Práctica 3: Memoria compartida y cola de mensajes.

Autores: Marcos Aarón Bernuy, Kevin de la Coba Malam. Pareja 04 del grupo 2922.

Ejercicio 1: Creación de Memoria compartida.

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

El código trata de abrir memoria compartida y después hace un control de errores sobre esta llamada. La llamada puede devolver 2 tipos de errores:

1. La función ha dado un error por una razón X.
2. La función ha dado un error porque ya existe la memoria compartida, por lo que volvemos a intentar abrir esta memoria, pero sin los argumentos O_CREAT y O_EXCL, ya que estos son los que juntos devuelven un error si ya existe la zona de memoria compartida

b) *En un momento dado se desearía 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 ´el) para forzar dicha inicialización.*

Añadir un shm_unlink en caso de que ya exista, haciendo después otro shm_open con los mismos argumentos.

Ejercicio 2: Tamaño de Ficheros.

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

```
struct stat fileStat;

fstat(fd, &fileStat);
```

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

```
ftruncate(fd, 5*sizeof(char));
```

Antes de cambiar el tamaño el fichero tiene "Test mesagge". Tras cambiar el tamaño el fichero contiene "Test".

Ejercicio 3: Mapeado de Ficheros.

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

Cada vez que se ejecuta el programa el contador aumenta su valor en 1. Esto es debido a que el valor se guarda en el archivo "test_file.dat".

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

No, es un archivo binario. Porque el valor se guarda en el archivo de forma binaria.

Ejercicio 4: Memoria Compartida.

a) *¿Tendría sentido incluir shm_unlink en el lector? ¿Por qué?*

No tendría sentido ya que el escritor ya hace `shm_unlink`.

b) *¿Tendría sentido incluir ftruncate en el lector? ¿Por qué?*

No tendría sentido ya que en el escritor se define el tamaño de la estructura a usar, si modificamos el tamaño en el lector y tratamos de acceder a la memoria, puede que tengamos un error ya que, o bien hemos aumentado la memoria y accedemos a un lugar que no nos corresponde, o bien hemos disminuido la memoria y a lo que antes accedíamos ahora ya no podemos.

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

`shm_open` devuelve un descriptor de fichero a la memoria compartida, mientras que `mmap` mapea la memoria compartida en una variable.

`shm_open` carga en RAM la memoria, pero `mmap` la añade al espacio de direcciones del proceso, por lo que si en algún caso no queremos tener memoria compartida en el espacio de direcciones del proceso pues solo haríamos `shm_open`.

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

Accediendo a /dev/shm, manipulando los ficheros.

Ejercicio 5: Envío y Recepción de Mensajes en Colas.

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

Los mensajes se envían en orden ascendente, 1, 2, 3, 4, 5, 6. Se reciben en este orden, 6, 4, 1, 2, 3, 4.

Los mensajes se envían en ese orden ya que es el orden en el que lo hacemos en el código.

Los mensajes se reciben en ese orden ya que dependen de la prioridad, a mayor prioridad, antes se reciben. Al ser una cola, si tenemos varios mensajes con la misma prioridad, se reciben en el orden de llegada (FIFO).

b) ¿Qué sucede si se cambia O_RDWR por O_RDONLY? ¿Y si se cambia por O_WRONLY?

Si cambiamos el argumento por O_RDONLY ocurre un error, ya que el descriptor recibido solo puede usarse para leer, por lo tanto, no se puede enviar ningún mensaje.

Si lo cambiamos por O_WRONLY, podemos enviar mensajes, pero a la hora de leerlos ocurre un error.

Ejercicio 6: Colas de Mensajes.

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

El receptor recibe el mensaje sin problemas.

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

Sucede lo mismo pero el receptor se bloquea hasta que recibe el mensaje. Una vez lo recibe continua su ejecución.

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

Si ejecutamos primero el receptor, la ejecución sería errónea ya que al llamar a `receive`, está función devuelve un error y se cierra el programa.

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

La cola de mensajes es manejada por el sistema operativo, por lo que no sería necesario sincronizar procesos, a no ser que se quiera un orden especifico de ejecución, por ejemplo, si tenemos 3 procesos receptores, y queremos que primero lea el P1, luego P3 y por último P2, si queremos ese orden siempre, sí podríamos usar un sistema se sincronización.

Ejercicio 7: Streaming (Codificación).

a) Memoria compartida.

El primer paso que realizamos fue la creación de `stream-ui.c` archivo en el cual implementamos la estructura que nos piden. Más tarde acabamos creando el fichero `ui_struct.h` porque esta estructura se empleará también en `stream-server.c` y `stream-client.c`.

En `stream-ui.c` una vez ya tenemos creada la estructura realizamos `shm_open` seguido de `ftruncate` para reajustar el tamaño de la memoria compartida y finalmente mapeamos la memoria compartida con una variable de la estructura previamente creada. Después inicializamos los valores de la estructura como nos pide el enunciado y además también inicializamos el valor del buffer a “`00000`”.

Posteriormente, realizamos los `fork` para conseguir el proceso *server* y el proceso *client*. Ambos procesos ejecutan `execl`.

