Sistemas Operativos - Práctica 3

FECHA DE ENTREGA:

DEL 8 AL 12 ABRIL (HORA LÍMITE DE ENTREGA: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS)

La tercera práctica se va a desarrollar en 2 semanas con el siguiente cronograma:

Semana 1: Memoria compartida

Semana 2: Colas de mensajes

Los ejercicios correspondientes a esta tercera 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 no puede ser accedida desde otro proceso.

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 en sistemas Unix usaremos 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 asociada al fichero o segmento de memoria asociado a un descriptor de fichero. Concretamente, lo usaremos 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>` .

La sintaxis de las funciones anteriores es la siguiente:

- *`int shm_open(const char *name, int oflag, mode_t mode);`*

  Es análoga a open pero para segmentos de memoria compartida. `name` debe comenzar por el caracter / y no tener ningún caracter / adicional, por ejemplo "/ejemplo". Los parámetros `oflag` y `mode` son similares a los de `open`. `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 escritura. Adicionalmente, se podrán usar los flag `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 0 si ya existía.

  Si se especifica el flag `O_CREAT` y se crea un nuevo segmento de memoria, entonces el parámetro `mode` especificará los permisos del segmento creado, de manera similar a `open`.

  El valor retornado por esta función es un descriptor de fichero haciendo referencia al segmento de memoria compartida abierto/creado. En caso de error devuelve -1.

- *`int ftruncate(int fd, off_t length);`*

  En esta función `fd` debe ser un descriptor de fichero válido. En nuestro caso, será el descriptor devuelto por `smh_open`. `length` será el tamaño que se le quiera dar a la memoria. Esta función devuelve 0 en caso de éxito y -1 si ocurre algún error.

  Solo es necesario llamar a esta función una vez, en el proceso que crea el segmento de memoria.

- *`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`*

  El parámetro `addr` será `NULL` para que el sistema operativo escoja una dirección de memoria apropiada para enlazar la memoria del fichero/segmento de memoria. `length` será el tamaño de la memoria que se desee enlazar (en nuestro caso, la memoria total del segmento). `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, flags que especifican información adicional. Es obligatorio especificar al menos uno, y solo uno, de los siguientes flags: `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. En nuestro caso, como queremos que todos los procesos vean la misma memoria, usaremos siempre `MAP_SHARED`. El parámetro `fd` es el descriptor de fichero cuyo contenido queremos enlazar a la memoria. En nuestro caso, es el descriptor retornado por `shm_open`. Por último `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. Dado que queremos 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 debemos 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 procesos. En caso de error `mmap` devolverá la constante `MAP_FAILED`.

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

  En esta función addr es la dirección original devuelta por `mmap`, y length el tamaño pasado a `mmap`.

  Esta función devuelve 0 si ha tenido éxito y -1 en caso de error.

- *`int close(int fd);`*

  Como se ha explicado en prácticas anteriores, `fd` es el descriptor de fichero a liberar, el cual puede ser liberado una vez realizado `mmap`.

  Esta función devuelve 0 si ha tenido éxito y -1 en caso de error.

- *`int shm_unlink(const char *name);`*

  En esta función `name` es el nombre asignado al segmento de memoria compartida que se empleó en `shm_open` y que se desea borrar.

  Esta función devuelve 0 si ha tenido éxito y -1 en caso de error.

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

En esta función `fd` es el descriptor de fichero del que se desea obtener la información, la cual se guardará en la estructura apuntada por `buf`. El campo más interesante de esta estructura es `st_size`, que nos dirá el tamaño del fichero o segmento de memoria.

Esta función devuelve 0 si ha tenido éxito y -1 en caso de error.

Nota

Las funciones anteriores se corresponden con la API más moderna del estándar POSIX para usar memoria compartida. En otros años se ha empleado la API de System V, usando las funciones `ftok`, `shmget`, `shmat`, `shmdt` y `shmctl`. La funcionalidad que ofrecen ambas es similar, pero la nueva API es más sencilla de usar.

