Sistemas Operativos, Práctica 3: Memoria

Leandro García y Fabián Gutiérrez (Pareja 02)

14 de abril de 2020

Ejercicio 1

a) No tendría sentido añadir la instrucción `shm_unlink`, ya que el proceso escritor es quién se encarga de crear la memoria y de realizar dicha acción. Esto implica que, una vez que todos los procesos que están usando la memoria compartida dejen de referenciarla ("cierren el fichero") o mapearla, esta se eliminará sin necesidad de que ningún otro proceso llame a la función.

b) Mientras que `shm_open` abre (o crea) el segmento correspondiente de memoria, `mmap` lo añade a la memoria virtual del proceso, es decir, le asigna una posición de memoria que puede ser usada en dicho proceso. El hecho de que se usen dos funciones en vez de una permite que se pueda obtener el descriptor de fichero de un segmento de memoria sin necesidad de mapearlo. Por ejemplo, permite modificar el tamaño del objeto de memoria compartida con `ftruncate`, especialmente al proceso que lo crea, antes de realizar (o no) el mapeo, o que cada proceso con acceso a la memoria compartida pueda mapearla como desee: qué permisos tiene el mapeo (siempre compatibles con los del descriptor de fichero), qué parte de la memoria se mapea, etc.

c) Podría haberse omitido el mapeo en memoria virtual realizando las lecturas y escrituras con `read` y `write` sobre el descriptor de fichero devuelto por `shm_open`. Estas lecturas y escrituras, sin embargo, no contarían con las comodidades que ofrece la manipulación de la información mediante la estructura de ejemplo definida.

Ejercicio 2

a) En el código primero se intenta abrir un objeto de memoria compartida con `shm_open` y las banderas `O_CREAT` y `O_EXCL`. Dadas estas banderas, si la apertura no da errores (en particular, el objeto que se pretende abrir no existe), se crea según las banderas restantes (`O_RDWR`, `S_IRUSR`, `S_IWUSR`) y `fd_shm` obtiene un descriptor de fichero válido. En cambio, si hubo algún error en la apertura, `fd_shm` recibe -1. En este caso, si el error es que el objeto ya existía, se intenta abrir de nuevo omitiendo las banderas asociadas a la creación del objeto de memoria compartida.

   Esta forma de abrir un objeto de memoria compartida permite crearlo en caso de que no exista (uso de `O_CREAT`) y además detectar si ya existía (uso de `O_EXCL`), lo que permite un manejo más seguro del objeto ya que las banderas de creación se ignoran en caso de que ya existiera el objeto y sería riesgoso asumirlas al no corroborar que el objeto abierto ya había sido creado con otras banderas (posiblemente distintas).

b) Habría que añadir una llamada a `ftruncate` solo en caso de que se creara un nuevo objeto, esto es, en caso de que no salte el error `EEXIST` (y en general, ningún otro error):

```
if (fd_shm == -1) {
    /* codigo */
}
else {
    if (ftruncate(fd_shm, 1000) == -1) {
        /* control de errores */
    }
    printf("Shared memory segment created \n");
}
```

c) Una opción sería eliminar el fichero correspondiente de `/dev/shm`, por ejemplo con el comando `rm`, antes de la siguiente ejecución. Otra opción sería ejecutar un programa que haga `shm_unlink` del objeto de memoria compartida, lo que provocaría que sea eliminado al no estár abierto o mapeado por proceso alguno.

Ejercicio 3

b) El planteamiento del ejercicio falla en la gestión de la zona de memoria compartida, que debería ser una zona crítica. Puesto que las variables compartidas pueden ser accedidas por distintos procesos simultáneamente, los cambios en ellas deben ser regulados por semáforos o algún método análogo para evitar que el correcto resultado de la ejecución dependa de la planificación del procesador (esto es, evitar condiciones de carrera).

Cabe destacar otro error, quizá de menor gravedad: el envío simultáneo de señales conlleva el riesgo de que haya pérdidas al recibir múltiples de estas mientras la señal se encuentra bloqueada (mientras se generan los procesos hijos o se ejecuta el manejador). Sin embargo, este error queda resuelto cuando se arregla el error anterior.

Ejercicio 4

a) Por simplicidad, en la implementación es el consumidor quien hace `shm_unlink` del objeto de memoria compartida (además de destruir los semáforos) al ser siempre el último proceso en utilizarlo. Esto permite que los procesos funcionen correctamente siempre (estrictamente hablando, "en general") que el productor sea lanzado antes que el consumidor. Si fuese el productor el encargado de liberar los semáforos y de desligar la memoria compartida, podría darse el caso de que el consumidor no logre acceder a la memoria compartida antes de que el productor la elimine, por ejemplo, cuando N es menor que el tamaño de la cola circular (el productor tiene espacio disponible para terminar su producción y por tanto no necesita esperar que el consumidor vaya liberando espacio).