A continuación, creamos los ficheros `stream-server.c` y `stream-client.c`. En ellos implementamos primero la recepción de argumentos al ser ejecutado (dichos argumentos son el archivo de entrada y salida), en ambos casos reciben un fichero, por lo que la implementación será la misma. El proceso *client* recibe el fichero de salida y el proceso *server* el de entrada. En `stream-ui.c` recibimos ambos ficheros como argumentos.

Como *stream-server* y *stream-client* necesitan utilizar la memoria compartida debemos hacer el mismo proceso que con *stream-ui*, pero en este caso necesitaremos menos `flags` al realizar `shm_open`, en concreto solo necesitaremos O_RDWR, tanto para *stream-server* como para *stream-client*, después realizaremos el mapeo de la memoria con un puntero. Estas flags permiten que editemos la memoria compartida, y aparte, al no incluir O_CREAT y O_EXCL, abrimos un segmento de memoria compartida solo si este ya se ha creado antes.

A parte ambos necesitarán abrir un archivo `.txt`, en el caso de *stream-client* se abrirá únicamente para la escritura y en el caso de *stream-server* para la lectura. En *stream-server* leemos carácter a carácter con `fgetc`, después copiamos los datos del fichero a la memoria compartida, es importante recordar que se trata de un buffer circular por lo que emplearemos "%" respecto al tamaño para que se mantenga entre los valores [0:(N-1)] (siendo N el tamaño del buffer, en este caso 5). En cuanto a *stream-client*, iremos leyendo los valores del buffer y los escribiremos en el file con `fwrite` y se tratará el counter para el buffer igual que con *stream-server*, donde al leer un carácter se aumenta `post_pos`, en *stream-client* se aumenta `get_pos`.

En *stream-client* para comprobar que al recibir \0 deje de escribir en el file usamos un `if` seguido de un `break`, en *stream-server* para añadir `\0` al final del fichero tenemos que en caso de que reciba EOF (End of file) el valor a introducir en el buffer sea `\0`.

b) Semáforos.

En esta parte primero usando el manual de Linux y referencias online nos informamos de como declarar semáforos sin nombre, para lo que usamos `sem_init`, y realizamos la inicialización de los semáforos mostrados en el algoritmo proporcionado. Para conocer que valor dar a los semáforos miramos en los apuntes el algoritmo productor-consumidor, y decidimos que `sem_fill` tendría que inicializarse a 0 y `sem_mutex` y `sem_empty` a 1.

Una vez tenemos la inicialización de los semáforos en *stream-ui* pasamos a implementar el algoritmo en *stream-server* y *stream-client* para lo que tuvimos que informarnos sobre `sem_timedwait` en el manual de Linux, y comprobamos que necesitaríamos añadir una estructura `timespec`, para lo que tuvimos que incluir `time.h`. Una vez creada una variable con esta estructura la tuvimos que usar `clock_gettime` para obtener el tiempo actual y sumarle dos segundos para que cumpliera con las especificaciones del ejercicio. Una vez el valor de la variable está definido, ya se pueden emplear las funciones `sem_timedwait`. Este proceso se lleva a cabo tanto en *stream-server* como *stream-client*. Como el ejercicio indica que se debe usar `sem_timedwait` en cada operación con semáforos lo empleamos con los tres, incluyendo `sem_mutex`. Por último, cabe destacar que para liberar los semáforos usamos `sem_destroy`.

c) Colas de instrucciones.

La creación de colas en *stream-ui* requiere una estructura de mensaje que definimos en `ui_struct.h`. Teniendo esta estructura definida al crear las colas en este fichero, ambas se inician con la constante O_WRONLY (en *stream-ui*). Cada cola estará destinada a cada proceso, ya sea *stream-server* o *stream-client*, que recibirán el nombre de `queue_client`, y `queue_server`. Ambas colas ejecutarán la función `mq_send` al recibir la instrucción `exit` redirigiendo dicha instrucción a los procesos *stream-server* y *stream-client*.

Para la creación de colas tanto para *stream-server* como para *stream-client* abrimos con `mq_open`, y ambas se abrirán con la constante O_RDONLY, ya que solo necesitamos leer la cola. Ambas colas realizarán la función `mq_receive` para recibir los mensajes que habían sido redirigidos desde *stream-ui*.

Las instrucciones que recibirá *stream-client* serán *get* y *exit*, y *post* y *exit* en el caso de *stream-server*. Todo esto será gestionado en *stream-ui.c*. En ambos casos la implementación respecto las instrucciones recibidas es; si recibe exit finaliza y si recibe cualquier otro valor hace la tarea (en el caso de *stream-server*) de *get* y (*stream-client*) de post. Está implementado de esta manera ya que realizamos control de errores en *stream-ui* para que *stream-client* no reciba instrucciones *get*, y *stream-server* no reciba instrucciones post.

Por último destacar que la ejecución correcta seria compilando con make y después ejecutando ./stream-ui a.txt b.txt en el terminal porque a.txt lo utilizamos como fichero de salida mientras que b.txt lo utilizamos de entrada.