Memoria Ejercicio feedback

Autor:

1. Índice de contenidos

2. Solución Ejercicio 1

En este ejercicio se ha desarrollado una solución en lenguaje C para gestionar un conjunto de números enteros positivos mediante una estructura de datos. El objetivo principal es aprender a agrupar información relacionada dentro de una estructura, así como implementar funciones que permitan operar sobre dicha estructura de forma segura y controlada.

La solución se basa en un array de tamaño fijo, cuya capacidad máxima es de 10 elementos. Sin embargo, no todas las posiciones del array contienen necesariamente información válida en todo momento. Por este motivo, se diferencia claramente entre la **capacidad del array** y el **número real de elementos útiles almacenados**, utilizando una variable específica para ello.

Además, se mantiene de forma actualizada la suma de los elementos almacenados, evitando tener que recalcularla cada vez que se necesite. Para facilitar la depuración y la visualización del estado interno del array, se utiliza el valor **-1** como marcador de posiciones vacías.

Todas las funciones implementadas incluyen comprobaciones básicas que permiten evitar errores comunes, como accesos fuera de rango, inserciones de valores no válidos o el uso de punteros nulos.

2.1. Solución Apartado 1

En este apartado se pide definir una estructura de datos que permita almacenar hasta 10 números enteros positivos. Dicha estructura debe incluir:

- un array de enteros de tamaño fijo,
- una variable que indique el número de elementos válidos almacenados,
- y una variable que contenga la suma de dichos elementos.

Además, se debe realizar un typedef de la estructura definida.

```c
#include <stdio.h>
#define SIZE 10

typedef struct {
    int arrInt[SIZE];   // Array de enteros
    int arrSize;        // Número de elementos válidos
    int arrAdd;         // Suma de los elementos
} arrayLength_t;
```

2.2. Solución Apartado 2

Función `initArray`

Esta función se encarga de inicializar la estructura completa. Todas las posiciones del array se establecen a **-1**, indicando que se encuentran vacías. Además, se inicializan a cero tanto el número de elementos válidos (`arrSize`) como la suma de dichos elementos (`arrAdd`).

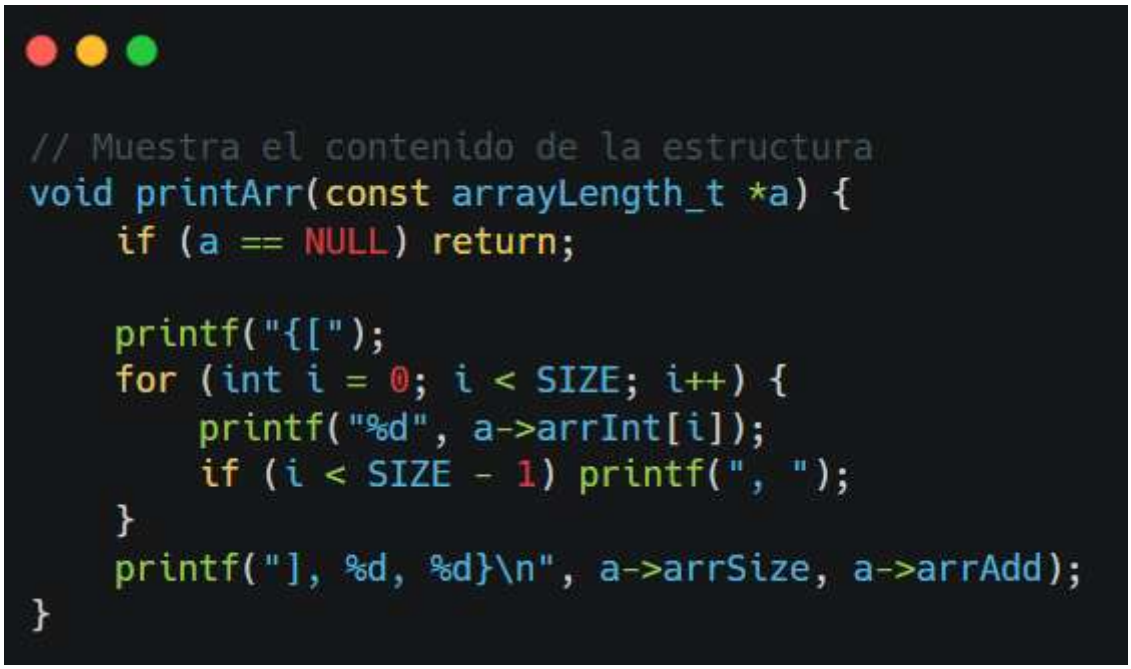
Aunque el array tiene capacidad para 10 elementos, tras la inicialización no hay ningún elemento útil almacenado, por lo que `arrSize` permanece en 0.

```
// ============================================
// Apartado 2
// Funciones initArray, printArr y addElement
// ============================================

// Inicializa la estructura
int initArray(arrayLength_t *a) {
    if (a == NULL) return -1;

    for (int i = 0; i < SIZE; i++) {
        a->arrInt[i] = -1;   // -1 indica posición vacía
    }

    a->arrSize = 0;
    a->arrAdd  = 0;

    return 0;
}
```

Función printArr

La función printArr permite mostrar por pantalla el contenido completo de la estructura. Se imprime el array completo, junto con el número de elementos válidos y la suma acumulada.

