Unidades Didácticas 1-3: Ejercicio de feedback 1

Sistemas operativos

Para aplicar los conocimientos adquiridos en la asignatura se propone la realización de este ejercicio feedback que corresponde a los contenidos desarrollados en las Unidades Didácticas 1, 2 y 3.

1. Objetivo

El objetivo del ejercicio es que el estudiante practique y adquiera destreza en el desarrollo de programas y aplicaciones de usuario escritas en lenguaje C. Así mismo, se pretende que los estudiantes hagan uso, en sus programas, de las funcionalidades que, a través de diferentes llamadas al sistema, ofrece el sistema operativo para la gestión de procesos y hebras. Finalmente, la última parte del ejercicio se centra en la elaboración de cronogramas que muestren el orden en que acceden al procesador los procesos de una determinada carga de trabajo en función de la política de planificación aplicada y de los parámetros de la misma.

2. Enunciado Ejercicio 1

Realice los siguientes apartados escribiendo el código necesario en lenguaje C:

2.1. Apartado 1

Declarar una estructura, de nombre ***arrayLength***, que disponga de tres campos: un array de enteros positivos (*arrInt*) con capacidad para 10 valores y, además, el número de elementos almacenados en el mismo (*arrSize*) y la suma de los elementos contenidos en el array (*arrAdd*). Convertir la estructura en un tipo de datos (arrayLength_t) con la cláusula typedef.

2.2. Apartado 2

Implementar una función de nombre ***initArray*** que reciba por parámetro un puntero a una estructura del tipo *arrayLength_t* y ponga a valor -1 todas sus posiciones. A los campos *arrSize* y *arrAdd* se les asignará el valor 0. La función devolverá 0 si no se produce ningún error y -1 en caso contrario.

Un valor -1 en una posición del array *arrInt* indicará que NO hay un valor a computar en esa posición.

Implementar una función de nombre ***printArr*** que reciba por parámetro un puntero a una estructura del tipo *arrayLength_t*.

La función mostrará por pantalla el contenido de la estructura en el siguiente formato: {[elto0ArrInt, elto1ArrInt, …, elto9ArrInt], arrSize, arrAdd}

Ejemplo: Nada más inicializar el array, la función mostrará:

```
{[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1], 0, 0}
```

Los campos *arrSize* y *arrAdd* tienen valor 0 porque se considera que el array está vacío.

Implementar una función de nombre ***addElement*** que reciba por parámetro un puntero a una estructura del tipo *arrayLength_t* y un valor entero que se añadirá a los existentes en el array.

En primer lugar, se comprobará si el valor recibido es positivo. También se verificará si existe hueco en el array para el nuevo elemento y, si es así, se añadirá en la posición siguiente a la última ocupada.

Finalmente, se actualizará el número total de elementos añadidos al array (*arrSize*), así como su suma (*arrAdd*) en los correspondientes campos.

La función devolverá 0 si el elemento se ha podido añadir al array sin que se haya producido ningún error y -1 en caso contrario.

Se incluye el siguiente código a modo de ejemplo:

```
… //Código anterior
if ( initArray(&arr) != 0){
   printf("Array inicializado");
}else{
   printf("Error en inicialización");
}
printArr(&arr);
if  ( addElement(&arr, 22) == 0) {
   printf("Elemento añadido");
}else{
   printf("Error al añadir elemento");
}
printArr(&arr);
if  ( addElement(&arr, 44) == 0) {
   printf("Elemento añadido");
}else{
   printf("Error al añadir elemento");
}
printArr(&arr);
… //Código posterior
```

La ejecución de este código dentro de la correspondiente función *main()* hará que se por pantalla la siguiente salida:

```
{[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1], 0, 0}
{[22, -1, -1, -1, -1, -1, -1, -1, -1, -1], 1, 22}
{[22, 44, -1, -1, -1, -1, -1, -1, -1, -1], 2, 66}
```

2.3. Apartado 3

Implementar una función de nombre ***getArrSize*** que reciba por parámetro un puntero a una estructura del tipo arrayLength_t. La función devolverá el número de elementos en el array y -1 si se produce algún error.

