Javier Delgado del Cerro y Javier López Cano
Pareja 6

MEMORIA PRÁCTICA 3

Ejercicio 2:

Apartado 1 (Ejercicio2.c):
Generamos el programa pedido Ejercicio2, en el que el padre declara su función de control para la señal SIGUSR1 captura() (explicada posteriormente), obtiene una key para una memoria compartida y crea esta con un nombre de usuario y un id que inicializa a 0, tras esto crea N hijos (mediante un bucle for) cada uno de los cuales duerme un tiempo aleatorio entre 1 y 5 segundos (empleando la función aleat_num empleada tambien en la práctica anterior) y pide por pantalla el nombre de un nuevo usuario y lo guarda en la memoria compartida incrementando el id que haya en esta en 1. Después envía la señal SIGUSR1 al padre para que este ejecute la funcion de control de esta señal, que se encarga de leer los datos de la memoria compartida (nombre de usuario e id) e imprimir por pantalla “El usuario: nombreUsuario tiene id numeroId”. Tras esto el padre realiza tantos wait como hijos haya creado (mediante otro bucle for) para esperar a que todos finalicen (durante este periodo los hijos le seguiran mandando señales y debera ir ejecutando la funcion de control) y por último liberará la memoria compartida y cabará el proceso con éxito.

Para que este programa funcione correctamente es necesario crear la función captura() que es la función de control del padre para la señal SIGUSR1, en esta funión se declara la zona de memoria para poder emplearla, se lee de ella los datos y se imprimen por pantalla con el formato explicado anteriormente, tras lo cual se sale de esta funcón con éxito. De este modo cuando algún hijo mande al padre la señal SIGUSR1 este imprimirá por pantalla los datos que haya en la memoria compartida.

Este programa, sin embargo, tiene un problema, ya que no se controla de ningún modo que varios procesos no puedan acceder a la vez a la zona de memoria compartida, para conseguir evitar esto se deben emplear semáforos como se hace en la segunda parte de este ejercicio Ejercicio2_solved.

Apartado 2 (Ejercicio2_solved.c):
En este caso implementamos el programa Ejercicio2_solved que realiza el mismo proceso que el Ejercicio2 pero empleando semáforos para controlar el acceso a la zona de memoria compartida.

De este modo el padre declara su función de control para la señal SIGUSR1 captura(), obtiene una key para el array de semáforos y otra para la memria compartida, crea el array de semáforos con 1 semáforo que inicializa a 1 (empleando para ello un array auxiliar con el valor que debe tomar el semáforo) y crea la memoria compartida tras lo cual crea a los N hijos del mismo modo que en Ejercicio2.

En este caso los procesos hijo realizan el mismo procedimiento que en el apartado anterior con la diferencia de que, antes de acceder a la memoria compartida para modificarla, hacen un down del semáforo creado para evitar que otros procesos puedan acceder a esta memoria al mismo tiempo, y al acabar de modificarla (antes de mandar la señal SIGUSR1 al padre) hacen un up del mismo para que el resto de procesos puedan volver a acceder a la memoria compartida.

El proceso que realiza el proceso padre tras crear a los hijos tan solo varía en 2 factores. El primero es que en la funcion de control de la señal SIGUSR1 captura() debe realizar un down del semáforo creado antes de acceder a la memoria compartida para leer los datos, y un up del mismo al terminar de acceder a la emoria. El segundo y último cambio necesario es que al terminar debe liberar la

memoria del semáforo creado así como la del array empleado para inicializar el valor del semaforo a 1. Este último array mencionado deben liberarlo tambien los hijos puesto que al haberse creado antes de que se hiciese el fork() para crear a estos, la memoria se habrá duplicado en estos y, por lo tanto, tendrán también memoria reservada para este array.