Mostrar todas las posiciones del array, incluidas las vacías, facilita la depuración y permite comprobar visualmente el estado interno de la estructura tras cada operación.



```c
// Muestra el contenido de la estructura
void printArr(const arrayLength_t *a) {
    if (a == NULL) return;

    printf("{[");
    for (int i = 0; i < SIZE; i++) {
        printf("%d", a->arrInt[i]);
        if (i < SIZE - 1) printf(", ");
    }
    printf("], %d, %d}\n", a->arrSize, a->arrAdd);
}
```

Función addElement

La función addElement añade un nuevo elemento al array, siempre que se cumplan las siguientes condiciones:

- el puntero a la estructura no sea nulo,
- el valor a insertar sea positivo,
- y exista espacio disponible en el array.

El nuevo valor se inserta en la primera posición libre del array, que coincide con el índice arrSize. A continuación, se incrementa el número de elementos válidos y se actualiza la suma total.

```c
// Añade un elemento al array
int addElement(arrayLength_t *a, int value) {
    if (a == NULL) return -1;
    if (value <= 0) return -1;
    if (a->arrSize >= SIZE) return -1;

    a->arrInt[a->arrSize] = value;
    a->arrSize++;
    a->arrAdd += value;

    return 0;
}
```

```
Array inicializado
{[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1], 0, 0}
Elemento añadido
{[22, -1, -1, -1, -1, -1, -1, -1, -1, -1], 1, 22}
Elemento añadido
{[22, 44, -1, -1, -1, -1, -1, -1, -1, -1], 2, 66}
```

La salida mostrada por el programa confirma el correcto funcionamiento de las funciones implementadas en el Ejercicio 1. En primer lugar, se observa que la estructura se inicializa correctamente, estableciendo todas las posiciones del array a -1y fijando tanto el número de elementos válidos como la suma total a cero.

Posteriormente, al añadir nuevos elementos mediante la función correspondiente, estos se insertan en la primera posición libre del array y se actualizan de forma coherente los valores de `arrSize` y `arrAdd`. La evolución de la salida por consola permite comprobar visualmente que cada operación produce el efecto esperado sobre la estructura.

En conjunto, los resultados obtenidos validan que la gestión del array se realiza de forma correcta, respetando las restricciones del ejercicio y manteniendo en todo momento la coherencia de los datos almacenados.

2.3. Solución Apartado 3

Función `getArrSize`

La función `getArrSize` devuelve el número de elementos válidos almacenados en el array. Este valor coincide con el campo `arrSize` de la estructura.

```
int getArrSize(const arrayLength_t *a) {
    if (a == NULL) return -1;
    return a->arrSize;
}
```

Esta función permite conocer cuántos elementos contiene realmente el array, independientemente de su capacidad máxima.

Función getArrAdd

La función getArrAdd devuelve la suma de los elementos válidos almacenados en el array. Este valor se obtiene directamente del campo arrAdd.

```
int getArrAdd(const arrayLength_t *a) {
    if (a == NULL) return -1;
    return a->arrAdd;
}
```

Gracias a esta función, no es necesario recorrer el array para calcular la suma, ya que esta se mantiene actualizada durante las operaciones de inserción y modificación.

Función getElement

La función getElement permite obtener el valor almacenado en una posición concreta del array. Para evitar errores, se comprueba que:

- el puntero a la estructura sea válido,

- el índice solicitado no sea negativo,
- y el índice sea menor que el número de elementos válidos (arrSize).

```c
int getElement(const arrayLength_t *a, int index) {
    if (a == NULL) return -1;
    if (index < 0 || index >= a->arrSize) return -1;

    return a->arrInt[index];
}
```

De esta forma, se garantiza que solo se accede a posiciones que contienen datos válidos, evitando accesos fuera de rango o a posiciones marcadas como vacías.