Resumiendo, para usar la memoria compartida en sistemas Unix se ha de hacer lo siguiente:

- Obtener un descriptor de fichero a la memoria compartida con `shm_open`. El proceso que cree la memoria debe usar el flag `O_CREAT` y dar tamaño a la misma con `ftruncate`. Los demás deben averiguar el tamaño del segmento de memoria usando `fstat` en caso de no conocerlo de antemano.
- Enlazar la memoria al espacio de direcciones del proceso con `mmap`, y cerrar el descriptor de fichero con `close`.
- Usar la memoria compartida.
- 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

El proceso anterior puede 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 de uso:

Proceso 1

```
/**
 * @file
 *
 * @brief Código de ejemplo de memoria compartida, para un proceso que crea
 * la memoria.
 */
```

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

#define MAXBUF 10

int main(void)
{

      /* We create the shared memory */
      int fd_shm = shm_open(SHM_NAME,
                  O_RDWR | O_CREAT | O_EXCL, /* Create it and open for reading
                                                and writing */
                  S_IRUSR | S_IWUSR); /* The current user can read and write */
      if(fd_shm == -1)
      {
            fprintf (stderr, "Error creating the shared memory segment \n");
            return EXIT_FAILURE;
      }

      /* Resize the memory segment */
      int error = ftruncate(fd_shm, sizeof(ShmExampleStruct));
      if(error == -1)
      {
            fprintf (stderr, "Error resizing the shared memory segment \n");
            shm_unlink(SHM_NAME);
            return EXIT_FAILURE;
      }

      /* Map the memory segment */
      ShmExampleStruct * example_struct = mmap(NULL, sizeof(*example_struct),
                  PROT_READ | PROT_WRITE, MAP_SHARED, fd_shm, 0);
      if(example_struct == MAP_FAILED)
      {
            fprintf (stderr, "Error mapping the shared memory segment \n");
            shm_unlink(SHM_NAME);
            return EXIT_FAILURE;
      }

      printf("Pointer to shared memory segment: %p\n", (void*)example_struct);

      /* Initialize the memory */
      memcpy(example_struct->message, MESSAGE, sizeof(MESSAGE));
      for (int i = 0; i < MAXBUF; i++)
      {
            example_struct->integer_list[i] = i;
```

```
	}

	/* The daemon executes until press some character */
	getchar();

	/* Free the shared memory */
	munmap(example_struct, sizeof(*example_struct));
	shm_unlink(SHM_NAME);

	return EXIT_SUCCESS;
}
```

Proceso 2

```
/**
 * @file
 *
 * @brief Código de ejemplo de memoria compartida, para un proceso que usa
 * la memoria.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.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;

#define MAXBUF 10

int main(void)
{

	/* We open the shared memory */
	int fd_shm = shm_open(SHM_NAME,
			O_RDONLY, /* Obtain it and open for reading */
			0); /* Unused */
	if(fd_shm == -1)
	{
		fprintf (stderr, "Error opening the shared memory segment \n");
		return EXIT_FAILURE;
	}

	/* Map the memory segment */
	ShmExampleStruct * example_struct = mmap(NULL, sizeof(*example_struct),
			PROT_READ, MAP_SHARED, fd_shm, 0);
	if(example_struct == MAP_FAILED)
	{
		fprintf (stderr, "Error mapping the shared memory segment \n");
		return EXIT_FAILURE;
	}
```

```
	printf("Pointer to shared memory segment: %p\n", (void*)example_struct);

	/* Read the memory */
	printf("%s\n", example_struct->message);
	for (int i = 0; i < MAXBUF; i++)
	{
		printf("%d\n", example_struct->integer_list[i]);
	}

	/* Unmap the shared memory */
	munmap(example_struct, sizeof(*example_struct));

	return EXIT_SUCCESS;
}
```

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

```
int fd_shm = shm_open(SHM_NAME, O_RDWR | O_CREAT | O_EXCL,
		S_IRUSR | S_IWUSR);
if(fd_shm == -1)
{
	if(errno == EEXIST)
	{
		fprintf (stderr, "The shared memory segment already exists \n");
		fprintf (stderr, "Trying opening as a client \n");

		fd_shm = shm_open(SHM_NAME, O_RDWR, 0);
		if(fd_shm == -1)
		{
			fprintf (stderr, "Error opening the shared memory segment \n");
			return EXIT_FAILURE;
		}
	}
	else
	{
		fprintf (stderr, "Error creating the shared memory segment \n");
		return EXIT_FAILURE;
	}
}
else
{
	fprintf (stderr, "Shared memory segment created \n");
}
```

Inspeccionar segmentos de memoria existentes (solo Linux)

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 filesystem especial, que usa RAM en lugar de disco, en `/dev/shm`. Por tanto en `/dev/shm` podremos ver la memoria compartida creada, e incluso manipularla con programas que trabajen con ficheros (por ejemplo, usar el comando `rm` para borrarla). Esto es útil en el caso de que procesos que usen memoria compartida aborten o sean finalizados sin borrar la misma.

Ejercicio 2. **(ENTREGABLE)(2.5 ptos)** **Condición de carrera.** En este ejercicio se pide un programa escrito en el lenguaje C, *ejercicio2.c*. El programa generará `n` procesos hijos, donde `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 de tipo:

```
typedef struct{
      int previous_id; //!< Id of the previous client.
      int id; //!< Id of the current client.
      char name[NAME_MAX]; //!< Name of the client.
} ClientInfo;
```

El campo `id` ha de inicializarse a 0 y el campo `previous_id`, a -1.

Nota

La memoria reservada con `mmap` se hereda por los procesos hijos, así que no es necesario reabrirla en cada uno de ellos. Lo mismo ocurre con otros recursos como semáforos abiertos con `sem_open`.

Cuando el proceso padre reciba la señal `SIGUSR1` leerá de la zona de memoria compartida e imprimirá su contenido, nombre del usuario e identificadores, tanto del cliente previo como del actual.

Cada proceso hijo realizará los siguientes pasos (en este orden):

1. Dormirá un tiempo aleatorio entre 1 y 10 segundos.
2. Incrementará en una unidad el id del cliente previo en la memoria compartida.
3. Solicitará al usuario que de de alta un cliente, y leerá su nombre de la terminal, escribiéndolo en la memoria compartida.
4. Incrementará en una unidad el id del cliente en la memoria compartida.
5. Enviará la señal `SIGUSR1` al proceso padre.
6. Terminará correctamente.

El proceso padre terminará cuando todos los procesos hijos hayan terminado.

a) Implementa el programa propuesto **(1.25 ptos)**.

b) Explica en qué falla el planteamiento del ejercicio **(0.25 ptos)**.

c) Implementa un mecanismo (*ejercicio2_solved.c*) para solucionar este problema, basado en tu conocimiento de la asignatura **(1 ptos)**.

Ejercicio 3. (ENTREGABLE)(4.5 ptos) Problema del productor-consumidor. Implementar el problema del productor/consumidor. Para ello se realizarán dos programas (*ejercicio3_productor.c* y *ejercicio3_consumidor.c*). El productor deberá crear la memoria compartida para almacenar una cola circular de caracteres de tamaño 10, junto con los semáforos que se consideren necesarios para gestionarla. Acto seguido deberá leer caracteres de la entrada estándar, insertándolos en la cola. Cuando se llegue a EOF, deberá escribir un caracter NUL ('\0') en la cola y terminar, borrando la memoria. El consumidor, en cambio, se conectará a la memoria y semáforos sin crearlos y leerá de la cola, imprimiendo los caracteres leídos por la salida estándar. Una vez leído el caracter '\0', terminará. Se recomienda crear un tipo abstracto de datos para representar la cola en un fichero aparte, y usarlo en ambos programas.

a) Implementa los programas propuestos **(4 ptos)**.

b) Realiza los cambios mínimos necesarios para que la cola esté en un fichero, en lugar de en memoria compartida, que también se enlazará al proceso con `mmap` **(0.5 ptos)**.

SEMANA 2
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 ya vimos que dos procesos podían intercambiar información en forma de flujo continuo de caracteres a través de un pipe. Además de esto, Unix permite el intercambio de fragmentos discretos de información, o mensajes. Uno de los mecanismos 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.

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.

Para trabajar con colas de mensajes en C en sistemas Unix usaremos 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>`.

La sintaxis de las funciones anteriores es la siguiente:

- *`mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);`*

  Es análoga a open pero para colas de mensajes. `name` debe comenzar por el caracter / y no tener ningún caracter / adicional, por ejemplo "/ejemplo". Los parámetros `oflag` y `mode` son similares a los de `open`. `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 los flag `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_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 el flag `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: 0 or O_NONBLOCK */
       long mq_maxmsg;   /* Max. # of messages on queue */
       long mq_msgsize;  /* Max. message size (bytes) */
       long mq_curmsgs;  /* # of messages currently in queue */
  };
  ```

  El parámetro `mq_flags` deberá establecerse a 0. 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 usaremos. 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. El parámetro `mq_curmsgs` no se usa en la creación de la cola y debe ser 0.

