Memoria Practica 3
Sistemas Operativos

Miguel Ibáñez y Sergio Hidalgo, Grupo 2273, Pareja 05

Ejercicio 1: Creación de memoria compartida

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

El código crea un segmento compartido de memoria, se puede saber debido a que se ve la flag O_CREAT, junto con las flags O_RDWR que sirve para leer y escribir, y la flag O_EXCL la cual se usa para devolver error si no existe el segmento. Y las dos flags de permisos son de permisos de usuario. Después del control de errores se abre el segmento (de lectura y escritura y sin permisos)

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.

Se debe quitar la flag O_EXCL patra evitar que devuelva error y se inicializa el objeto de memoria compartida SHM_NAME.

Ejercicio 2: Tamaño de ficheros

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

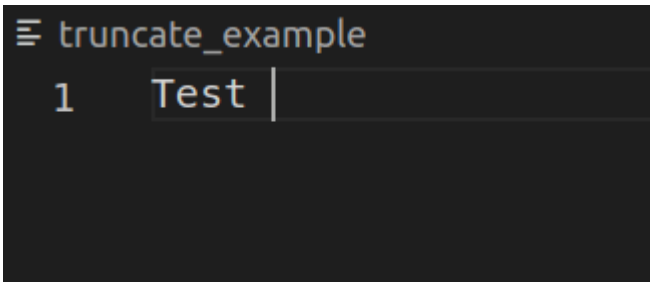
Para obtener el tamaño del código se usa la función *fstat*, con la cual la información se almacena en la estructura apuntada por el segundo argumento(statbuf), de esta estructura usamos st_size que nos dice el tamaño del fichero, el tamaño es 12.

```c
/* Get size of the file. */

if (fstat(fd, &statbuf) < 0)
{
    perror("fstat");
    exit(EXIT_FAILURE);
}

printf("Size : %ld\n", statbuf.st_size);
```

```
sergio@seregio-pc:~/Escritorio/uni/2o/2ocuatri/SOPER-P/Practicas-Soper/p3/ejs 1-
6$ ./file_truncate truncate_example
Size : 12
```

b) Completar el código anterior para truncar el tamaño del fichero a 5 B. ¿Qué contiene el fichero resultante?
Para truncar el tamaño a 5B usamos la función *ftruncate,* en la cual se pone de segundo argumento el tamaño que se le quiera dar a la memoria.

```c
/* Truncate the file to size 5. */
if (ftruncate(fd, (off_t)5) < 0)
{
    perror("ftruncate");
    exit(EXIT_FAILURE);
}
```



Ejercicio 3: Mapeado de ficheros
a) ¿Qué sucede cuando se ejecuta varias veces el programa anterior?
Aumenta el contador (el valor guardado dentro del fichero) y lo imprime.
¿Por qué?
Porque aumenta el valor guardado en la dirección de memoria mapped (que es el fichero).

```c
if (created) {
    *mapped = 0;
}
*mapped += 1;
printf("Counter: %d\n", *mapped);
```

b) ¿Se puede leer el contenido del fichero test_file.dat con un editor de texto?
No.
¿Por qué?
Porque el Sistema Operativo escribe en él como si fuese una dirección de memoria, por lo que está escrito en binario.

```
sergio@seregio-pc:~/Escritorio/uni/2o/2ocua
6$ hexdump -C test_file.dat
00000000  06 00 00 00
00000004
```

Ahí se puede ver que el contador está a 6.

Ejercicio 4: Memoria compartida
a) ¿Tendría sentido incluir shm_unlink en el lector?
No
¿Por qué?
Porque en caso de que otro proceso necesitase leer también los datos no podría hacerlo, pues al terminar el lector, el segmento queda inaccesible.

b) ¿Tendría sentido incluir ftruncate en el lector?
No.
¿Por qué?.
Porque si se intenta modificar el tamaño sin los permisos necesarios, el ftruncate fallará.

c) ¿Cuál es la diferencia entre shm_open y mmap?
Con shm_open se obtiene un descriptor de fichero y con el mmap se obtiene un puntero a la dirección de memoria del segmento.
¿Qué sentido tiene que existan dos funciones diferentes?
Que con shm_open se puede utilizar esta zona de memoria compartida como si fuese una pipe y con mmap se utiliza como si fuese un fichero, dependiendo del uso que se le quiera dar es más útil utilizar solo shm_open o ambas.

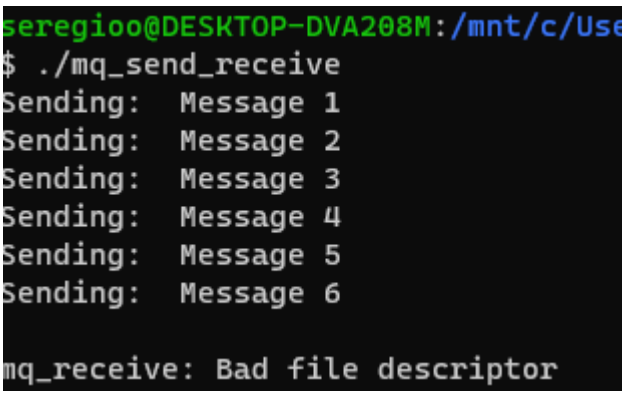
d) ¿Se podría haber usado la memoria compartida sin enlazarla con mmap? Si es así, explicar cómo.
Si, creando un fichero en el directorio (tal y como se hace en file_truncate.c).