Implementar una función de nombre ***getArrAdd*** que reciba por parámetro un puntero a una estructura del tipo arrayLength_t. La función devolverá la suma de los elementos almacenados en el array y -1 si se produce algún error.

Implementar una función de nombre ***getElement*** que reciba por parámetro un puntero a una estructura del tipo arrayLength_t y la posición de un elemento dentro del array. La función comprobará que la posición indicada se encuentra dentro de los límites del array y, además, está ocupada por un valor positivo. En tal caso se devolverá el valor almacenado en la posición indicada o el valor -1 en cualquier otra situación.

2.4. Apartado 4

Implementar una función de nombre ***setElement*** que reciba por parámetro un puntero a una estructura del tipo arrayLength_t, la posición de un elemento dentro del array y el valor a asignar a dicha posición.

La función comprobará que la posición indicada se encuentra dentro de los límites y que está ocupada por algún valor positivo. También se comprobará que el valor a introducir es positivo.

Si se cumplen las condiciones indicadas se procederá a sustituir el valor que ocupa la posición dada por el nuevo valor. Además, será necesario actualizar el valor del campo *arrAdd*

Si la posición no se encuentra dentro de los límites del array o no almacena un valor positivo se devolverá el valor -1. También se devolverá -1 en caso de que el valor recibido sea negativo.

2.5. Apartado 5

Implementar una función de nombre ***resetArr*** que reciba por parámetro un puntero a una estructura del tipo *arrayLength_t*. La función asignará el valor -1 a todas las posiciones del array, también asignará el valor 0 a los campos *arrSize* y *arrAdd*. Se tratará de hacer uso de las funciones previamente implementadas.

2.6. Apartado 6

Escribir una función ***main()*** para comprobar el funcionamiento de las funciones anteriores. Para ello:

1) Declarar dos estructuras de tipo *arrayLength_t* de nombre ***al1*** y ***al2***.
2) Utilizando las funciones de los apartados anteriores, hacer que el array de ***al1*** almacene los valores 0, 10, 20, …, 90 y se actualicen adecuadamente los valores *arrSize* y *arrAdd*.
3) Haciendo uso de las funciones de los apartados anteriores, mostrar la estructura almacenada en *al1*.

4) Haciendo uso de las funciones de los apartados anteriores, actualizar las posiciones **impares** del array para que almacenen, respectivamente, los valores 1, 3, 5, 7 y 9.

5) Mostrar de nuevo la estructura.

6) Utilizando las funciones desarrolladas, hacer que se añadan al array de *al2* las posiciones pares del array *al1*. Estas posiciones se deben almacenar de forma consecutiva en el array *al2*.

7) Utilizando las funciones desarrolladas, actualizar las posiciones finales del array de *al2* para que almacenen los valores de 0 a 4

8) Mostrar la estructura *al2*

3. Enunciado Ejercicio 2

Escribir el código C necesario para implementar los elementos indicados en los siguientes apartados:

3.1. Apartado 1

1) Los arrays utilizados en el ejercicio tendrán un tamaño de 10 elementos definido por una constante simbólica de nombre *SIZE*.

2) Se declarará un tipo de datos que corresponda a una estructura con tres campos enteros, uno almacenará el valor numérico correspondiente a la base de una potencia (*base*), otro a su exponente (*exp*) y un tercero (*potencia*) que consistirá en el resultado de elevar el valor *base* al de *exp*. El tipo se denominará ***potencia_t***.

3) Se declarará un tipo de datos (***potenciaP_t)*** correspondiente a puntero a la estructura definida en 2.

4) Se implementará una función de nombre **setBaseExp** que reciba por parámetro un puntero a una estructura *potencia_t* y dos valores enteros correspondientes, respectivamente, a la base y el exponente de una potencia. La función asignará los valores recibidos a los campos *base* y *exp* de la estructura y dejará el valor -1 en el campo *potencia*.

