Practica 3
Sistemas Operativos

Grupo: 2214

Cristian Tatu Pareja: 3 Sara Sanz

Semana 1

Memoria Compartida

Ejercicio 1:

En este ejercicio se utiliza la función *semget()* para devolver un identificador para la entrada a la región de memoria a compartir indicada por *key*, además entre los *flags* que se incluyen como parámetro se encuentra *IPC_CREAT*, que creará la región de memoria.

Lo primero que se hace es comprobar el resultado devuelto por esta función.

Si todo va bien, se habrá creado correctamente el identificador de la región de memoria y se imprimirá el mensaje: *"Nuevo segmento creado", p*ero en el caso de que este valor sea -1, significará que la región de memoria ya estaba creada; que se indicará con el mensaje correspondiente; y por lo tanto solo habría que abrir el segmento de memoria ya existente. Para esto se vuelve a utilizar la función *semget()*, solo que esta vez se pasará 0 como argumento en lugar de los *flags* como *IPC_CREAT*. En este caso también se comprobará el resultado devuelto por la función, que si es -1, significará que ha habido algún error a la hora de abrir el segmento y sino, se dará por hecho que el segmento se ha abierto correctamente.

Ejercicio 2:

En este ejercicio se plantea que se genere un programa cuyo padre tenga *N* hijos. Tanto padre como hijos compartirán una región de memoria en la que se encontrará una estructura con un nombre y un identificador. Cada hijo pedirá por pantalla que se escriba un nombre y aumentará el valor del identificador en 1, tras lo cual, dormirá un tiempo aleatorio y mandará una señal al proceso padre (*SIGUSR1*). Para esto, se armará un *manejador* para la señal anterior, que cuando sea recibida, imprimirá los campos de esta estructura.

Para realizar este ejercicio lo primero que hacemos es crear la memoria compartida para la estructura tal y como se ha hecho en los ejemplos de la práctica utilizando la librería de memoria compartida que hemos creado. Tras crear la memoria, armamos la señal SIGUSR1 con su *manejador*, en el que simplemente se hará un *attach* para la memoria compartida creada y se

imprimirán los valores de los campos de la estructura. Tras esto, se hace un *fork()* para generar los *N* procesos y se comprueba si el proceso ejecutando el programa es el hijo o el padre, si es el hijo, pedirá que se introduzca un nombre por pantalla que se guardará en la estructura para la que se ha generado memoria compartida. Después este mandará la señal *SIGUSR1* al padre que realizará las acciones descritas en su *manejador*.

Una vez planteado y ejecutado el programa en C, se puede ver que este no funciona como debería, los hijos piden que se introduzcan las dos cadenas a la vez y se entremezclan las funcionalidades ejecutadas por cada proceso. Esto se debe a una falta de sincronización entre los procesos, ya que si no tenemos recursos que controlen los tiempos de ejecución de cada uno, no se puede predecir en qué orden se van a ejecutar o si se van a ejecutar y sobreescribir en las variables compartidas unos a otros.

Para solucionar esto, empleamos semáforos que den permiso a escribir a cada proceso en el tiempo adecuado. Por ello, creamos un semáforo y lo inicializamos a 1, este semáforo se situará justo antes de la impresión por pantalla que nos pedirá introducir un nombre, haciendo un *Down*, que provocará que el semáforo tenga valor 0, y por lo tanto quede bloqueado y no deje acceder a esa región a ningún otro proceso hasta que se haga un *Up*, que se situará justo después de enviar la señal al proceso padre.

De esta forma, se controlará que solo un proceso pueda escribir en la región de memoria compartida a la vez y así se solucionarán los problemas de concurrencia.

De este ejercicio cabría destacar el aprendizaje de la importancia del uso de semáforos, ya que aunque lo hemos podido comprobar en la práctica anterior, al hacer este ejercicio se puede comprobar que cuando usamos memoria compartida el uso de semáforos va de la mano. Al plantear el ejercicio queda claro que es imposible utilizar variables compartidas modificadas por distintos procesos en el orden y el tiempo que utilizan cada vez ya que estos no se pueden predecir y por lo tanto tenemos que ser nosotros los que definen su orden para evitar posibles problemas con la sincronización de procesos a la hora de escribir o leer de memoria compartida a la vez.