2.4. Solución Apartado 4

Función `setElement`

La función `setElement` recibe como parámetros un puntero a la estructura, un índice y el nuevo valor a almacenar. En caso de que alguno de los parámetros no sea válido, la función devuelve -1.

```c
int setElement(arrayLength_t *a, int index, int value) {
    if (a == NULL) return -1;
    if (value <= 0) return -1;
    if (index < 0 || index >= a->arrSize) return -1;

    a->arrAdd -= a->arrInt[index];
    a->arrInt[index] = value;
    a->arrAdd += value;

    return 0;
}
```

2.5. Solución Apartado 5

Función `resetArr`

La función `resetArr` recibe un puntero a la estructura y comprueba previamente que este sea válido. A continuación, recorre todas las posiciones del array, marcándolas como vacías, y reinicia los contadores.

```c
int resetArr(arrayLength_t *a) {
    if (a == NULL) return -1;

    for (int i = 0; i < 10; i++) {
        a->arrInt[i] = -1;
    }

    a->arrSize = 0;
    a->arrAdd  = 0;

    return 0;
}
```

2.6. Solución Apartado 6

Programa principal (main)

A continuación se muestra el código correspondiente a la función principal utilizada para verificar el funcionamiento del ejercicio completo:

```c
int main(void) {
    arrayLength_t al1, al2;

    // Inicialización de ambas estructuras
    initArray(&al1);
    initArray(&al2);

    // 1) al1 almacena los valores 0, 10, 20, ..., 90
    for (int i = 0; i < 10; i++) {
        addElement(&al1, i * 10);
    }

    // 2) Mostrar la estructura al1
    printArr(&al1);

    // 3) Actualizar posiciones impares de al1 con los valores 1, 3, 5, 7 y 9
    for (int i = 1; i < al1.arrSize; i += 2) {
        setElement(&al1, i, i);
    }

    // 4) Mostrar de nuevo la estructura al1
    printArr(&al1);

    // 5) Copiar las posiciones pares de al1 en al2
    for (int i = 0; i < al1.arrSize; i += 2) {
        addElement(&al2, getElement(&al1, i));
    }

    // 6) Actualizar las posiciones finales de al2 con valores de 0 a 4
    for (int i = 0; i < 5; i++) {
        setElement(&al2, al2.arrSize - 5 + i, i);
    }

    // 7) Mostrar la estructura al2
    printArr(&al2);

    return 0;
}
```

En primer lugar, se declaran dos estructuras de tipo arrayLength_t y se inicializan mediante la función initArray, lo que deja ambos arrays vacíos y preparados para su uso.

A continuación, se insertan en la estructura al1 los valores 0, 10, 20, …, 90. Debido a la validación implementada en la función addElement, el valor 0 no se almacena, por lo que finalmente se insertan únicamente los valores positivos válidos. Tras esta operación, se muestra por pantalla el contenido de al1.

Posteriormente, se modifican las posiciones impares del array al1, asignándoles los valores 1, 3, 5, 7 y 9 respectivamente. Esta operación se realiza mediante la función setElement, que actualiza correctamente tanto el contenido del array como la suma total de los elementos. De nuevo, se muestra el estado actualizado de al1.

En el siguiente paso, se copian las posiciones pares del array al1 en la estructura al2, utilizando las funciones getElement y addElement. Los valores copiados se almacenan de forma consecutiva en al2.

Por último, se modifican las posiciones finales de al2 para que almacenen los valores del 0 al 4 y se muestra por pantalla el estado final de la estructura al2, comprobando que todas las operaciones se han realizado correctamente.

```
{[10, 20, 30, 40, 50, 60, 70, 80, 90, -1], 9, 450}
{[10, 1, 30, 3, 50, 5, 70, 7, 90, -1], 9, 266}
{[10, 1, 2, 3, 4, -1, -1, -1, -1, -1], 5, 20}
```

3. Solución Ejercicio 2

En este ejercicio se trabaja con estructuras que representan potencias matemáticas y se comparan distintas formas de cálculo utilizando procesos y hebras. El objetivo principal es afianzar el manejo de estructuras, punteros y mecanismos de concurrencia en C.

3.1. Solución Apartado 1

Definición de constantes y tipos de datos

En primer lugar, se define una constante simbólica `SIZE` que establece el tamaño máximo de los arrays utilizados en el ejercicio. A continuación, se declara la estructura `potencia_t`, que agrupa la base, el exponente y el resultado de la potencia.

```c
#define SIZE 10

typedef struct {
    int base;
    int exp;
    int potencia;
} potencia_t;

typedef potencia_t* potenciaP_t;
```

La estructura permite almacenar de forma conjunta todos los datos necesarios para representar una potencia, mientras que el tipo potenciaP_t facilita el uso de punteros a dicha estructura.

