sucede cuando se ejecuta varias veces el programa anterior? ¿Por qué?
- ¿Se puede leer el contenido del fichero «test\_file.dat» con un editor de texto? ¿Por qué?

## 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 mapas 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 descriptors 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.

### Notas.

- 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 reabrirlos en cada uno de ellos.
- 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

**Ejercicio 4: Memoria compartida.** Dado el siguiente código en C:

shm\_rdwr.h

```
1 #define SHM_NAME "/shm_example"
2 #define INT_LIST_SIZE 10
3 #define MSG_MAX 100
4
5 typedef struct {
6     int integer_list[INT_LIST_SIZE];
7     char message[MSG_MAX];
8 } ShmExampleStruct;
```

shm\_writer.c

```
1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/mman.h>
6 #include <sys/stat.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9
10 #include "shm_rdwr.h"
11
12 #define MESSAGE "Hello world shared memory!"
13
14 int main(void) {
15     int i, fd_shm;
16     ShmExampleStruct *shm_struct = NULL;
17
18     /* Creation of the shared memory. */
19     if ((fd_shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRUSR |
20         S_IWUSR)) == -1) {
21         perror("shm_open");
22         exit(EXIT_FAILURE);
23     }
24
25     /* Resize of the memory segment. */
26     if (ftruncate(fd_shm, sizeof(ShmExampleStruct)) == -1) {
27         perror("ftruncate");
28         shm_unlink(SHM_NAME);
29         exit(EXIT_FAILURE);
30     }
31
32     /* Mapping of the memory segment. */
33     shm_struct = mmap(NULL, sizeof(ShmExampleStruct), PROT_READ | PROT_WRITE
34         , MAP_SHARED, fd_shm, 0);
35     close(fd_shm);
36     if (shm_struct == MAP_FAILED) {
37         perror("mmap");
38         shm_unlink(SHM_NAME);
39         exit(EXIT_FAILURE);
40     }
41     printf("Pointer to shared memory segment: %p\n", (void *)shm_struct);
42
43     /* Initialization of the memory. */
44     memcpy(shm_struct->message, MESSAGE, sizeof(MESSAGE));
```



```

43     for (i = 0; i < INT_LIST_SIZE; i++) {
44         shm_struct->integer_list[i] = i;
45     }
46
47     /* The daemon executes until some character is pressed. */
48     getchar();
49
50     /* Unmapping and freeing of the shared memory */
51     munmap(shm_struct, sizeof(ShmExampleStruct));
52     shm_unlink(SHM_NAME);
53
54     exit(EXIT_SUCCESS);
55 }

```

#### shm\_reader.c

```

1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <unistd.h>
6
7  #include "shm_rdwr.h"
8
9  int main(void) {
10     int i, fd_shm;
11     ShmExampleStruct *shm_struct = NULL;
12
13     /* Open of the shared memory. */
14     if ((fd_shm = shm_open(SHM_NAME, O_RDONLY, 0)) == -1) {
15         perror("shm_open");
16         exit(EXIT_FAILURE);
17     }
18
19     /* Mapping of the memory segment. */
20     shm_struct = mmap(NULL, sizeof(ShmExampleStruct), PROT_READ, MAP_SHARED,
21         fd_shm, 0);
22     close(fd_shm);
23     if (shm_struct == MAP_FAILED) {
24         perror("mmap");
25         exit(EXIT_FAILURE);
26     }
27     printf("Pointer to shared memory segment: %p\n", (void *)shm_struct);
28
29     /* Reading of the memory. */
30     printf("%s\n", shm_struct->message);
31     for (i = 0; i < INT_LIST_SIZE; i++) {
32         printf("%d\n", shm_struct->integer_list[i]);
33     }
34
35     /* Unmapping of the shared memory */
36     munmap(shm_struct, sizeof(ShmExampleStruct));
37
38     exit(EXIT_SUCCESS);
39 }

```

- ¿Tendría sentido incluir `shm_unlink` en el lector? ¿Por qué?
- ¿Tendría sentido incluir `ftruncate` en el lector? ¿Por qué?
- ¿Cuál es la diferencia entre `shm_open` y `mmap`? ¿Qué sentido tiene que existan dos funciones diferentes?

d) ¿Se podría haber usado la memoria compartida sin enlazarla con `mmap`? Si es así, ¿cómo se haría?