Si quisiera coordinarse de forma más fina la ejecución de los procesos, además de imponer que el productor es quien libera los recursos compartidos, deberían implementarse mecanismos de comunicación adicionales que permitan que:

- El consumidor espere a que el productor cree el objeto de memoria compartida antes de intentar acceder a este.
- El productor espere a que el consumidor finalice su uso de los recursos compartidos antes de liberarlos.

Por ejemplo, podría hacerse uso de dos semáforos con nombre, cada uno creado por el primer proceso que lo intente e inicializados a 0, y desligados por el segundo (uno lo crea, el otro lo desliga tras abrirlo). El productor haría `up` del primero después de crear la memoria compartida y `down` del segundo antes de liberar los recursos compartidos, y el consumidor, análogamente, haría `down` del primero antes de intentar acceder a la memoria compartida y `up` del segundo después de consumir el último producto (el -1 final). Esta alternativa conlleva sus propios problemas cuando uno de los procesos falla, ya que los semáforos pueden quedar no-desligados (podría arreglarse haciendo `sem_unlink` redundantes) o el otro proceso esperando un `up` que nunca llegará.

Otra alternativa, quizá mas rebuscada, sería crear un proceso controlador que lance los procesos productor y consumidor y los coordine mediante el envío y recepción de señales (y la función `sigsuspend`).

b) Los cambios realizados fueron, fundamentalmente, eliminar la expresión regular "`shm_`" y cambiar el nombre de la memoria compartida por uno más apropiado (en este caso, "`/shm_producer_consumer`" por "`shm_producer_consumer.dat`").

Ejercicio 5

a) Si se lanza primero el proceso *sender* este enviará el mensaje y se quedará esperando por un caracter para terminar. Cuando se lance el proceso *receptor*, este mostrará por pantalla el mensaje recibido y pasará a estar listo para terminar. Esto se debe a que, cuando el proceso receptor se ejecuta, ya hay un mensaje en la cola y por tanto se lee de forma inmediata.

   Ha de tenerse en cuenta que, si antes de ejecutar el proceso receptor el emisor termina, se hará una llamada a `mq_unlink`, por lo que se borrará la cola de mensajes al no estar en uso y el segundo proceso creará una nueva cola vacía, quedando así bloqueado en espera de un mensaje.

b) Al lanzar el proceso receptor, este crea la cola, pero no hay ningún mensaje que recibir, por lo que se bloquea. En el momento en que se lanza el emisor, el mensaje se envía, el receptor lo muestra por pantalla y ambos finalizan.

c) Una vez se tiene la cola como no bloqueante, si se repite el apartado a) se puede comprobar que funcionará igual, ya que el emisor añade en primer lugar el mensaje en la cola y el receptor lo lee cuando sea ejecutado.

   Sin embargo, al repetir el apartado b) con esta nueva condición, se llamará a la función `mq_receive` antes de que haya mensajes en la cola, por o que el proceso receptor terminará con un error al no poder recibir el mensaje (ya que ahora no queda bloqueado a la espera de que este llegue).

Ejercicio 6

a) Para elaborar `mq_injector` se leen fragmentos de texto que se envían sucesivamente hasta llegar al final del fichero. En este momento se envía un mensaje cualquiera que tenga como primer caracter `EOF` a modo de mensaje final.

   En el código de `mq_workers_pool`, cada trabajador entra en un bucle en el que va leyendo mensajes y buscando el caracter indicado. En el momento que un trabajador recibe el mensaje que comienza por `EOF`, este sale del bucle y envía a la cola de mensajes N-1 cadenas aleatorias comenzadas por `'~'` (es un caracter poco utilizado). Con esto se permite al programa sacar del

bucle al resto de trabajadores una vez hayan acabado sin interrumpir su trabajo y distinguirlos del primero.

b) Para implementar este apartado se establece, además de los ya presentes, un manejador para `SIGALRM`. Este manejador, junto con la función `ualarm` (para utilizar alarmas inferiores a un segundo), permite interrumpir la ejecución de la función `mq_receive` cuando está bloqueada. Esta llamada acabará en ese caso devolviendo -1 y poniendo `errno` a `EINTR`. Si en cambio se recibe la alarma en otro punto de la ejecución, particularmente después de recibir un mensaje, se ejecuta el manejador vacío sin mayores consecuencias.