5) Se implementará una función de nombre ***getPotencia*** que reciba por parámetro dos parámetros enteros correspondientes, respectivamente, a la base y al exponente de una potencia. La función devolverá el resultado de $base^{exp}$. La implemtanción de la esta función **no** hará uso de ninguna otra definida en previamente en C como *pow*, *sqr*, etc..

6) Se implementará una función de nombre ***setPotenciaEst*** que reciba por parámetro un puntero a una estructura *potencia_t* y deje en el campo *potencia* el resultado de hacer $base^{exp}$. La función no devolverá ningún valor.

3.2. Apartado 2

1) Se implementará una función ***initArrayEst*** que reciba por parámetro un array de estructuras potencia_t y lo inicialice de forma que el campo *base* de cada posición almacene el valor correspondiente al índice de dicha posición más 1. Los valores *exp* de todas las posiciones se inicializarán a valor 0 y los valores *potencia*, también de todas las posiciones, a valor 1. La función no devolverá ningún valor. La situación del array recibido tras la ejecución de la función:

Índice	i:0	i:1	i:2	i:3	i:4	i:5	i:6	i:7	i:8	i:9
base	1	2	3	4	5	6	7	8	9	10
exp	0	0	0	0	0	0	0	0	0	0
potencia	1	1	1	1	1	1	1	1	1	1

2) Se implementará una función ***printArrayEst*** que reciba por parámetro un array de estructuras *potencia_t* y muestre por pantalla su contenido de la forma que se indica más abajo. La función no devolverá ningún valor.

   Nada más inicializar un hipotético array *arr*, la función *printArrayEst* mostraría la siguiente salida:

```
arr[0]: base:  1 exp: 0 potencia 1
arr[1]: base:  2 exp: 0 potencia 1
arr[2]: base:  3 exp: 0 potencia 1
…
arr[9]: base: 10 exp: 0 potencia 1
```

3) Se implementará una función de nombre calcuPotHeb para crear hebras sobre ella. Esta función, por tanto, recibirá un puntero a void (* void) que se convertirá a puntero a estructura potencia_t y también devolverá el tipo void *. La función obtendrá el resultado de baseexp y lo almacenará en el campo potencia de la estructura.

3.3. Apartado 3

1) Escribir el cuerpo de la función **main()** de manera que se declara un array (*arr1*) de estructuras *potencia_t* de SIZE elementos de longitud.

2) Utilizando las funciones ya definidas, inicializar el array *arr1*.

3) Utilizando las funciones ya definidas, mostrar el array *arr1*.

4) Utilizando las funciones ya definidas (**setBaseExp)**, hacer que el campo *base* de las estructuras del array *arr1* almacene el valor correspondiente a su índice. El campo *exp* de todas las estructuras almacenará el valor 2.

Índice	i:0	i:1	i:2	i:3	i:4	i:5	i:6	i:7	i:8	i:9
base	0	1	2	3	4	5	6	7	8	9
exp	2	2	2	2	2	2	2	2	2	2
potencia	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

5) Utilizando las funciones ya definidas, mostrar el array *arr1*.

6) Hacer que se creen SIZE hijos con la función ***fork()*** y hacer que cada hijo devuelva el resultado de calcular la potencia de una de las posiciones del array. El proceso padre quedará a la espera de la terminación de cada proceso hijo y se asignará el valor **obtenido por el proceso hijo (comunicado al padre mediante exit)** al campo *potencia* de la estructura correspondiente en el array *arr1* definido en el proceso padre (función *main*).

7) Utilizando las funciones ya definidas, mostrar de nuevo el array *arr1*.

8) Se declarará un nuevo array de estructuras *potencia_t* de nombre *arr2*.

9) Utilizando las funciones ya definidas, inicializar el array *arr2*.

10) Utilizando las funciones ya definidas, mostrar el array *arr2*.