### 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`. Como la variable semáforo es visible para todos los elementos que hacen uso de la memoria (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 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 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 `<mqqueue.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 `"/mq_example"`. 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:

```
1 struct mq_attr {
2     long mq_flags;    /* Flags (ignored for mq_open()) */
3     long mq_maxmsg;   /* Max. # of messages on queue */
4     long mq_msgsize;  /* Max. message size (bytes) */
5     long mq_curmsgs;  /* # of messages currently in queue
6                       (ignored for mq_open()) */
7 };
```

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 para enviar o mensajes que recibir.

**Ejercicio 5: Envío y recepción de mensajes en colas.** Dado el siguiente código en C:

`mq_send_receive.c`

```
1 #include <mqueue.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 #define MAX_MESSAGE 1024
7 #define MQ_NAME "/mq_example"
8 #define N_MESSAGES 6
9
10 int main(void) {
11     char aux[MAX_MESSAGE];
12     int i;
13     struct mq_attr attributes;
14     mqd_t mq;
15     unsigned int priors[N_MESSAGES] = {1, 1, 1, 2, 1, 3}, prior;
16
17     attributes.mq_maxmsg = 10;
18     attributes.mq_msgsize = MAX_MESSAGE;
19
20     if ((mq = mq_open(MQ_NAME, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR, &
21         attributes)) == (mqd_t)-1) {
22         perror("mq_open");
23     }
24 }
```

```

22     exit(EXIT_FAILURE);
23 }
24 mq_unlink(MQ_NAME);
25
26 for (i = 0; i < N_MESSAGES; i++) {
27     sprintf(aux, "Message %d", i + 1);
28     printf("Sending:  %s\n", aux);
29     if (mq_send(mq, aux, strlen(aux) + 1, priors[i]) == -1) {
30         perror("mq_send");
31         mq_close(mq);
32         exit(EXIT_FAILURE);
33     }
34 }
35 printf("\n");
36 for (i = 0; i < N_MESSAGES; i++) {
37     if (mq_receive(mq, aux, MAX_MESSAGE, &prior) == -1) {
38         perror("mq_receive");
39         mq_close(mq);
40         exit(EXIT_FAILURE);
41     }
42     printf("Received: %s (Priority: %u)\n", aux, prior);
43 }
44
45 mq_close(mq);
46
47 exit(EXIT_SUCCESS);
48 }

```

- ¿En qué orden se envían los mensajes y en qué orden se reciben? ¿Por qué?
- ¿Qué sucede si se cambia `O_RDWR` por `O_RDONLY`? ¿Y si se cambia por `O_WRONLY`?

## 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

**Ejercicio 6: Colas de mensajes.** Dado el siguiente código en C:

`mq_sd_rc.h`

```

1 #include <mqueue.h>
2
3 #define MQ_NAME "/mq_example"
4 #define N 33
5
6 typedef struct {
7     int value;
8     char warning[80];
9 } Message;

```

```

10
11 struct mq_attr attributes = {.mq_flags = 0,
12                             .mq_maxmsg = 10,
13                             .mq_curmsgs = 0,
14                             .mq_msgsize = sizeof(Message)};

