Sistemas Operativos

Práctica 3

Antonio Javier Casado

Aurora Pérez Lázaro

1. Ejercicio 1

a. No tiene sentido incluir `shm_unlnk` en el lector porque es posible que haya más lectores que necesiten acceder a la memoria compartida que todavía no hayan abierto el descriptor de fichero correspondiente.

b. La función `shm_open` sirve para crear una zona de memoria compartida (o abrirla) mientras que mmap se encarga de unir el fichero creado por shm_open con el espacio de direccionamiento virtual de un proceso. Para poder hacer esto último, es necesario que anteriormente se haya creado la zona por lo que es necesario la existencia de ambas funciones. El sentido de que existan dos funciones así es que con `mmap` puedes abrir una parte de esa memoria compartida, sin tener que abrirla toda.

c. Sí sería posible usar la memoria compartida sin usar `mmap`. Se haría modificando el archivo en `/dev/shm`.

2. Ejercicio 2

a. El código dado consiste en abrir un segmento de memoria compartida, para ello comprueba si el segmento ya existía con `if (errno == EEXIST)`, si existe intenta abrir crear otra zona de memoria (`SHM_NAME`) con `O_RDWR`

que es para leer-escribir junto con los bits de permiso `mode_t` en este caso 0, que según el manual serán ignorado al no especificarse `O_CREAT` o `O_TMPFILE`.

b. Para inicializar la memoria a 1000B habría que añadir `ftruncate(fd_shm, sizeof(char) * 1000)`. Para que futuras ejecuciones no destruyan la zona de memoria haría falta (CREO) mapear la memoria con `mmap` después de comprobar que la zona de memoria existe.

c. Para forzar la inicialización de *"/shared"* se podría hacer eliminando el fichero de `/dev/shm` y volver a abrirlo con ese nombre.

3. Ejercicio 3

a. Se ha creado el código *shm_concurrence* en el que se crea memoria compartida y se crean procesos hijos. Solo ha sido necesario hacer una vez `shm_open()` y `mmap()` debido a que los procesos hijo heredan la zona de memoria reservada por `mmap`.

   El padre realiza una espera *semi-activa* (sleep dentro de un bucle while) en el que cuando un hijo le manda la señal, este comprueba el número de log para saber si todos han escrito ya, en el siguiente apartado se explica porque falla esto.

b. El ejercicio falla en que no se respeta la zona de exclusión mútua para el contador `logid` lo que significa que no se registrará correctamente los incrementos de esta variable. El programa probablemente (dependiendo de la planificación) no funcionará para 2 o más procesos hijos y el padre no sabrá cuál ha sido el último en modificar la memoria compartida.

c. Para arreglar el problema se ha puesto un semáforo en la zona de memoria compartida, para esto no hace falta crear el semáforo con `sem_open()`, si no solo inicializarlo con `sem_init()`.

   El semáforo protege la zona en la que se comprueba y se aumenta la variable `logid` para que no se solapen los procesos, de esta forma el proceso padre sabrá cuando ha sido la última vez que se escribe en la memoria compartida.

4. Ejercicio 4

a. Se han creado los programas *shm_consumer* y *shm_producer*. El funcionamiento principal de ellos consiste en que el productor guarde en una cola una serie de números entre 0 y 9 y los comparta mediante un segmento de memoria compartida y el consumidor pueda acceder a la cola y leer los elementos. En cada programa se ha creado (shm_producer) y abierto (shm_consumer) la memoria compartida, se ha ajustado el tamaño, se ha mapeado con la estructura donde se guarda la cola y se han inicializado tres semáforos. Los semáforos se encuentran junto con la cola en la estructura que se va a mapear. Después, cada proceso ha hecho su trabajo necesario. En el caso del productor, se ha considerado insertar en la cola los números desde 0 hasta N en orden repitiendo la secuencia 0-9, es decir, i % 10. Se ha usado el módulo de *queue.c* y *queue.h* para poder implementar las funciones de la cola. Además se ha agregado un *types.h* que facilita la implementación de esas funciones. Sin embargo, debido a diferentes problemas que han surgido con las funciones, se ha acabado poniendo el código correspondiente a cada función en el productor o el consumidor.

b. Para este apartado se han cogido los programas anteriores y se ha cambiado la forma de almacenar la cola. En este caso, se ha creado un fichero en el productor y se ha abierto en el consumidor. El insertado en la cola se ha conservado y las estructuras también. Para el manejo de ficheros se ha usado la función *open()*.

5. Ejercicio 5

a. Al ejecutar ambos códigos, el emisor ejecuta el código satisfactoriamente y sale como debería. Sin embargo, el receptor se queda bloqueado y no consigue completar su código. Esto se debe a que el receptor se queda en la llamada bloqueante de mq_receive esperando el mensaje. Este bloqueo es debido a que el campo *.mq_flags* de *atributos* tiene un valor de 0, lo que implica que las funciones de recepción y envío de mensajes se bloquearán.
b. Al ejecutar primero el receptor, como no hay ningún mensaje que recibir,

se va a quedar bloqueado esperando. Si salimos de la ejecución para poder ejecutar el siguiente

c. En el apartado a hemos visto que el campo *.mq_flags* puede hacer que las funciones de recepción y envío de mensajes sean bloqueantes o no. Para que no sean bloqueantes debemos cambiar el valor de ese campo por O_NONBLOCK.

6. Ejercicio 6

a. Se ha implementado los programas *mq_injector.c* y *mq_workers_pool.c* modularizado en los archivos *pool_trabajdores.c* junto con la cabecera *pool_trabajadores.h*.

   El programa *mq_injector* se ha implementado de forma que crea la cola de mensajes y posteriormente vaya leyendo del fichero trozos del tamaño indicado y enviandolos consecutivamente a la cola. Una vez leído todo el fichero, se manda un mensaje especial de finalización (cadena *fin_de_mensaje)* que los programas receptores sabrán que este mensaje indica que se ha leído todo el fichero.

   El programa *mq_workers_pool* de forma similar abre la cola en el padre (los hijos heredan el descriptor de fichero) y van leyendo de la cola siempre que el padre no les haya enviado la señal *SIGTERM.* Debido a que la función *mq_receive()* puede ser interrumpida por esta señal, se comprueba si ha sido así y se sale del bucle con normalidad.

   El programa se ha comprobado con el fichero *prueba.txt.*

b. El único cambio hecho para este programa es usar la función *mq_timedreceive()* y comprobando los valores de retorno y *errno:* si el valor de *errno* es *ETIMEDOUT* ha pasado el tiempo.