Junco de las Heras & Marta Vaquerizo

SISTEMAS OPERATIVOS
PRÁCTICA 3

Ejercicio 1: Memoria Compartida.

a) No, no tiene sentido incluir **shm_unlink** en el lector, ya que esta llamada ya se hará en el escritor. Si se hace en el lector, solo se puede ejecutar 1 vez el código del lector, ya que al ejecutarse otra vez no se encontrará el segmento de memoria compartido.

b) **shm_open** lo que hace es abrir un descriptor de fichero útil para luego mapearlo, mientras que **mmap** mapea un segmento de memoria para que puedan acceder a él varios procesos. Tiene sentido que existan las dos, ya que **mmap** también puede compartir otro tipo de variables que no sean de la familia shm (como un int por ejemplo).

c) No, si no se ejecuta **mmap**, el segmento de memoria de un proceso no se comparte con el otro proceso, así que el lector no puede acceder a los datos que ha escrito el escritor.

Ejercicio 2: Creación de Memoria Compartida.

a) El código de este ejercicio consiste básicamente en intentar crear un segmento de memoria compartida con **shm_open**, donde en el argumento flags se pone **O_EXCL** aparte de **O_CREAT**. Si la llamada no da error, es que no existía el segmento y se ha creado correctamente. En otro caso, se pueden dar dos situaciones, la primera es que ya se había creado un segmento con ese nombre (**if (errno == EEXIT)**), y en ese caso, se intenta abrir el segmento. Si da error al abrir, acaba el programa, si no, se continúa, como si no se hubiera producido el error. Y la segunda situación posible es que no se haya podido crear el segmento, en ese caso, el programa finaliza.

Tiene sentido abrir un objeto de memoria compartida de esta forma, ya que se permite al proceso continuar si el objeto, que el proceso quiere crear con ese nombre, ya se ha creado.

b) Por un lado, en el else más externo fijar el tamaño del segmento con:

ftruncate (fd_shm, 1000);

Y por otro lado, para que los otros procesos no puedan destruir lo ya inicializado, la bandeja **O_RDWR** se cambia por **O_RDONLY**.

c) Después de **if (errno == EEXIST)** se puede insertar un **shm_unlink (SHM_NAME);** para que le diga al sistema operativo que ya no necesita este trozo de memoria, y se vuelve a intentar abrir la memoria con **shm_open**. Si otro proceso

está usando el trozo de memoria, el bucle se volverá a repetir, hasta que ese trozo de memoria sea liberada, para que el programa principal la cree, como se pide. Para detalles de implementación véase **shm_open.c**.

Ejercicio 3: Concurrencia en Memoria Compartida.

a) El código está en **shm_concurrence.c**. El código incluye comentarios para cada acción importante que realiza.

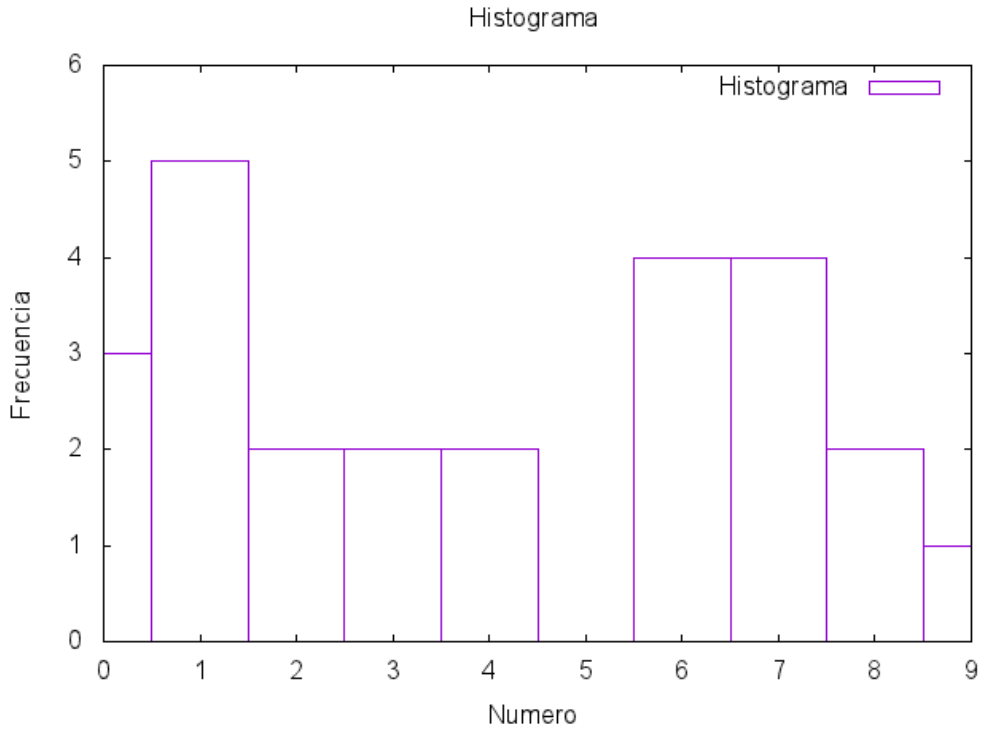
b) Ha fallado que al dormir menos de un segundo los procesos no les da tiempo a guardar todos los campos antes de que se acabe su turno de procesamiento, así que se sobrescriben y se pierden datos.