Ejercicio 3:

Para la realizacion de este ejercicio creamos el archivo Ejercicio3.c, que implementa un problema de productor-consumidor.
En primer lugar el proceso padre obtiene una key para un array de semáforos y otro para una memoria compartida, crea la memoria compartida con un array de N caracteres y un int contador, inicializando este último a 0; y crea el array de semáforos con 1 semáforo que inicializa a 1 empleando un array auxiliar con el valor para inicializar el semáforo. Tras esto crea un hijo mediante la función fork(), el padre será el productor y el hijo el consumidor.

Tras crear el hijo, el padre ejecuta continuamente la función produce_item() hasta que esta devuelva 0(se ha terminado de producir los elementos) o -1(se ha producido un error), en ambos casos libera la memoria del array de semáforos, de la memoria compartida y del array auxiliar empleado para inicializar el array de semáforos, espera a que termine el proceso hijo con un wait y termina su ejecución, con la diferencia de que si devuelve -1 sale del proceso con un error, y si no, con éxito.

Mientras tanto el hijo realiza el mismo procedimiento en lugar de con la función produce_item(), con la función consume_item(), con la diferencia de que al terminar su ejecución solamente debe liberar el array auxiliar para inicializar el array de semáforos (por el mismo motivo que en el ejercicio anterior) y no debe esperar a ningún hijo ni liberar la memoria del array de semáforos ni de la memoria compartida pues de esto ya se encarga el padre. Del mismo modo que el padre, si la función consume_item() devuelve 0 sale con éxito y si devuelve -1 termina con un error.

La funciónproduce_item() que realiza el padre accede a la zona de memoria compartida y va añadiendo un carácter nuevo al array que hay en esta cada vez que accede. El caracter nuevo que se introduce se obtiene sumandole 1 al ultimo caracter añadido convirtiendolo así en la siguiente letra del abecedario y cuando llega a la z comienza con los números del 1 al 9, al acabar devuelve 0. Cada vez que accede a la zona de memoria compartida baja el semáforo y al terminar de acceder a esta lo sube, de este modo se evita que el padre y el hijo accedan a esta al mismo tiempo.

La función consume_item() que realiza el hijo consiste unicamente en acceder a la memoria compartida (haciendo un down del semáforo antes y un up al acabar de acceder a ella) e imprimir por pantalla los caracteres que haya en el array de esta, una vez ha impreso todos, devuelve 0.

Para facilitar la implementación de estas 2 funciones hemos añadido un int contador a la memoria compartida, que indica el número de caracteres presentes en el array de la memoria compartida, por tanto cada vez que produce_item cree un nuevo carácter, incrementa este contador en 1, y cada vez que consume_item consuma uno, lo decrementa. Este contador se emplea para hacer que el hijo no realize ninguna acción (espera activa) cuando el array esté vacío, y el padre realize lo mismo cuando el array esté lleno.

De este modo, la salida de este programa deberá simplemente imprimir por pantalla las letras de la a a la z y los números del 0 al 9.

Javier Delgado del Cerro y Javier López Cano
Pareja 6

Ejercicio 4

Para este ejercicio creamos el archivo Ejercicio4.c que implementa un programa que se encarga de declarar 2 hilos y crearlos de modo que el primero ejecute la función writeFile, luego se espera a que este acabe empleando pthread_join(), y tras esto se hace que el segundo ejecute la función readFile tras lo cual se espera de nuevo con pthread_join a que este acabe y se acaba la ejecución del programa.

La función writeFile que ejecuta el primer hilo abre un nuevo fichero con el nombre fichero.txt (si no existe lo crea) e imprime en este entre 1000 y 2000 números aleatorios entre el 100 y el 1000 separados por comas (para obtener el número de números que se imprimen asi como cada número en concreto se ha empleado la función aleat_num ya empleada en el ejercicio 2 y en la práctica anterior que devuelve un número aleatorio entre 2 valores dados por parámetro), tras lo cual cierra el fcihero y termina su ejecución devolviendo NULL.

