Memoria Práctica 3- Sistemas Operativos

- **Ejercicio 2**

En este ejercicio se nos pide implementar comunicación entre procesos mediante el uso de memoria compartida. Esto lo hacemos gracias a las funciones de C dedicadas a la memoria compartida (*shmget*, *shmat*...).

El programa lanzará tantos procesos hijos como especificados por parámetro y tiene que gestionar la correcta compartición y la sincronización entre procesos, para que la lectura y escritura de memoria compartida se haga de forma ordenada y efectiva, sin que haya ningún tipo de interbloqueo al acceder a la misma y que a la vez se cumpla la exclusión mutua. Para ello hemos gestionado todo esto con semáforos.

Este es el error que se describe en el enunciado, que es imposible ejecutar correctamente este ejercicio sin semáforos. Para controlar esto hemos diseñado un nuevo ejercicio con nombre de fichero ejercicio2_solved.c el cual implementa la misma funcionalidad que el ejercicio2 solo que está coordinada a través de semáforos.

Este ejercicio ha sido bastante complejo tanto a nivel de código como conceptual, ya que hemos tenido que realizar la correcta sincronización de muchos procesos y además hemos tenido que aprender a usar de forma correcta las funciones de memoria compartida que C tiene. Esto ha requerido bastante tiempo de lectura de los manuales correspondientes, además de bastante tiempo de depuración del ejercicio (debido a la gran cantidad de código pedida).

En conclusión, ha sido un ejercicio bastante completo que nos ha ayudado a consolidar los conceptos de exclusión mutua y memoria compartida vistos en clase de teoría.

- **Ejercicio3**

En este ejercicio implementamos el algoritmo para la sincronización de procesos "Productor-Consumidor", el cual es útil cuando un proceso "produce" cierto recurso, información... etc. y otro proceso desea "consumirlo", todo esto de forma sincronizada y asegurándonos de que va a haber exclusión mutua de la zona crítica "El almacén" y también de que no se va a producir ningún interbloqueo. Para esto necesitamos hacer un buen uso de los elementos software y del Sistema Operativo.

Este algoritmo está implementado en las trasparencias de Teoría, con lo cual el esfuerzo más significativo en la realización de este ejercicio ha sido el traducir este seudocódigo a C, ya que el manejo de semáforos en este lenguaje no es muy intuitivo.

Al ser un ejercicio de importante envergadura "Hablando de extensión de Código" siempre esto supone un esfuerzo extra a la hora de que todo compile y ejecute de forma adecuada, ya que puede haber muchos fallos y la depuración de ejercicios largos conlleva largos periodos de tiempo.

En conclusión, ha sido un ejercicio muy completo y didáctico para comprender cómo llevar la implementación de los algoritmos dados en la teoría a la práctica, lo cual nos hace dar un gran salto hacia la verdadera implementación y comprensión de dichos algoritmos.

Miguel Arconada Manteca y José Manuel Chacón Aguilera

- **Ejercicio4**

En este ejercicio se nos introduce la noción de mapeo de memoria, el cual es llevado a cabo por la función de C *mmap*.

En este ejercicio se nos pide que un hilo cree un fichero de texto en el cual escribe números al azar. La "gracia" de este ejercicio es implementar el código necesario para que un segundo hilo con la función *mmap* sea capaz de acceder a la información creada por el primer hilo, y sea capaz de modificarla e incluso imprimirla por pantalla.

No ha sido un ejercicio difícil, porque en sí no había mucho código que escribir. La dificultad de este ejercicio reside en hacer un buen uso de la función *mmap* (que es bastante compleja). Para ello ha sido necesaria una lectura atenta tanto de la información que se nos da en el enunciado de la práctica, como del propio manual de la función (invocado con "*man mmap*"). No ha dado problemas a la hora de hacer depuración ya que no tenía una extensión muy grande como es el caso de otros ejercicios desarrollados tanto en esta práctica como en prácticas anteriores.

- **Ejercicio 5**

En este ejercicio se nos pedía implementar una cadena de montaje, en la cual los procesos hijos leen de la cola de mensajes los datos (excepto el primero que lo lee de un fichero), lo modifican, y envían el resultado a la cola de mensajes para que lo modifique el siguiente.

Utilizamos una cola compartida por todos los procesos, con diferentes tipos de mensaje para cada par de procesos. Esto presentó una dificultad, puesto que inicialmente, nuestra condición de parada para los bucles de los procesos era que la cola estuviera vacía. Sin embargo, esto producía que el proceso B no terminara, puesto que quedaba un mensaje por leer en la cola (a leer por el proceso C), y se bloqueaba en el *receive*, y no terminaba nunca.

Una primera solución que pensamos fue utilizar dos colas diferentes, lo cual no nos pareció correcto, puesto que en proyectos con más tipos de mensajes en inviable por uso de memoria y dificultad añadida al programar. Por lo tanto, tuvimos que hacer uso de señales y flags globales para poder solucionarlo.