PRÁCTICA 3

SISTEMAS OPERATIVOS

Jorge de Miguel Pires
Laura de Paz Carbajo
Grupo 2202
3/04/2020

EJERCICIO 1

a) Suponiendo que en un primer lugar se ejecuta siempre el programa del archivo `shm_writer.c`, no tiene sentido incluir `shm_unlink()` en el lector porque la memoria compartida ya se ha desvinculado en este programa, que es donde se escribe en la memoria y espera a que pueda ser leída. Para ello, el programa realiza una llamada a `getchar()` que espera indefinidamente hasta que el usuario introduce un caracter y, entonces se elimina la memoria del espacio virtual de direcciones del proceso mediante `munmap()` y luego se llama a `shm_unlink()`. Esta última línea del programa ordena el borrado de la memoria compartida y este se realiza cuando no queda ningún proceso que la tenga mapeada.

b) La función `shm_open()` permite crear memoria compartida en el sistema, mientras que `mmap()` permite al proceso desde donde se llama a esta función vincular este fragmento de memoria compartida a su memoria virtual. Tiene sentido que existan ambas funciones ya que sin `shm_open()` toda la memoria que se cree sería bien compartida por todos los procesos o propia de cada proceso, y sin `mmap()` dos procesos diferentes no podrían acceder a datos comunes que se encuentren en esa memoria compartida.

c) Al realizar la llamada a `shm_open()`, esta función te devuelve un descriptor de fichero como haría de manera análoga la función open. Si no utilizásemos la función `mmap()` para vincular el espacio de memoria compartida a la memoria virtual del proceso se podría acceder y modificar dicha región de memoria utilizando el descriptor de fichero devuelto por `shm_open()` y obtener información sobre esta memoria mediante `fstat()`.

EJERCICIO 2

a) El código del archivo `shm_open.c` crea primero un nuevo segmento de memoria compartida mediante `shm_open()` incluyendo como flags `O_CREAT` y `O_EXCL`. Al combinar estos dos macros, se crea memoria compartida solo si no existe ya memoria con el nombre pasado como argumento. En caso de que exista, la función devuelve -1 y errno se modifica con el valor `EEXIST`.
A continuación, el programa comprueba si ha habido algún error (`fd_shm == -1`) y si ese error ha sido porque ya existía memoria con dicho nombre (`errno == EEXIST`). En tal caso, se intenta abrir ese fragmento de memoria compartida usando una vez más `shm_open()`, pero esta vez utilizando la flag `O_RDWR` únicamente.
Si en la primera llamada a `shm_open()` ha ocurrido un error pero no ha sido de tipo `EEXIST`, se imprimer que ha ocurrido un error y se sale del programa sin éxito.
Por último, si en esa primera llamada no se ha producido un error, lo cual quiere decir que no existía anteriormente memoria compartida con ese nombre, se imprime un mensaje de éxito y se finaliza la ejecución.

b) Para que la memoria se inicialice con un tamaño de 1000B, es necesario usar la función `ftruncate()`, la cual establece el tamaño del segmento de memoria compartida al tamaño que le pasemos como argumento.
Como se ha explicado en el apartado a), este código solo crea nueva memoria compartida cuando no se ha creado con anterioridad, por lo que durante la ejecución no se destruiría

ningún segmento de memoria ya creada. Para evitar sucesivas llamadas a `ftruncate()`, la llamada a esta función debería realizarse tras comprobar que el primero `shm_open()` no da error, es decir, en el `else` final.

c) Para forzar la inicialización de dicho objeto se nos ocurre la opción de entrar en el directorio `/dev/shm` y borrar el descriptor de fichero a esta memoria compartida mediante el comando "`rm /nombre_del_descriptor`" por lo que, cuando se realice la siguiente llamada a `shm_open()`, esta no devolverá error porque no existirá la memoria compartida con dicho nombre.

EJERCICIO 3

a) shm_concurrence.c (comentarios incluidos)

b) El principal problema de este planteamiento es que, dependiendo de el orden en el que se ejecuten los diferentes hilos del proceso, la memoria compartida puede sobreescribirse antes de que el padre la lea e imprima por pantalla. Para solucionar este problema utilizaremos semáforos para controlar el acceso a dicha memoria.

c) shm_concurrence_solved.c (comentarios incluidos)

EJERCICIO 4

a) shm_producer.c y shm_consumer.c (comentarios incluidos)
b) (OPCIONAL) shm_producer_file.c y shm_consumer_file.c (comentarios incluidos)

EJERCICIO 5

a) Al ejecutar primero el emisor y luego el receptor, este último imprime la frase "`29: Hola a todos`" la cual corresponde con el mensaje enviado por el emisor a este proceso.

b) Si lo hacemos al revés, podemos observar como el receptor se queda "parado" (se bloquea) y una vez ejecutamos el proceso emisor y se envía el mensaje, entonces el receptor imprime el mensaje y acaba su ejecución. Esto ocurre porque las funciones `mq_send()` y `mq_receive()` son bloqueantes.

c) 1. En el primer caso no cambia nada. Cuando el receptor quiere acceder al mensaje, como este ha sido enviado anteriormente, no se bloquea, sino que lo recibe y continúa su ejecución.
2. En este segundo caso, si ejecutas primero el programa del receptor se produce un error "`Error receiving message`" ya que en la función mq_receive() no ha recibido nada y el programa finaliza sin esperar a que llegue el mensaje.

EJERCICIO 6

a) mq_injector.c y mq_workers_pool.c (comentarios incluidos)