Ejercicio 3:

Para resolver el problema del productor consumidor nos harán falta 3 semáforos. La implementación por la que hemos optado ha sido crear cada semáforo con una longitud igual al numero de elementos del buffer. Así podemos tener mas control sobre cada posición del array y quien lee y escribe en cada una.

El primer semáforo implementado es el que nos garantiza la exclusión mutua, *mutex*, garantizándonos que solo uno de los dos procesos accederá a la posición de memoria especifica en cada momento.

Dado que el consumidor no podrá leer nada a menos que el productor no haya colocado nada en la posición correspondiente y que el productor no pueda escribir nada a menos que el consumidor haya leído alguna posición, nos requiere el uso de otros 2 semáforos que nos indica que posición del array se puede leer y cual se puede escribir.

Si el consumidor es el primero que empieza hará un *down* en el semáforo *sePuedeLeer* en el índice correspondiente que solamente estará a 1 si el productor pasó antes por ahí para escribir algo. Así garantizamos que el consumidor tenga que esperar al productor si empieza antes.

Al igual pasa si es el productor el que empieza primero, aunque realmente el único problema que habría sería que el productor vaya mucho más rápido que el consumidor y que llegue a alcanzarle. Entonces el productor tendría que esperarle antes de poder escribir nada. Esto lo logra haciendo un *down* en el semáforo *sePuedeEscribir* que inicialmente esta inicializado a 1 para todas las posiciones del array, por tanto, cuando el consumidor lea ese elemento volverá a poner esa posición a 1 para que el productor pueda escribir otra vez.

Para poder contemplar los dos casos hemos añadido un argumento más al ejecutable para distinguir cual de los dos procesos va a empezar primero.

Observamos en la *Figura1.2* que, al haber empezado el productor primero, ha llegado a alcanzar al consumidor y por tanto sus índices están muy cerca uno del otro pero lo que es mas importante es que el índice del productor está por debajo del consumidor como esperábamos.

En la *Figura2* observamos lo contrario. Al haber empezado el consumidor llega al alcanzar al productor y por tanto sus índices están juntos pero el del consumidor es más bajo.

Ejercicio 4:

En este ejercicio se pide que se cree un hilo que invoque a una función que escribirá en un fichero *n* números aleatorios siendo *n* también un número aleatorio. Tras esperar a este hilo se invocará a otro que leerá el fichero y hará un *mapeo*. Una vez *mapeado*, el hilo cambiará los espacios del fichero por comas a partir de la dirección donde está *mapeado*.

Para realizar el ejercicio hemos creado una función que genera un número aleatorio dado un límite inferior y uno superior. Además tendremos dos funciones para trabajar con el fichero, una que escriba en él, utilizando la función que genera números aleatorios y la otra que leerá este fichero y cambiará los espacios por comas.

Para esto en el *main* crearemos un hilo que invoque a la primera función y tenga como argumento el fichero que se pasa como argumento al ejecutar el ejercicio y le esperaremos con *pthread_join()*, tras esto crearemos otro hilo para invocar a la función que lee el fichero y a este hilo se le pasará como argumento el descriptor del fichero que queremos leer.

En la función que lee el fichero se hará el *mapeo* del contenido del fichero, primero obteniendo su tamaño con la función *fstat()* a la que se le pasará el descriptor del fichero y la estructura *stat, y después usando st_size junto con la estructura stat para sacar la variable del tamaño. U*na vez obtenido el tamaño se utilizará la función *mmap()* para sacar la variable *map* del *mapeo* mediante la que se modificará el fichero sin necesidad de utilizar el fichero en sí. Usando la variable *map* como un *array* este se recorre comprobando si el carácter es un espacio y si lo es se cambiará por una coma. Para actualizar los cambios realizados en el fichero se usará la función *msync()* al que se le pasará la variable *map* y el tamaño de esta.