Ejercicio 5: Envío y recepción de mensajes en colas
a) ¿En qué orden se envían los mensajes y en qué orden se reciben?
6, 4, 1, 2, 3 y 5
¿Por qué?
Porque el 6 tiene prioridad 3 (la más alta), el 4 tiene prioridad 2, y el resto tienen prioridad 1 (así que llegan en orden de envío)

b) ¿Qué sucede si se cambia O_RDWR por O_RDONLY?
Que al ser de solo de lectura y no de lectura y escritura, no se puede escribir el mensaje para mandarlo, por lo que aparece un mensaje de descriptor de fichero.

```
seregioo@DESKTOP-DVA208M:/mnt/c/Us
$ ./mq_send_receive
Sending:  Message 1
mq_send: Bad file descriptor
```

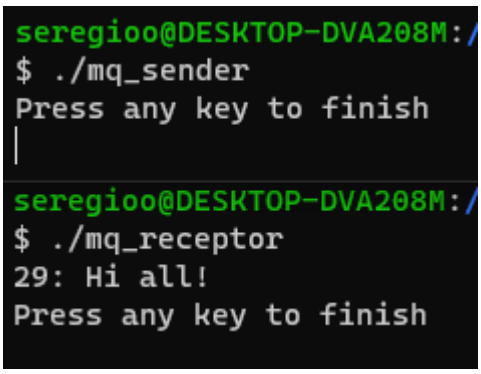
¿Y si se cambia por O_WRONLY?
Ahora solo es de escritura, por lo que solo se enviarán los mensajes, pero al no poder leer en el fichero dará un error en la lectura del fichero.



Ejercicio 6: Colas de mensajes

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

Cuando se ejecuta el receptor ya aparece el mensaje tanto de la cola como del código ("Press any key to finish")

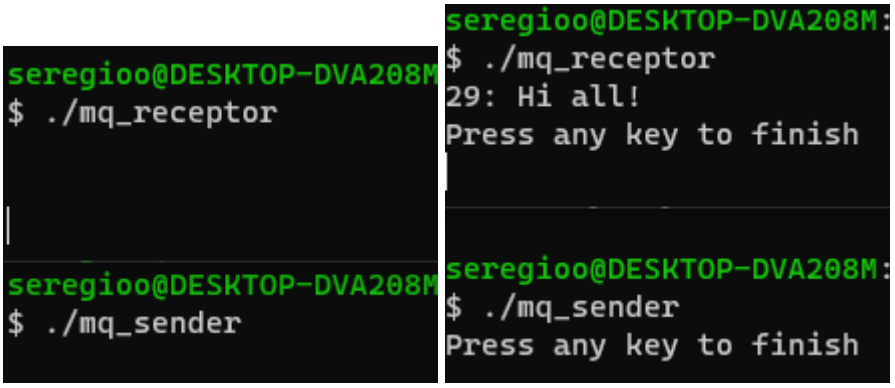


¿Por qué?

Porque el emisor envía el mensaje por la cola y cuando el receptor se ejecuta lo tiene pendiente y lo lee directamente nada más ejecutarse.

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

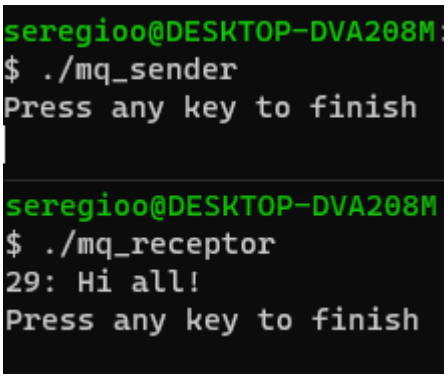
El receptor se queda esperando a que llegue algún mensaje por la cola, y cuando llega actúa de la manera esperada.



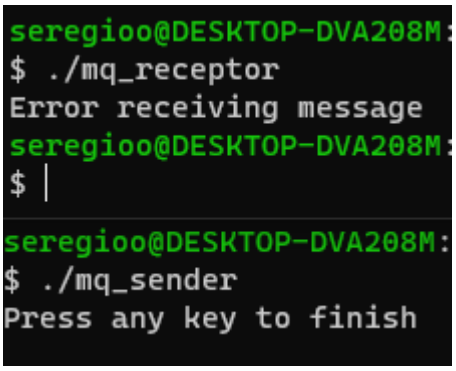
¿Por qué?

Porque al ser una cola bloqueante, bloqueará las operaciones de envío o recepción cuando no haya espacio en la cola o mensajes a recibir.

c) Repetir las pruebas anteriores creando la cola de mensajes como no bloqueante. ¿Qué sucede ahora?
Tras añadir la flag “O_NONBLOCK” y repetir la primera prueba todo funciona normalmente.



Pero en la segunda, al ejecutarse primero la espera de mensajes y luego la emisión de los mismos, la cola de mensajes para el receptor está vacía y salta error.



d) Si hubiera más de un receptor en el sistema, ¿sería adecuado sincronizar los accesos a la cola usando semáforos?
Si.
¿Por qué?
Porque para que todos recibiesen el mensaje, se debería enviar tantos mensajes como receptores haya, pues al recibir un mensaje se borra de la cola. De esta manera se asegura que todos reciban el mensaje.