PRÁCTICA 3 – SOPER
Pareja 1
Alba Ramos
Andrea Salcedo

Todos los ejercicios que hay que entregar se compilan haciendo make nombre_archivo.c (este nombre se indica al final de cada apartado). También podemos compilar todos a la vez haciendo make.
Para ejecutarlos, se escribe ./nombre_archivo, sin el '.c'.

EJERCICIO 2

a) y c) Para este ejercicio hemos tenido que crear e inicializar un semáforo y la memoria compartida para la estructura proporcionada en el enunciado del ejercicio. Esta memoria es creada por el proceso padre que posteriormente compartirá con los procesos hijo. Tras esto, decidimos crear un semáforo para controlar el acceso a la zona restringida. Desde un principio, nosotras pensamos que el código debía implementarse con semáforos, ya que tanto el proceso padre como los procesos hijos van a necesitar acceder a la zona crítica del programa, y es por esto que los usamos, para que los procesos no se solapen.
A continuación, inicializamos el contenido de la estructura, a los valores indicados. Donde el id y el id previo de la estructura Cliente, están inicializados a 0 y -1, respectivamente.
Una vez hecho esto, el padre creará n hijos mediante la llamada a la función fork(), cuyo valor es dado por argumento desde la terminal.
Si el proceso creado es el hijo, seguimos las indicaciones dadas en el enunciado, añadiendo el semáforo cuando el proceso hijo quiera acceder a la estructura Cliente para escribir los datos. Es entonces, justo cuando termina de escribir, cuando se realiza un up() del semáforo y el hijo envía la señal SIGURS1 al padre.
Si el proceso creado es el padre, usamos un manejador, que reciba la señal enviada por el hijo, en el cual tendremos modificada una variable global de tipo volátil inicializada a 0. Esta variable es necesaria, para que el proceso padre pueda esperar con un pause() a la llegada de la señal enviada por el hijo, donde mientras esta variable(llamada salir), sea igual a 0, el bucle while continúa. Cuando salir sea igual a 1, termina el bucle y el proceso padre leerá de la estructura lo que el proceso hijo a escrito, y a su vez escribirá por la terminal los datos leídos, todo ello controlado por el semáforo, el cual hace un down() para poder acceder y un up() cuando termina de escribir por pantalla. (***ejercicio2.c***)

b) Decidimos implementar el anterior apartado con semáforos, ya que, tanto el proceso padre como los hijos deben acceder a una zona crítica. Sin el uso de los semáforos, todos los procesos podrían acceder a la vez, lo que implica que no se modifiquen correctamente las variables de la estructura Cliente, dadas en los procesos hijos, las cuales ahora se encuentran en la zona crítica.

EJERCICIO 3

a) Para este ejercicio hemos tenido que crear e inicializar los semáforos y la memoria compartida para la cola (para la cual hemos utilizado los ficheros queue.c y queue.h, que implementan una cola circular) en el productor. Tras esto, el productor lanzará un hijo (esto es así ya que el productor es el primero que debe ejecutarse, pues es el que crea los recursos que el consumidor va a utilizar), y con un exec ejecutará el programa del consumidor. Para controlar el acceso a la sección crítica (introducir y sacar productos de la cola) hemos utilizado el semáforo sem_recurso. Para controlar que el consumidor consuma sólo si existen productos en la cola, hemos utilizado el semáforo sem_lleno (cada vez que el productor crea un producto, hace un up en este semáforo). Del mismo modo, para controlar que el productor produzca sólo si la cola no está llena hemos utilizado el semáforo sem_vacio (cada vez que el consumidor consume, incrementa este semáforo, que está inicializado al valor del máximo de elementos que puede albergar la cola -menos uno, por ser cola circular-). En el programa se solicita al usuario meter caracteres (para terminar, debe introducir EOF mediante ctrl+d). Los puede introducir separándolos por \n (y verá la ejecución paso a paso de cómo se produce y se consume el carácter), o introducir unos cuantos separados por espacios y luego introducir el \n (esto es útil para ver cómo la cola está bien implementada, pues se puede observar que cuando la cola se llena, no se vuelve a producir hasta que el consumidor no haya extraído elementos). (***ejercicio3_productor.c***), (***ejercicio3_consumidor.c***)

b) Para hacer que la cola esté en un fichero en vez de en memoria compartida, hemos sustituido las llamadas a shm_open y a shm_unlink por open y unlink respectivamente (también le hemos cambiado el nombre a la variable fd_shm por fd simplemente, para mayor claridad), dejando el resto como estaba. La macro SHM_NAME también la hemos cambiado, simplemente eliminando '/' del nombre (el cual era necesario si usábamos memoria compartida, pero nos da error de permisos si utilizamos un fichero normal, ya que estos no pueden comenzar por ese carácter -se puede comprobar el error si lo intentamos crear a mano con ese nombre en el explorador de archivos-).
(***ejercicio3_productor_b.c***), (***ejercicio3_consumidor_b.c***)