Función setBaseExp

La función setBaseExp se encarga de inicializar los campos base y exp de la estructura, dejando el campo potencia con valor -1. Este valor indica que la potencia todavía no ha sido calculada.

```c
void setBaseExp(potenciaP_t p, int base, int exp) {
    if (p == NULL) return;

    p->base = base;
    p->exp = exp;
    p->potencia = -1;
}
```

Esta función resulta útil para preparar la estructura antes de realizar el cálculo de la potencia.

Función getPotencia

La función getPotencia calcula el valor de una potencia sin hacer uso de funciones predefinidas del lenguaje, como pow. El cálculo se realiza mediante multiplicaciones sucesivas.

```c
int getPotencia(int base, int exp) {
    int resultado = 1;

    for (int i = 0; i < exp; i++) {
        resultado *= base;
    }

    return resultado;
}
```

Este enfoque permite entender de forma clara cómo se obtiene el resultado de una potencia a nivel básico.

Función setPotenciaEst

La función setPotenciaEst utiliza los valores almacenados en la estructura para calcular la potencia y guardar el resultado en el campo correspondiente.

```c
void setPotenciaEst(potenciaP_t p) {
    if (p == NULL) return;

    p->potencia = getPotencia(p->base, p->exp);
}
```

Gracias a esta función, el cálculo de la potencia queda completamente encapsulado dentro de la estructura, lo que mejora la organización y reutilización del código.

3.2. Solución Apartado 2

Función initArrayEst

La función initArrayEst inicializa un array de estructuras potencia_t. Para cada posición del array:

- el campo base toma el valor del índice más uno,
- el campo exp se inicializa a cero,
- y el campo potencia se establece a uno.

Este comportamiento coincide con el estado inicial indicado en el enunciado del ejercicio.

```c
void initArrayEst(potencia_t arr[]) {
    for (int i = 0; i < SIZE; i++) {
        arr[i].base = i + 1;
        arr[i].exp = 0;
        arr[i].potencia = 1;
    }
}
```

Tras la ejecución de esta función, el array queda correctamente inicializado y preparado para ser utilizado en los cálculos posteriores.

Función printArrayEst

La función printArrayEst muestra por pantalla el contenido completo del array de estructuras. Para cada posición, se indica el índice del array y los valores de los campos base, exp y potencia.

```c
void printArrayEst(potencia_t arr[]) {
    for (int i = 0; i < SIZE; i++) {
        printf("arr[%d]: base: %d exp: %d potencia %d\n",
               i, arr[i].base, arr[i].exp, arr[i].potencia);
    }
}
```

Esta función resulta especialmente útil para comprobar visualmente el estado del array tras cada operación realizada sobre él.

Función calculoPotHeb

La función calculoPotHeb se implementa para ser utilizada como rutina de ejecución de una hebra. Recibe un puntero genérico void*, que se convierte internamente en un puntero a una estructura potencia_t.

A partir de los valores almacenados en los campos base y exp, la función calcula la potencia correspondiente mediante multiplicaciones sucesivas y almacena el resultado en el campo potencia.

```c
void* calculoPotHeb(void* arg) {
    potencia_t* p = (potencia_t*) arg;

    int resultado = 1;
    for (int i = 0; i < p->exp; i++) {
        resultado *= p->base;
    }

    p->potencia = resultado;
    return NULL;
}
```

El uso de un puntero genérico permite que esta función sea compatible con la creación de hebras mediante la librería pthread.

3.3. Solución Apartado 3

En primer lugar, se inicializa el array arr1 utilizando la función initArrayEst. A continuación, se modifican los exponentes de cada estructura, asignando como valor del exponente el índice correspondiente.

Para calcular las potencias, se crea un proceso hijo por cada posición del array mediante la función fork(). Cada proceso hijo calcula el valor de la potencia correspondiente y finaliza su ejecución devolviendo el resultado al proceso padre mediante la función exit().

El proceso padre espera la finalización de cada hijo utilizando wait() y recoge el resultado calculado, almacenándolo en el campo potencia de la estructura correspondiente.

Este enfoque permite realizar el cálculo de las potencias utilizando procesos independientes, cada uno con su propio espacio de memoria.

```
ARR1 inicial:
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[3]: base: 4 exp: 0 potencia 1
arr[4]: base: 5 exp: 0 potencia 1
arr[5]: base: 6 exp: 0 potencia 1
arr[6]: base: 7 exp: 0 potencia 1
arr[7]: base: 8 exp: 0 potencia 1
arr[8]: base: 9 exp: 0 potencia 1
arr[9]: base: 10 exp: 0 potencia 1

ARR1 con exponentes modificados:
arr[0]: base: 1 exp: 0 potencia 1
arr[1]: base: 2 exp: 1 potencia 1
arr[2]: base: 3 exp: 2 potencia 1
arr[3]: base: 4 exp: 3 potencia 1
arr[4]: base: 5 exp: 4 potencia 1
arr[5]: base: 6 exp: 5 potencia 1
arr[6]: base: 7 exp: 6 potencia 1
arr[7]: base: 8 exp: 7 potencia 1
arr[8]: base: 9 exp: 8 potencia 1
arr[9]: base: 10 exp: 9 potencia 1

ARR1 tras fork():
arr[0]: base: 1 exp: 0 potencia 1
arr[1]: base: 2 exp: 1 potencia 2
arr[2]: base: 3 exp: 2 potencia 9
arr[3]: base: 4 exp: 3 potencia 64
arr[4]: base: 5 exp: 4 potencia 113
arr[5]: base: 6 exp: 5 potencia 96
arr[6]: base: 7 exp: 6 potencia 145
arr[7]: base: 8 exp: 7 potencia 0
arr[8]: base: 9 exp: 8 potencia 65
arr[9]: base: 10 exp: 9 potencia 0
```