Finalmente se liberará el *mapeo* con la función *munmap()* y cuando el hilo que había invocado a esta función (leer fichero) termina, ya que hemos usado

pthread_join() para esperarlo, se cerrará el fichero finalmente y terminará el programa.

En este ejercicio queda visible las facilidades que nos dan el uso de *mapeo* ya que con este no es necesario acceder al fichero en sí para leerlo, modificarlo y actualizar las modificaciones. El *mapeo* es una forma mucho más sencilla de usar ficheros, leer y escribir en ellos.

Semana 2

Colas de Mensajes

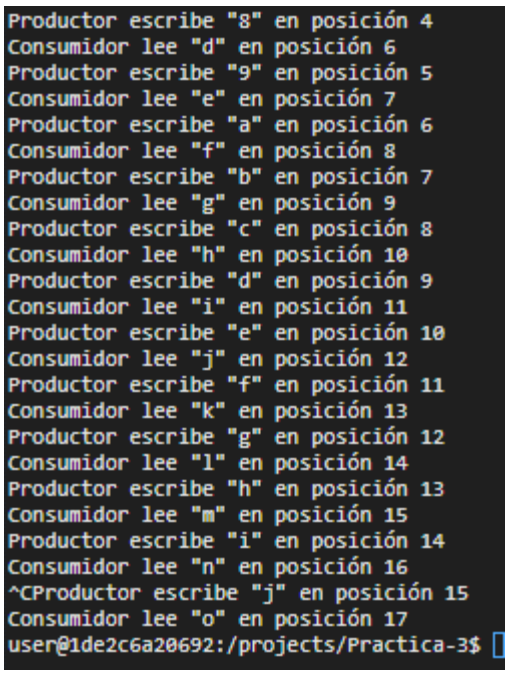
Ejercicio 5:

En contraste con las decisiones de diseño tomadas, otra solución al problema implementaría también colas de mensajes, pero solamente una. Ya que podemos darles a los mensajes un identificador distinto y así saber que mensajes pertenecen a que parte de la cadena de montaje.

Por ejemplo, los mensajes que se envían desde A a B podrían tener ID = 1 y los que se envían de B a C tener ID = 2. Así B solo leería los mensajes de tipo 1 y C los de tipo 2 sin tener que crear 2 colas de mensajes. Pero del enunciado hemos entendido que cada par de procesos que se van a comunicar deben tener una cola diferente.

Para la prueba del programa se ha generado antes un fichero (*random.txt*) que contiene solamente caracteres alfabéticos minúsculas y mayúsculas con el recurso de Linux *urandom*.

ANEXO:



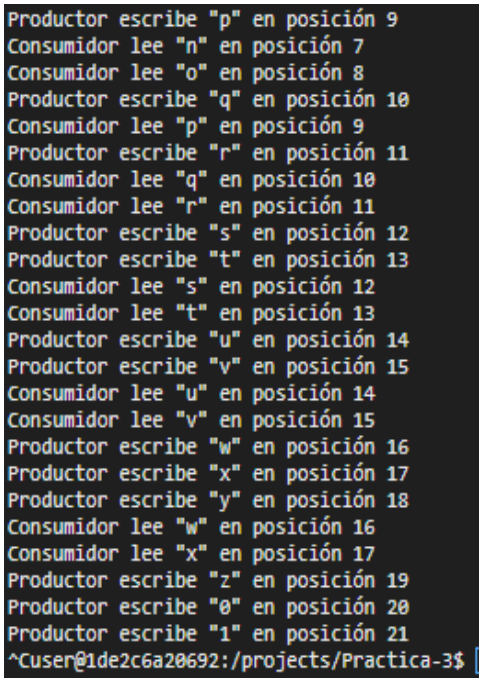


Ilustración 1.2[El Consumidor Empieza]