MEMORIA PRÁCTICA 3

CLAUDIA GONZÁLEZ ARTEAGA Y LAURA INIESTA RODRÍGUEZ
Nota: el makefile está hecho para ejecutar todos los ejercicios menos el 4, que tiene su propio makefile.

1. a) No sería necesario, unlink borra el nombre de la memoria compartida pero espera a que todos los procesos acaben de usarla para eliminarlo del todo por lo que con ponerlo una vez éste esperará tanto a que el escritor como el lector terminen de usarlo para borrarlo.

   b) Shm_open se encargá de crear o abrir si ya está creado el espacio de memoria compartido devolviendo un descriptor de fichero para acceder a ésta y mmap se encarga de enlazar la memoria al espacio de direcciones virtual del proceso devolviendo una estructura a través de la cual se puede acceder a la memoria sin necesidad de usar el descriptor que se puede cerrar con close sin afectar al mapeado.
   Se tienen las dos funciones para facilitar el acceso a la memoria compartida ya que usando la estructura devuelta por mmap es más sencillo que con el descriptor de fichero.

   c) Si, usando las funciones read y write con el descriptor de fichero pero sería más complejo que con mmap.

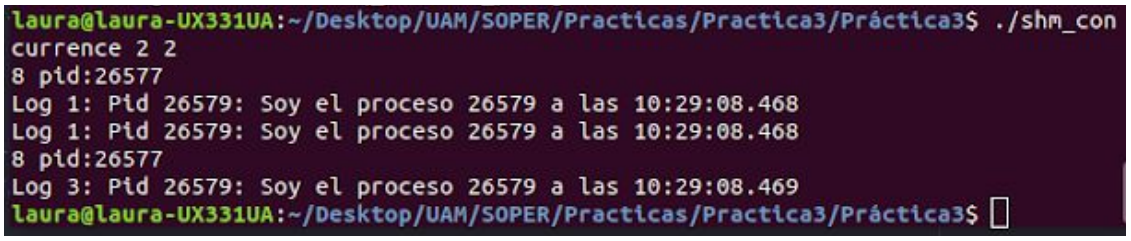
2. a) Este programa comienza con una llamada a shm_open, que crea o abre un segmento de memoria compartida, y devuelve un descriptor de fichero que hace referencia al mismo (fd en este caso).
   Si el descriptor devuelto (fd) es -1, comprobamos si el segmento de memoria compartida ya existe. Si esto ocurre, Si (errno devuelve EXISTS) Volvemos a hacer shm_open para intentar acceder a él, pero no podemos así que devolvemos un error: Error al abrir el segmento de memoria compartida.
   Por otro lado, si el descriptor devuelto no es -1 entonces se ha creado correctamente el segmento de memoria compartida.

   b) Para cambiar el tamaño del fichero o segmento de memoria compartida indicado usamos la función ftruncate, que recibe como primer argumento, fd y como segundo argumento, length, donde indicamos el tamaño que se le quiere dar a la memoria.

   c) Podemos hacer posibles soluciones para forzar la inicialización del objeto de memoria compartida "/shared". Por un lado tenemos la opción 1, sin programar, usando la consola de comandos , donde accederemos al directorio /dev/shm y ahí borrar la memoria compartida creada para que en la siguiente ejecución el sistema tenga que volver a crear un nuevo segmento de memoria compartida e inicializarlo

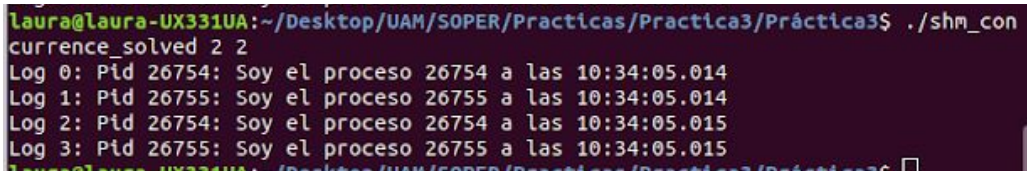
Por otro lado, tenemos la opción 2, en la que introducimos otra llamada a la función ftruncate después de la llamada shm_open si errno da EEXIST.

3. a) shm_concurrence.c adjunto



b) El planteamiento falla en cuanto al control de concurrencia ya que no hay.
Al ejecutar el código del apartado a podemos observar como no escribe todas las veces que debería y las llamadas que hace al manejador quién se encarga de sacar por pantalla la información no siempre es lo que debería, repite información, se salta números… Como vemos en la foto del apartado a, se imprime dos veces log1 y no hay log2.
Esto es debido a que no hay ningún tipo de control sobre la concurrencia de los procesos, quién y cuándo se escribe/ lee de la memoria compartida por lo que puede que se escriba 3 veces seguidas sin leer entre ellas porque nadie controla el acceso a la memoria dando lugar a que se lea información repetida.

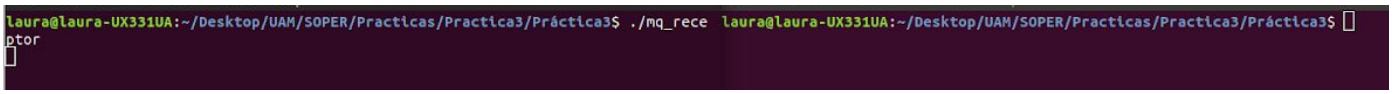
c) shm_concurrence_solved.c adjunto.

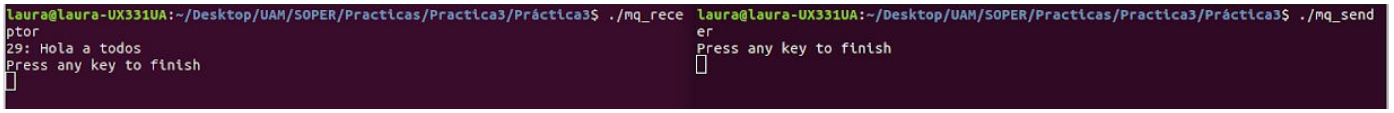


4. b) shm_producer.c y shm_consumer.c adjunto con el makefile correspondiente al ejercicio 4.

5. a) Si ejecutamos primero el sender, en cuanto ejecutamos receptor éste lee el mensaje y lo imprime por pantalla ya que sender nada más ejecutarse lo escribe en la cola por lo que el receptor no tiene que quedarse esperando a recibir el mensaje.

b) Si ejecutamos primero el receptor debido a que la llamada a receive es bloqueante, éste se queda esperando a recibir el mensaje que debe leer, en cuanto ejecutamos el sender, el receptor imprime el mensaje que ha recibido por pantalla.





c) Al hacer la cola no bloqueante,cuando ejecutamos el receptor antes que el sender, el proceso no se queda a la espera de que haya un mensaje para leer y recibe error.
Si ejecutamos el sender antes que el receptor no hay fallos porque escribe el mensaje y el receptor lo lee.

6. mq_injector.c y mq_workers_pool.c adjuntos
   Al ejecutarlos podemos ver que hacen su trabajo correctamente pero en caso de que un proceso haya acabado de leer y envíe SIGUSR1 y mientras otro proceso esté entrando en mq_receive se imprime error debido a que se interrumpe la ejecución del receive por una señal pero todos los procesos hacen lo que deben y terminan imprimiendo la información de los mensajes que han leído.