A continuación, se inicializa el array arr2 de forma análoga al caso anterior y se modifican los valores de los exponentes.

En este caso, el cálculo de las potencias se realiza mediante hebras. Para ello, se crea una hebra por cada posición del array utilizando la función pthread_create(). Cada hebra ejecuta la función calculoPotHeb, que calcula la potencia correspondiente y almacena el resultado directamente en la estructura asociada.

Una vez creadas todas las hebras, el programa principal espera su finalización mediante la función pthread_join(), garantizando que todos los cálculos han finalizado antes de mostrar los resultados por pantalla.

Este método resulta más eficiente que el uso de procesos, ya que las hebras comparten el mismo espacio de memoria y su creación tiene un menor coste.

```
ARR2 inicial:
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[3]: base: 4 exp: 0 potencia 1
arr[4]: base: 5 exp: 0 potencia 1
arr[5]: base: 6 exp: 0 potencia 1
arr[6]: base: 7 exp: 0 potencia 1
arr[7]: base: 8 exp: 0 potencia 1
arr[8]: base: 9 exp: 0 potencia 1
arr[9]: base: 10 exp: 0 potencia 1

ARR2 con exponentes modificados:
arr[0]: base: 1 exp: 0 potencia 1
arr[1]: base: 2 exp: 1 potencia 1
arr[2]: base: 3 exp: 2 potencia 1
arr[3]: base: 4 exp: 3 potencia 1
arr[4]: base: 5 exp: 4 potencia 1
arr[5]: base: 6 exp: 5 potencia 1
arr[6]: base: 7 exp: 6 potencia 1
arr[7]: base: 8 exp: 7 potencia 1
arr[8]: base: 9 exp: 8 potencia 1
arr[9]: base: 10 exp: 9 potencia 1

ARR2 tras pthread:
arr[0]: base: 1 exp: 0 potencia 1
arr[1]: base: 2 exp: 1 potencia 2
arr[2]: base: 3 exp: 2 potencia 9
arr[3]: base: 4 exp: 3 potencia 64
arr[4]: base: 5 exp: 4 potencia 625
arr[5]: base: 6 exp: 5 potencia 7776
arr[6]: base: 7 exp: 6 potencia 117649
arr[7]: base: 8 exp: 7 potencia 2097152
arr[8]: base: 9 exp: 8 potencia 43046721
arr[9]: base: 10 exp: 9 potencia 1000000000
```

Comparación entre procesos y hebras

A partir de los resultados obtenidos, se puede observar que ambos mecanismos permiten obtener correctamente el valor de las potencias. Sin embargo, el uso de hebras presenta

ventajas en términos de eficiencia y simplicidad, ya que no es necesario gestionar la comunicación entre procesos ni la recogida de valores de salida.

Por el contrario, el uso de procesos implica un mayor coste en la creación y finalización de cada proceso, además de requerir mecanismos adicionales para recuperar los resultados calculados.

4. Solución Ejercicio 3

En este apartado se analiza el funcionamiento de un planificador basado en colas multinivel con prioridades dinámicas y algoritmo Round Robin. El sistema dispone de tres colas de prioridad, siendo la prioridad 1 la más alta y la prioridad 3 la más baja. Cada cola utiliza un cuanto de tiempo diferente: 2 unidades de tiempo para la cola 1, 3 unidades de tiempo para la cola 2 y 4 unidades de tiempo para la cola 3.

El sistema es expulsivo, de manera que cuando llega un proceso con mayor prioridad que el que se está ejecutando, este último es expulsado del procesador. En dicho caso, el proceso expulsado vuelve al principio de su cola con el cuanto restante. Cuando un proceso se bloquea, al desbloquearse vuelve siempre a la cola desde la que partió, se inserta al final de la cola y recupera el cuanto completo. En general, cuando varios procesos entran en una cola, se da prioridad al proceso que más tiempo lleve sin estar en posesión del procesador.