c) El código está en **shm_concurrence_solved.c**. Se ha utilizado un semáforo sin nombre dentro de la estructura. Cada vez que un proceso quiere escribir el log, después de la espera, lo que hace es **sem_wait** (deja el mutex a 0) y entra en la zona crítica. Dentro escribe el log, manda la señal SIGUSR1 al padre y hace otro **sem_wait**. Espera a que el padre lea el log y sea este el que haga un **sem_post** para que el hijo pueda salir de la zona crítica. Finalmente el hijo hace otro **sem_post** para que el mutex quede a 1 y pueda servir para el siguiente proceso.

Ejercicio 4: Problema del Productor-Consumidor.

a) El código se encuentra en **shm_producer.c** y **shm_consumer.c**. Se implementa el algoritmo productor-consumidor. Hay comentarios en el código por cada bloque importante de funcionalidad.

Se ha probado el código con N = 25 y que se generen de forma aleatoria, y el histograma que ha quedado, con los datos del **data.txt**, es:



b) La solución a este apartado se encuentra en los ficheros **shm_consumer_file.c** y **shm_producer_file.c**.

En este apartado, la estructura del fichero está compuesta de una cola circular de números enteros y los semáforos: mutex, empty y fill. Es decir, el mínimo cambio ha sido abrir el fichero, mapearlo, usarlo y borrarlo, en vez del segmento de memoria

Ejercicio 5: Colas de Mensajes.

a) Lo que sucede al ejecutar primero el del emisor es que crea la cola y envía un mensaje. Se queda esperando y (sin que acabe el proceso y haga **unlink**) ejecutando desde otra terminal el receptor, puede leer el mensaje e imprime por pantalla: "29: Hola a todos".

b) Lo que sucede ahora al ejecutar primero el receptor es que este se queda esperando a que alguien inserte un mensaje en la cola. Una vez que ejecutamos el proceso del emisor, inserta un mensaje en la cola, por lo que el receptor lo lee, muestra por pantalla "29: Hola a todos" y finaliza.

c) Ahora en **mq_open** añadimos el atributo **O_NONBLOCK**, que permite a la cola no bloquearse en las situaciones en las que antes se bloqueaba (Para leer si la cola estaba vacía y para escribir si la cola estaba llena). La prueba de a) sigue siendo correcta, el emisor envía el mensaje y el receptor lo recibe correctamente, imprimiendo "29: Hola a todos". Ahora el b) al ejecutar primero el receptor, y este encontrarse la cola vacía, y la función **mq_receive** devuelve -1, entrando en el tratamiento del error, imprimiendo "Error receiving message" y saliendo con EXIT_FAILURE.

Ejercicio 6: Pool de Trabajadores.

a) El código se encuentra en los ficheros **mq_injector.c** y **mq_workers_pool.c**. Hay comentarios en el código por cada bloque importante de funcionalidad.

b) El código se encuentra en el fichero **mq_workers_pool_timed.c.** Se ha usado la función **ualarm** para gestionar el tiempo de espera, indicando **ualarm (100*1000, 0);** para esperar a 100 ms. Para que funcione el **ualarm**, el **mq_receive** debe de ser bloqueante (no se le debe poner la flag **O_NONBLOCK** a **mq_open**)