EJERCICIO 4

En el enunciado aparece repetido como Ejercicio 3, pero nosotras lo hemos llamado Ejercicio 4.
Para el paso de mensajes a la cola estamos utilizando una estructura llamada Mensaje, que consta de un texto (de tamaño máximo TAMPAQUETE) y un entero indicando la longitud del mismo (o -1 si es el mensaje indicador de fin de fichero).

a) Este proceso se encarga de trocear el fichero recibido por argumento en paquetes de tamaño TAMPAQUETE. Para evitar tener que leer el fichero entero utilizando un bucle (procedimiento que vemos ineficiente, dado el tamaño de los paquetes especificado), hemos obtenido el descriptor de fichero y, a partir de él, toda la información del fichero mediante la función fstat: esto nos permitirá saber el tamaño en bytes del fichero. Después, simplemente copiaremos en una cadena, con mmap, el contenido indicado por el descriptor, cuyo tamaño obtenido con fstat.
Una vez que tenemos el texto del fichero en la cadena, calcularemos con una operación de

división y módulo cuántos paquetes de tamaño TAMPAQUETE habrá, y veremos si sobran bytes (algún paquete de ese tamaño no ha podido completarse. Es importante remarcar que este resto sí acaba con un \0, al contrario que los paquetes, y por tanto es necesario reservar memoria suficiente con memcpy a la hora de establecer el texto del mensaje a enviar). Todo esto se envía a la primera cola. Además, cuando se ha procesado el fichero entero se envía un mensaje especial que indica fin de fichero. (***ejercicio4A.c***)

b) Este proceso se encarga de procesar y codificar todos los mensajes leídos de la primera cola. Para ello, va a leer en un bucle todos los mensajes y va a recorrer cada carácter con otro bucle, aplicando el algoritmo de codificación (el cual tiene en cuenta si la letra es mayúscula o minúscula, o si es una Z que pase a ser una A) a cada uno, transformando toda la cadena, que finalmente envía a la segunda cola. En este caso, hay una primera lectura de mensaje fuera del bucle más externo: esto es porque, si nos han pasado una cadena vacía, no queremos entrar a procesar y a hacer comprobaciones para ella, simplemente enviar que la cadena era vacía y acabar. (***ejercicio4B.c***)

c) Este proceso se encarga de mostrar por pantalla el mensaje final, codificado. Para ello, leerá de la segunda cola cada mensaje en un bucle (hasta llegar al del fin de fichero, identificado por la longitud -1) y lo mostrará por pantalla (se realizan las recepciones de mensajes en un do-while, ya que no nos importa imprimir también el mensaje de fin de fichero, por ser su texto un espacio en blanco, y nos parecía más eficiente que tener que recibir un mensaje fuera del bucle, comprobar que no fuera el fin de fichero -por si te pasaban una cadena vacía- y repetir la llamada a la función mq_receive dentro del bucle de nuevo). (***ejercicio4C.c***)

d) Para el programa principal, hemos utilizado varias llamadas a fork y hemos hecho llamadas a execlp para ejecutar cada uno de los tres hijos. La complicación de este apartado ha venido, sobre todo, al ejecutar a los hijos. La llamada a execlp recibe como primer parámetro el nombre del programa a ejecutar (nosotras estábamos pasándole el fichero .c al principio), y luego una lista con los argumentos del programa (nos resultaba más sencillo escribir la lista en cada llamada a exec, ya que cada programa recibe unos argumentos diferentes. Podríamos haber usado execvp y pasarle un array con los argumentos, pero creíamos que era más eficiente hacer la lista y así evitar tener que rellenar un array distinto cada vez). Es importante destacar que el primer parámetro de esta lista debe ser el propio nombre del programa, ya que en el main se recibe el array de argumentos y el primero es, precisamente, este.
En este mismo programa se crean las colas, y así los hijos ya las tienen creadas y disponibles para usar, abriéndolas en el modo deseado (lectura o escritura). Se han creado tanto para lectura como para escritura porque cada hijo va a usarlas para ambos fines. (***ejercicio4.c***)