Los procesos considerados en el sistema son los siguientes:

Proceso PA: tiempo de llegada 0, tiempo de servicio 6, prioridad 1.
Proceso PB: tiempo de llegada 1, tiempo de servicio 4, prioridad 2.
Proceso PC: tiempo de llegada 2, tiempo de servicio 3, prioridad 2.
Proceso PD: tiempo de llegada 4, tiempo de servicio 6, prioridad 1.
Proceso PE: tiempo de llegada 8, tiempo de servicio 2, prioridad 1.

El proceso PD se bloquea tras ejecutar una unidad de tiempo y permanece bloqueado durante dos unidades de tiempo.

4.1. Solución Apartado 1

En este apartado se resuelve el problema de planificación utilizando un sistema basado en colas de prioridad con planificación expulsiva (*preemptive scheduling*).
Se emplean tres colas de prioridad, siendo la cola 1 la de mayor prioridad y la cola 3 la de menor. Cada cola se gestiona mediante un algoritmo Round-Robin con cuantum diferente:

- Prioridad 1: q = 2 u.t.
- Prioridad 2: q = 3 u.t.
- Prioridad 3: q = 4 u.t.

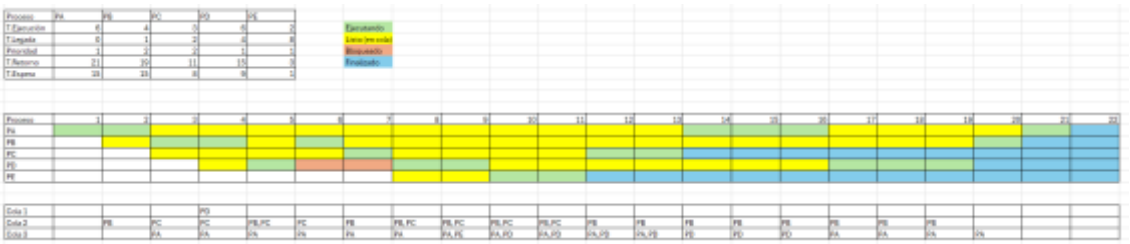
Cuando un proceso agota su cuantum, desciende a la cola de prioridad inmediatamente inferior. Si un proceso es expulsado por otro de mayor prioridad, vuelve al principio de su cola con el

cuantum restante. En caso de bloqueo, el proceso vuelve a su cola original al final y con cuantum completo.

Proceso	T. Retorno	T. Espera
PA	21	15
PB	19	15
PC	11	8
PD	15	9
PE	3	1

En la Figura 1 se muestra el cronograma de ejecución completo correspondiente a este apartado. En dicho cronograma se representa el estado de cada proceso en cada unidad de tiempo, así como el contenido de las colas de prioridad del sistema.

La disposición de los procesos en el cronograma se ha realizado colocando el proceso PA en la parte superior y, a continuación, el resto de procesos (PB, PC, PD y PE) en orden alfabético, tal y como se indica en las instrucciones del ejercicio.



- ***Figura 1: Cronograma de ejecución con planificación Round Robin.***

A partir del cronograma obtenido se puede comprobar que el comportamiento del planificador cumple con las normas establecidas en el enunciado. Los procesos de mayor prioridad acceden antes al procesador, mientras que el uso de Round Robin garantiza un reparto equitativo del tiempo de CPU dentro de cada cola. Los bloqueos y expulsiones se gestionan correctamente, respetando tanto la prioridad como el cuanto restante.

4.2. Solución Apartado 2

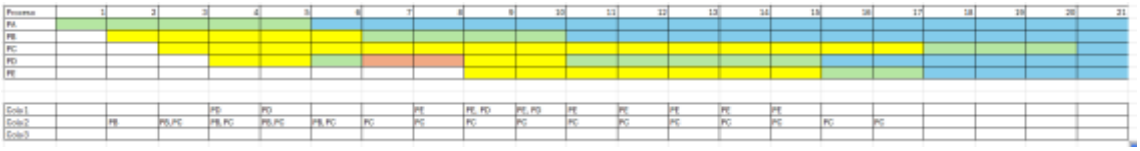
En este apartado se resuelve el mismo escenario de planificación descrito en el apartado anterior, manteniendo el sistema de colas de prioridad, pero sustituyendo el algoritmo Round-Robin por la política FIFO (First Come First Served) dentro de cada cola.

El sistema sigue siendo expulsivo, de forma que un proceso puede ser interrumpido únicamente por la llegada de otro proceso de mayor prioridad. Sin embargo, al emplear FIFO, los procesos no rotan ni se les asigna un cuantum máximo de ejecución, sino que mantienen el procesador hasta finalizar su ejecución o quedar bloqueados.