La función readFile que ejecuta el segundo hilo abre el fichero fichero.txt mediante la función open que devuelve el descriptor de fichero. Tras esto lee el fichero por completo con la función mmap que devuelve un void* con todos los caracteres del fichero, porlo que al emplearla hacemos un casting a char*. A la función mmap se le debe pasar el tamaño del fichero, para ello empleamos la función fstat que nos devuelve el tamaño de este en bytes. Tras leer el fichero con mmap, lo modificamos cambiando las comas del array obtenido al leerlo por espacios, e imprimimos el array modificado por pantalla, al terminar cerramos el map con munmap de modo que ya no se pueda modificar el fichero a traves del array obtenido con mmap, por último se cierra el fichero con close y se acaba la ejecución del hilo devolviendo NULL.

De este modo la salida del programa consistirá en imprimir por pantalla entre 1000 y 2000 números entre el 100 y el 1000 separados por espacios y obtener un fichero fichero.txt con estos mismo números separados por espacios.

Ejercicio 5

En este ejercicio realizamos el archivo Ejercicio5.c que implementa un programa que emplea una cola de mensajes para sincronizar 3 procesos A, B y C.

De este modo lo primero que realiza el programa es obtener una key para la cola de mensajes mediante ftok, y crear esta cola con msgget (obteniendo asi el id de la cola que se empleara para acceder a ella y modificarla). Tras esto el proceso crea un hijo con fork(), que será el proceso A.

Este hijo (proceso A) abrirá un fichero que se habrá pasado como parámetro con fopen y lo va leyendo en trozos de 16 bytes, estos trozos que va leyendo los va añadiendo a la string que hay en una estructura message declarada anteriormente que, ademas de este char[16] tiene un long que indica el tipo de mensaje y un int llamado flag que indica si es el ultimo mensaje que un proceso debe leer. El proceso tras introducir el char[16] en la estructura, establece el tipo del mensaje en 1 (porque de este modo podemos hacer que más tarde el proceso B lea solo los de tipo 1) y el flag a 0 (para indicar que no es el último mensaje) tras lo cual añade el mensaje a la cola de mensajes con msgsnd. Realiza este procedimiento hasta acabar de leer el fichero completo (con un bucle) tras lo cual envia a la cola un último mensaje de tipo 1 con flag 1(para indicar que es el último) y terminará su ejecución con éxito.

Mientras tanto el padre habrá creado otro hijo con fork() que será el proceso B. Este hijo irá recibiendo en un while(1) los mensajes del tipo 1 de la cola con msgrcv y los irá modificando cambiando cada letra del char[16] del mensaje por la siguiente del abecedario (sumandole 1) y cambiará el tipo del mensaje a 2 (para que mas tarde el proceso C lea solo los de tipo 2), pondrá la bandera a 0 y subirá los mensajes a la cola con msgsnd. Antes de modificar los mensajes comprueba si el flag del mensaje que ha recibido es 1, y si este es el caso sale del bucle while (con un break) y envía a la cola un mensaje del tipo 2 con el flag 1 (que será el último mensaje que tenga que leer el proceso C) y terminará su ejecución con éxito.

Al mismo tiempo, tras crear los 2 hijos el proceso padre pasará a ser el proceso C. Este proceso recibirá en un while(1) los mensajes de tipo 2 de la cola de mensajes con msgrcv e irá imprimiendo el contenido del char[16] de estos mensajes en un fichero de salida que se pasa también como parámetro de entrada al programa, para ello abrirá el fichero con fopen e irá imprimiendo en este las cadenas de caracteres con fprintf. Al recibir los mensajes irá comprobando si el flag es 1, y si es 1 saldrá del bucle while, cerrará el fichero con fclose y esperará con 2 waits a que terminen los procesos hijo, tras esto libera la memoria de la cola de mensajes con msgctl pasandole como parámetro el proceso IPC_RMID y terminará su ejecución con éxito.

De este modo a la salida deberíamos obtener un fichero que sea igual que el fichero de entrada pero cambiando todas las letras por su siguiente en el abecedario.

En nuestro caso para probar el funcionamiento de este programa hemos creado un fichero source.txt que es el fichero que empleamos como prueba para que lea el fichero A. Este fichero se proporciona junto con el código pero se puede probar con cualquier otro pasandolo como parámetro al programa.