```

#### mq\_sender.c

```

1 #include <fcntl.h>
2 #include <mqueue.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <sys/stat.h>
7
8 #include "mq_sd_rc.h"
9
10 int main(void) {
11     Message msg;
12     mqd_t queue;
13
14     queue = mq_open(
15         MQ_NAME,
16         O_CREAT | O_WRONLY, /* This process is going to send messages. */
17         S_IRUSR | S_IWUSR, /* The user can read and write. */
18         &attributes);
19
20     if (queue == (mqd_t)-1) {
21         fprintf(stderr, "Error opening the queue\n");
22         return EXIT_FAILURE;
23     }
24
25     msg.value = 29;
26     strcpy(msg.warning, "Hi all!");
27
28     if (mq_send(queue, (char *)&msg, sizeof(msg), 1) == -1) {
29         fprintf(stderr, "Error sending message\n");
30         return EXIT_FAILURE;
31     }
32
33     /* Wait for input to end the program. */
34     fprintf(stdout, "Press any key to finish\n");
35     getchar();
36
37     mq_close(queue);
38     mq_unlink(MQ_NAME);
39
40     return EXIT_SUCCESS;
41 }

```

#### mq\_receptor.c

```

1 #include <fcntl.h>
2 #include <mqueue.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/stat.h>
6
7 #include "mq_sd_rc.h"
8
9 int main(void) {

```

```

10 Message msg;
11 mqd_t queue;
12
13 queue = mq_open(
14     MQ_NAME,
15     O_CREAT | O_RDONLY, /* This process is going to receive messages. */
16     S_IRUSR | S_IWUSR, /* The user can read and write. */
17     &attributes);
18
19 if (queue == (mqd_t)-1) {
20     fprintf(stderr, "Error opening the queue\n");
21     return EXIT_FAILURE;
22 }
23
24 if (mq_receive(queue, (char *)&msg, sizeof(msg), NULL) == -1) {
25     fprintf(stderr, "Error receiving message\n");
26     return EXIT_FAILURE;
27 }
28
29 printf("%d: %s\n", msg.value, msg.warning);
30
31 /* Wait for input to end the program. */
32 fprintf(stdout, "Press any key to finish\n");
33 getchar();
34
35 mq_close(queue);
36
37 return EXIT_SUCCESS;
38 }

```

- Ejecutar el código del emisor, y después el del receptor. ¿Qué sucede? ¿Por qué?
- Ejecutar el código del receptor, y después el del emisor. ¿Qué sucede? ¿Por qué?
- Repetir las pruebas anteriores creando la cola de mensajes como no bloqueante. ¿Qué sucede ahora?
- Si hubiera más de un receptor en el sistema, ¿sería adecuado sincronizar los accesos a la cola usando semáforos? ¿Por qué?

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

## Ejercicio de codificación

10,00 ptos.

**Ejercicio 7: Sistema de monitoreo multiproceso.** Escribir dos programas en C para implementar un sistema de monitoreo de bloques del sistema Blockchain. El ejercicio se divide en varias partes diferenciadas:

1,00 ptos.

- Programa de monitoreo:
  - El programa se debe ejecutar con un parámetro:
 

```
./monitor <LAG>
```

 donde `monitor` es el nombre del ejecutable y `<LAG>` es el retraso en milisegundos entre cada monitorización.
  - Se deberán lanzar dos procesos correspondientes a este programa:
    - El primer proceso detectará que la memoria compartida no está creada y se convertirá en el proceso Comprobador, encargado de verificar los bloques

4,50 ptos.

- recibidos.
- El segundo proceso detectará que la memoria compartida está creada y se convertirá en el proceso **Monitor**, encargado de mostrar los resultados por pantalla.
- b) Comunicación desde **Comprobador** a **Monitor** usando productor-consumidor:
- El proceso **Comprobador**:
    - Creará e inicializará un segmento de memoria compartida.
    - A continuación, recibirá un bloque (a través de la cola de mensajes, ver Apartado c)) y lo comprobará, añadiéndole una bandera que indique si es correcto o no.
    - Una vez comprobado, lo introducirá en memoria compartida para que lo lea **Monitor**.
    - Realizará una espera de <LAG> milisegundos.
    - Repetirá el proceso de recepción/comprobación/escritura hasta que reciba un bloque especial que indique la finalización del sistema.
    - Cuando reciba el bloque de finalización, lo introducirá en memoria compartida para notificar al **Monitor** de la finalización del sistema, liberará los recursos y terminará.
  - El proceso **Monitor**:
    - Abrirá el segmento de memoria compartida.
    - Extraerá un bloque.
    - Lo mostrará por pantalla, con la sintaxis "**Solution accepted: %08ld --> %08ld**" (el primer número es el objetivo y el segundo la solución) para los bloques correctos, y "**Solution rejected: %08ld !-> %08ld**" para los incorrectos.
    - Realizará una espera de <LAG> milisegundos.
    - Repetirá el ciclo de extracción/muestra hasta que reciba un bloque especial que indique la finalización del sistema.
    - Cuando reciba el bloque de finalización, liberará los recursos y terminará.
  - La introducción/extracción de bloques en memoria compartida seguirá el esquema 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 garantizará que no se pierde ningún bloque con independencia del retraso que se introduzca en cada proceso.
- Se usará un buffer circular de 6 bloques en el segmento de memoria compartida.