En este caso, los procesos se insertan en las colas respetando el orden de llegada, priorizando siempre aquellos que llevan más tiempo esperando sin haber ejecutado. Cuando un proceso se bloquea, como ocurre con el proceso PD tras una unidad de tiempo de ejecución, este se elimina temporalmente del sistema y, una vez desbloqueado, vuelve a la cola correspondiente a su prioridad original, incorporándose al final de dicha cola.

En la siguiente figura se muestra el cronograma de ejecución obtenido para este escenario, donde se representan los estados de cada proceso en cada instante de tiempo, así como el contenido de las colas de prioridad en cada momento.

En la Figura 2 se presenta el cronograma de ejecución obtenido al aplicar la política FIFO dentro de cada cola de prioridad. En este cronograma se observa la evolución temporal de los estados de los procesos y el contenido de las colas en cada instante.



- ***Figura 2: Cronograma de ejecución con planificación FIFO.***

A partir de los resultados obtenidos se puede observar que, en este ejemplo concreto, la planificación FIFO permite una finalización ligeramente más temprana de los procesos respecto al uso de Round Robin. No obstante, FIFO puede provocar mayores tiempos de espera para procesos que llegan más tarde, mientras que Round Robin ofrece un reparto más equitativo del procesador.

Ambos métodos presentan ventajas e inconvenientes, siendo su idoneidad dependiente del tipo de carga y de los objetivos del sistema operativo.

5. Anexo1: Código de la solución.

Código fuente ej1.c

```
// ========================================
// ej1.c
// Ejercicio 1 – Sistemas Operativos
```

```
// ========================================
#include <stdio.h>
#define SIZE 10
// ========================================
// Apartado 1
// Definición de la estructura y typedef
// ========================================
typedef struct {
   int arrInt[SIZE];   // Array de enteros
   int arrSize;       // Número de elementos válidos
   int arrAdd;        // Suma de los elementos
} arrayLength_t;
// ========================================
// Apartado 2
// Funciones initArray, printArr y addElement
// ========================================
// Inicializa la estructura
int initArray(arrayLength_t *a) {
   if (a == NULL) return -1;
   for (int i = 0; i < SIZE; i++) {
      a->arrInt[i] = -1;   // -1 indica posición vacía
   }
   a->arrSize = 0;
   a->arrAdd  = 0;
   return 0;
}
// Muestra el contenido de la estructura
void printArr(const arrayLength_t *a) {
```

```
    if (a == NULL) return;
    printf("{[");
    for (int i = 0; i < SIZE; i++) {
        printf("%d", a->arrInt[i]);
        if (i < SIZE - 1) printf(", ");
    }
    printf("], %d, %d}\n", a->arrSize, a->arrAdd);
}
// Añade un elemento al array
int addElement(arrayLength_t *a, int value) {
    if (a == NULL) return -1;
    if (value <= 0) return -1;
    if (a->arrSize >= SIZE) return -1;
    a->arrInt[a->arrSize] = value;
    a->arrSize++;
    a->arrAdd += value;
    return 0;
}
// ========================================
// Apartado 3
// Funciones de consulta (getters)
// ========================================
int getArrSize(const arrayLength_t *a) {
    if (a == NULL) return -1;
    return a->arrSize;
}
int getArrAdd(const arrayLength_t *a) {
    if (a == NULL) return -1;
```

```
    return a->arrAdd;
}
int getElement(const arrayLength_t *a, int index) {
    if (a == NULL) return -1;
    if (index < 0 || index >= a->arrSize) return -1;
    return a->arrInt[index];
}
// ========================================
// Apartado 4
// Modificación de un elemento del array
// ========================================
int setElement(arrayLength_t *a, int index, int value) {
    if (a == NULL) return -1;
    if (value <= 0) return -1;
    if (index < 0 || index >= a->arrSize) return -1;
    a->arrAdd -= a->arrInt[index];
    a->arrInt[index] = value;
    a->arrAdd += value;
    return 0;
}
// ========================================
// Apartado 5
// Reinicio de la estructura
// ========================================
int resetArr(arrayLength_t *a) {
    if (a == NULL) return -1;
    for (int i = 0; i < SIZE; i++) {
        a->arrInt[i] = -1;
```

```
    }
    a->arrSize = 0;
    a->arrAdd  = 0;
    return 0;
}
// ========================================
// Apartado 6
// Programa principal de prueba
// ========================================
int main(void) {
    arrayLength_t al1, al2;
    // Inicialización de estructuras
    initArray(&al1);
    initArray(&al2);
    // 1) al1 almacena los valores 0, 10, 20, ..., 90
    for (int i = 0; i < SIZE; i++) {
        addElement(&al1, i * 10);
    }
    // 2) Mostrar al1
    printArr(&al1);
    // 3) Actualizar posiciones impares con 1, 3, 5, 7, 9
    for (int i = 1; i < al1.arrSize; i += 2) {
        setElement(&al1, i, i);
    }
    // 4) Mostrar de nuevo al1
    printArr(&al1);
    // 5) Copiar posiciones pares de al1 en al2
    for (int i = 0; i < al1.arrSize; i += 2) {
```

```
        addElement(&al2, getElement(&al1, i));
    }
    // 6) Actualizar las posiciones finales de al2 con valores de 0 a 4
    for (int i = 0; i < 5; i++) {
        setElement(&al2, al2.arrSize - 5 + i, i);
    }
    // 7) Mostrar al2
    printArr(&al2);
    return 0;
}
```