  El valor retornado por esta función es un descriptor de cola de mensajes haciendo referencia a la cola abierta/creada. En caso de error devuelve `(mqd_t)-1`.

- *`int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);`*

  Esta función envía el mensaje apuntado por `msg_ptr` y con longitud `msg_len` y prioridad `msg_prio` a la cola de mensajes a la que hace referencia `mqdes`. Si el mensaje no fuera

una cadena de caracteres, deberá hacerse casting para enviarlo. El valor de `msg_len` debe ser menor o igual al tamaño de mensaje máximo especificado en la creación de la cola.

Esta función devuelve 0 si ha tenido éxito y -1 en caso de error. Recordad que al ser una llamada bloqueante, uno de los posibles errores es `EINTR`, en el caso de que haya un manejador de señal instalado y se reciba dicha señal durante el bloqueo.

- *`ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,size_t msg_len, unsigned *msg_prio);`*

  Esta función recibe un mensaje de la cola la que hace referencia `mqdes`. El mensaje se escribirá en el buffer `msg_ptr`, que tiene longitud `msg_len`. 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. El valor de `msg_len` debe ser mayor o igual al tamaño de mensaje máximo especificado en la creación de la cola.

  Esta función devuelve el número de bytes del mensaje leído si ha tenido éxito y -1 en caso de error. Recordad que al ser una llamada bloqueante, uno de los posibles errores es `EINTR`, en el caso de que haya un manejador de señal instalado y se reciba dicha señal durante el bloqueo.

- *`int mq_close(mqd_t mqdes);`*

  En este caso `mqdes` es el descriptor de cola de mensajes a liberar.

  Esta función devuelve 0 si ha tenido éxito y -1 en caso de error.

- *`int mq_unlink(const char *name);`*

  En esta función `name` es el nombre asignado a la cola de mensajes que se empleó en `mq_open` y que se desea borrar.

  Esta función devuelve 0 si ha tenido éxito y -1 en caso de error.

Nota

Las funciones anteriores se corresponden con la API más moderna del estándar POSIX para usar colas de mensajes. En otros años se ha empleado la API de System V, usando las funciones `ftok`, `msgget`, `msgsnd`, `msgrcv` y `msgctl`. La funcionalidad que ofrecen ambas es similar (aunque hay algunas diferencias), pero la nueva API es más sencilla de usar.

Ejemplo de uso:

Proceso 1

```
/**
 * @file
 *
 * @brief Código de ejemplo de cola de mensajes, para un proceso emisor.
 */

#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.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 */
	getchar();

	mq_close(queue);
	mq_unlink(MQ_NAME);

	return EXIT_SUCCESS;
}
```

Proceso 2

```c
/**
 * @file
 *
 * @brief Código de ejemplo de cola de mensajes, para un proceso receptor.
 */

#include <fcntl.h>
#include <sys/stat.h>
```

```c
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.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_RDONLY, /* 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;

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

	printf("%d: %s", msg.valor, msg.aviso);

	/* Wait for input to end the program */
	getchar();

	mq_close(queue);
	mq_unlink(MQ_NAME);

	return EXIT_SUCCESS;
}
```

Ejercicio 3. (ENTREGABLE)(3 ptos) Cadena de montaje. 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 abre con `mmap` un fichero cuyo nombre recibe como primer argumento de `main` y escribe en la primera cola de mensajes, cuyo nombre recibe como segundo argumento, 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 los caracteres en el rango a-z por su siguiente en el abecedario, convirtiendo la "z" en "a". Una vez realizada esta transformación, escribe el contenido en la segunda cola de mensajes. Este programa recibe como argumentos de `main` los nombres de ambas colas.
- El último proceso lee de la cola el trozo de memoria y lo vuelca por pantalla. Recibe como único parámetro de `main` el nombre de la segunda cola.

Por último se implementará un programa que reciba como argumentos el nombre del fichero y las dos colas. Este programa será el encargado de crear las colas, ejecutar los otros programas mediante la función `fork` y la familia de funciones `exec`, y borrar las colas cuando estos procesos terminen.

a) Implementa el programa del proceso A **(0.75 ptos)**.

b) Implementa el programa del proceso B **(0.75 ptos)**.

c) Implementa el programa del proceso C **(0.75 ptos)**.

d) Implementa el programa principal **(0.75 ptos)**.