11) Utilizando las funciones ya definidas (**setBaseExp)**, hacer que el campo *base* de las estructuras del array *arr2* almacene el valor correspondiente a su índice. El campo *exp* de todas las estructuras almacenará el valor 2.

12) Utilizando las funciones ya definidas, mostrar el array *arr2*.

13) Hacer que se creen SIZE hebras de forma que obtengan el resultado de $base^{exp}$ en cada estructura del array arr2 y lo asignen al correspondiente campo *potencia*.

14) Utilizando las funciones ya definidas, mostrar de nuevo el array *arr2*.

4. Enunciado Ejercicio 3

Sea un sistema operativo que mantiene un planificador basado en colas con distintos niveles de prioridad. Se utilizan 3 colas cuyos niveles de prioridad son, respectivamente, 1, 2 y 3. Las prioridades se encuentran ordenadas en orden decreciente, es decir, la ejecución de los procesos en la cola de prioridad 1 es más prioritaria para el S.O. que la de los procesos en la cola de prioridad 2.

La cola de prioridad 1 es gestionada con un algoritmo Round-Robin de cuánto tiempo **q=2 u.t.**, la de prioridad 2 es gestionada con un algoritmo Round-Robin de cuanto tiempo **q=3 u.t.** y, finalmente, la cola de prioridad 3 es gestionada por un algoritmo Round-Robin de cuánto tiempo **q=4 u.t.**

Se considerará que la prioridad de planificación es gestionada de manera expulsiva (*preemptive schedulling*) por parte del sistema operativo de manera que un proceso podrá ser expropiado del procesador a favor de otro más prioritario. En tal caso el proceso expulsado del procesador volverá al frente de la cola de donde partió.

Por otro lado, las prioridades se tratarán dinámicamente de forma que cuando un proceso agota el cuanto establecido para una cola es transferido a la cola correspondiente al nivel de prioridad inmediatamente inferior. Es decir, los procesos de la cola de prioridad 1 pasarían a la de prioridad 2, los de la cola de prioridad 2 a la de prioridad 3 y así sucesivamente.

Los tiempos de servicio, llegada al sistema y Prioridad de cada proceso se muestran en la siguiente tabla.

	PA	**PB**	**PC**	**PD**	**PE**
T. Llegada al Sist.	0	1	2	4	8
T. Servicio	6	4	3	6	2
Prioridad	1	2	2	1	1

Además, se tendrá en cuenta que el proceso **PD** al cabo de **una unidad de tiempo de ejecución** invoca una llamada al sistema que le mantiene bloqueado durante **2 u.t.**

Se pide

1) Obtener el cronograma de ejecución y los tiempos de retorno y espera de todos los procesos de tabla anterior en el escenario descrito.
2) Obtener el cronograma de ejecución y los tiempos de retorno y espera de todos los procesos suponiendo el mismo sistema de planificación basado en colas de prioridad pero todas gestionadas mediante la política FCFS o FIFO.

5. Instrucciones de entrega

Para la entrega de la propuesta de solución al ejercicio feedback se seguirán las siguientes indicaciones:

- **Se subirá un único archivo comprimido de nombre <NPAlumno>.zip**
- **En el archivo comprimido se incluirán 2 archivos de código fuente, ej1.c y ej2.c, con la propuesta de solución a los Ejercicios 1 y 2. También se incluirá una memoria en formato PDF que recoja los elementos más importantes realtivos al desarrollo del código de los Ejercicios 1 y 2, así como las correspondientes capturas de pantalla con los resultados de su compilación y ejecución. En la memoria, también se adjuntará la propuesta de solución del Ejercicio 3. La disposición de los procesos en el cronograma deberá incluir el proceso PA en la parte más alta del cronograma y, debajo, el resto de procesos en orden alfabético.**
- **Dentro de los archivos ej1.c y ej2.c se inclurirán comentarios internos para indicar los distintos apartados y subapartados. A modo de ejemplo:**

```
//Apartado 2
//Apartado 2.1
```

UAX Universidad Alfonso X el Sabio

G R A C I A S

UAX.COM