- Se alojarán en el segmento de memoria compartida tres semáforos sin nombre para implementar el algoritmo de productor–consumidor.

***Nota.** Se puede comprobar que este apartado es correcto enviando bloques aleatorios generados por el proceso **Comprobador**.*

3,50 ptos.

c) Programa de minería:

- El programa se debe ejecutar con dos parámetros:

```
./miner <ROUNDS> <LAG>
```

donde `miner` es el nombre del ejecutable, `<ROUNDS>` el número de rondas que se va a realizar y `<LAG>` es el retraso en milisegundos entre cada ronda.

- El proceso resultante de ejecutar este programa, **Minero**:
  - Creará una cola de mensajes con capacidad para 7 mensajes.
  - Establecerá un objetivo inicial fijo para la POW (por ejemplo, 0).
  - Para cada ronda, resolverá la POW; en el caso de que no se tenga un minero funcional capaz de generar una secuencia de bloques encadenados, se puede utilizar el fichero con la secuencia de 200 rondas proporcionado para «falsear» el minero.
  - Enviará un mensaje por la cola de mensajes que contenga, al menos, el objetivo y la solución hallada.
  - Realizará una espera de `<LAG>` milisegundos.
  - Establecerá como siguiente objetivo la solución anterior, y repetirá la ronda.
  - Una vez terminadas las rondas, enviará un bloque especial con algún código que permita saber al proceso **Comprobador** que el sistema está finalizando.
  - Liberará los recursos y terminará.

1,00 ptos.

d) En base al funcionamiento del sistema anterior:

- ¿Es necesario un sistema tipo productor–consumidor para garantizar que el sistema funciona correctamente si el retraso de **Comprobador** es mucho mayor que el de **Minero**? ¿Por qué?
- ¿Y si el retraso de **Comprobador** es muy inferior al de **Minero**? ¿Por qué?
- ¿Se simplificaría la estructura global del sistema si entre **Comprobador** y **Monitor** hubiera también una cola de mensajes? ¿Por qué?

El esquema global del programa se resume en la Figura 2.

Se deberán tener en cuenta los siguientes aspectos: (a) entrega de la documentación en formato .pdf requerida en la normativa de prácticas y con las respuestas a las cuestiones planteadas, (b) control de errores, (c) eliminación de la cola, (d) eliminación del segmento de memoria compartida, y (e) garantizar que no se producen bloqueos ni se pierden mensajes con independencia del retraso introducido en cada proceso.

A continuación se muestran dos ejemplos de ejecución del sistema final. En el primer ejemplo, el proceso **Minero** será más rápido que **Monitor**, que será más rápido que **Comprobador**:

<pre>\$ ./miner 15 0 [21901] Generating blocks... [21901] Finishing \$</pre>	<pre>\$ ./monitor 500 [21902] Checking blocks... [21902] Finishing \$</pre>	<pre>\$ ./monitor 100 [21903] Printing blocks... Solution accepted: 00000000 --&gt; 38722988 Solution accepted: 38722988 --&gt; 82781454 Solution accepted: 82781454 --&gt; 59403743 Solution accepted: 59403743 --&gt; 44638907 Solution accepted: 44638907 --&gt; 98780967 Solution accepted: 98780967 --&gt; 49574592 Solution accepted: 49574592 --&gt; 16391022 Solution accepted: 16391022 --&gt; 41194344 Solution accepted: 41194344 --&gt; 61259182 Solution accepted: 61259182 --&gt; 18852291 Solution accepted: 18852291 --&gt; 74660642 Solution accepted: 74660642 --&gt; 59894312 Solution accepted: 59894312 --&gt; 55978326 Solution accepted: 55978326 --&gt; 18633092 Solution accepted: 18633092 --&gt; 61031588 [21903] Finishing</pre>
--	---	--

En el segundo ejemplo, el proceso Comprobador será más rápido que Minero, que será más rápido que Monitor:

<pre>\$ ./miner 8 100 [21942] Generating blocks... [21942] Finishing \$</pre>	<pre>\$ ./monitor 0 [21943] Checking blocks... [21943] Finishing \$</pre>	<pre>\$ ./monitor 500 [21945] Printing blocks... Solution accepted: 00000000 --&gt; 38722988 Solution accepted: 38722988 --&gt; 82781454 Solution accepted: 82781454 --&gt; 59403743 Solution accepted: 59403743 --&gt; 44638907 Solution accepted: 44638907 --&gt; 98780967 Solution accepted: 98780967 --&gt; 49574592 Solution accepted: 49574592 --&gt; 16391022 Solution accepted: 16391022 --&gt; 41194344 [21945] Finishing</pre>
---	---	--

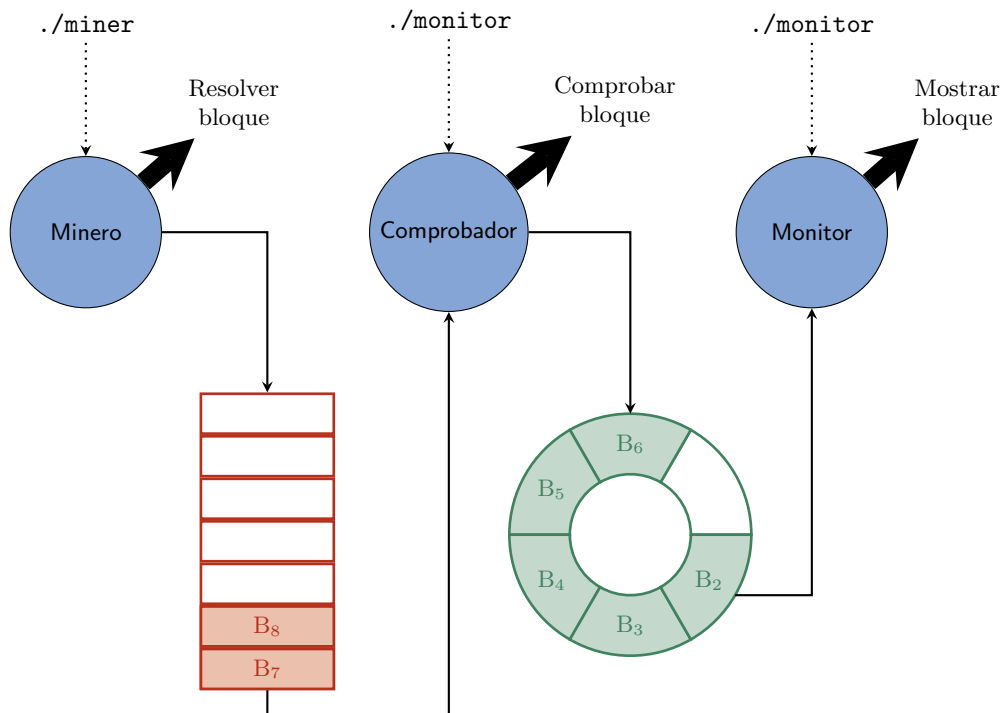


Figura 2: Esquema global del funcionamiento del sistema de monitoreo.