Código fuente ej2.c

```
// ========================================
// Ejercicio 2
// Sistemas Operativos
// ========================================
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#define SIZE 10
// ========================================
// Apartado 3.1
// ========================================
// Estructura potencia_t
typedef struct {
```

```
    int base;
    int exp;
    int potencia;
} potencia_t;
// Tipo puntero a potencia_t
typedef potencia_t* potenciaP_t;
// Inicializa base y exp, y pone potencia a -1
void setBaseExp(potenciaP_t p, int base, int exp) {
    if (p == NULL) return;
    p->base = base;
    p->exp = exp;
    p->potencia = -1;
}
// Calcula base^exp SIN usar pow
int getPotencia(int base, int exp) {
    int res = 1;
    for (int i = 0; i < exp; i++) {
        res *= base;
    }
    return res;
}
// Guarda en la estructura el valor base^exp
void setPotenciaEst(potenciaP_t p) {
    if (p == NULL) return;
    p->potencia = getPotencia(p->base, p->exp);
}
// ========================================
// Apartado 3.2
```

```
// ========================================
// Inicializa el array de estructuras
// base = indice + 1
// exp = 0
// potencia = 1
void initArrayEst(potencia_t arr[]) {
   for (int i = 0; i < SIZE; i++) {
      arr[i].base = i + 1;
      arr[i].exp = 0;
      arr[i].potencia = 1;
   }
}
// Muestra el contenido del array
void printArrayEst(potencia_t arr[]) {
   for (int i = 0; i < SIZE; i++) {
      printf("arr[%d]: base: %d exp: %d potencia %d\n",
            i, arr[i].base, arr[i].exp, arr[i].potencia);
   }
}
// Función para hebra (thread)
void* calculoPotHeb(void* arg) {
   potencia_t* p = (potencia_t*) arg;
   int res = 1;
   for (int i = 0; i < p->exp; i++) {
      res *= p->base;
   }
   p->potencia = res;
   return NULL;
```

```
}
// ========================================
// Apartado 3 – main() completo
// ========================================
int main(void) {
    potencia_t arr1[SIZE];
    potencia_t arr2[SIZE];
    pthread_t hilos[SIZE];
    pid_t pid;
    int status;
    // ---------- ARR1: procesos (fork) ----------
    initArrayEst(arr1);
    printf("\nARR1 inicial:\n");
    printArrayEst(arr1);
    // Modificar exponentes
    for (int i = 0; i < SIZE; i++) {
        arr1[i].exp = i;
    }
    printf("\nARR1 con exponentes modificados:\n");
    printArrayEst(arr1);
    // Cálculo con procesos
    for (int i = 0; i < SIZE; i++) {
        pid = fork();
        if (pid == 0) {  // Proceso hijo
            int res = getPotencia(arr1[i].base, arr1[i].exp);
            exit(res);
        }
        else {
```

```
            wait(&status);
            arr1[i].potencia = WEXITSTATUS(status);
        }
    }
    printf("\nARR1 tra fork():\n");
    printArrayEst(arr1);
    // ---------- ARR2: hebras (pthread) ----------
    initArrayEst(arr2);
    printf("\nARR2 inicial:\n");
    printArrayEst(arr2);
    // Modificar exponentes
    for (int i = 0; i < SIZE; i++) {
        arr2[i].exp = i;
    }
    printf("\nARR2 con exponentes modificados:\n");
    printArrayEst(arr2);
    // Cálculo con hebras
    for (int i = 0; i < SIZE; i++) {
        pthread_create(&hilos[i], NULL, calculoPotHeb, &arr2[i]);
    }
    for (int i = 0; i < SIZE; i++) {
        pthread_join(hilos[i], NULL);
    }
    printf("\nARR2 tras pthread:\n");
    printArrayEst(arr2);
    return 0;
}
```

UAX Universidad Alfonso X el Sabio

G R A C I A S

UAX.COM