

Memoria Ejercicio feedback

Autor: Martina García González NP:141478

1. Índice de contenidos

1.	ÍNDICE DE CONTENIDOS	2
2.	SOLUCIÓN EJERCICIO 1	3
2.1.	APARTADO 1	3
2.2.	APARTADO 2	3
2.3.	APARTADO 3	5
2.4.	APARTADO 4	6
2.5.	APARTADO 5	7
2.6.	APARTADO 6	8
3.	SOLUCIÓN EJERCICIO 2	9
3.1.	APARTADO 1	9
3.2.	APARTADO 2	10
3.3.	APARTADO 3	12
4.	SOLUCIÓN EJERCICIO 3	14
4.1.	SOLUCIÓN APARTADO 1	14
4.2.	SOLUCIÓN APARTADO 2	15
5.	ANEXO 1: CÓDIGO DE LA SOLUCIÓN	17
•	<i>Github: https://github.com/martinagg7/so_feedback.git</i>	17
•	<i>Cronogramas y tablas : ejercicio_3.xlsx.....</i>	17

2. Solución Ejercicio 1

2.1. Apartado 1

ArrayLength_t

En este ejercicio definimos la estructura **arrayLength_t**. El objetivo de esta es facilitar la organización de los datos y evitar operaciones innecesarias, como recorrer repetidamente el array para calcular cuántos elementos contiene o cuál es su suma total. Para ello, utilizamos el valor -1, que nos indica qué posiciones están vacías, evitando realizar búsquedas costosas en cada operación.

- **arrInt[10]: array de diez números enteros.** Cada posición puede almacenar un valor positivo o el valor -1, que se utilizará para indicar que dicha posición está vacía.
- **arrSize:** número de **elementos almacenados en el array**, es decir, cuántas posiciones contienen un valor distinto de -1.
- **arrAdd:** suma total de los valores almacenados en el array (solo los distintos de -1).

Código:

```
typedef struct {
    int arrInt[10];
    int arrSize;
    int arrAdd;
} arrayLength_t;
```

2.2. Apartado 2

Init Array():

La función **initArray** accede a la estructura definida previamente y la inicializa de la siguiente forma:

- Asigna el valor **-1** a todas las posiciones del array arrInt, indicando que no hay ningún elemento almacenado.
- Inicializa el campo **arrSize a 0**, ya que no existen elementos válidos en el array.
- Inicializa el campo **arrAdd a 0**, al no haber valores que sumar

Código:

```
int initArray(arrayLength_t *p) {

    // Error:-1
    if (p == NULL) {
        return -1;
    }
    // Bucle inicializar todas las posiciones del array a -1
    for (int i = 0; i < 10; i++) {
        p->arrInt[i] = -1;
    }
    // arrSize=arrAdd=0
    p->arrSize = 0;
    p->arrAdd = 0;

    return 0;
}
```

PrintArr():

La función **printArr** imprime el contenido de la estructura en el siguiente formato:

{[v0, v1, ..., v9], arrSize, arrAdd},

donde se muestran los valores almacenados en el array, junto con el número de elementos válidos y la suma de los mismos.

Código:

```
void printArr(const arrayLength_t *p) {

    printf("[");
    for (int i = 0; i < 10; i++) {
        printf("%d", p->arrInt[i]); //Elementos del array
        if (i < 9) {
            printf(",");
        }
    }
    printf("]", p->arrSize, p->arrAdd); //arrSize y
    arrAdd
}
```

AddElement():

La función **addElement** se encarga de añadir un nuevo valor al array de la estructura y realiza las siguientes acciones:

- Comprueba en primer lugar que el array no esté completo, ya que en provocaría un acceso fuera de rango (out of bounds)
- Utiliza el campo **arrSize**, que indica el número de elementos almacenados, para saber si existe espacio disponible y determinar directamente la posición en la

que debe insertarse el nuevo valor. De esta forma se evita recorrer todo el array, que tendría una complejidad $O(n)$

- Finalmente, se actualizan los campos **arrSize** y **arrAdd** para reflejar la inserción realizada.

Código:

```
int addElement(arrayLength_t *p, int value) {
    // Array lleno
    if (p->arrSize >= 10) {
        return -1;
    }

    // Valor negativo
    if (value < 0) {
        return -1;
    }

    // Añadir valor en la primera posición libre
    p->arrInt[p->arrSize] = value;

    // Actualizar arrSize y arrAdd
    p->arrSize++;
    p->arrAdd += value;

    return 0;
}
```

2.3. Apartado 3

getArrSize() y getArrAdd():

Las funciones **getArrSize** y **getArrAdd** permiten acceder de forma segura a la información almacenada en la estructura:

- Ambas funciones reciben un puntero a una estructura **arrayLength_t** y comprueban previamente que sea válido.
- **getArrSize** devuelve el número de elementos válidos almacenados en el array.
- **getArrAdd** devuelve la suma de estos

Código:

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

    return p->arrSize;
}

int getArrAdd(const arrayLength_t *p) {
    if (p == NULL) {
        return -1;
    }

    return p->arrAdd;
}
```

getElement():

La función **getElement** permite obtener un elemento concreto del array:

- Comprueba que el puntero y la posición sean válidos y se encuentren dentro de los límites del array.
- Verifica que la posición corresponde a un elemento almacenado y que el valor sea positivo.

Finalmente devuelve el valor solicitado o -1 en caso de error

Código:

```
int getElement(const arrayLength_t *p, int pos) {  
  
    if (p == NULL) {  
        return -1;  
    }  
    if (pos < 0 || pos > 9) {  
        return -1;  
    }  
  
    // position < arrSize  
    if (pos >= p->arrSize) {  
        return -1;  
    }  
  
    // valor almacenado debe ser positivo  
    if (p->arrInt[pos] < 0) {  
        return -1;  
    }  
  
    return p->arrInt[pos];  
}
```

2.4. Apartado 4

setElement():

La función **setElement** permite modificar el valor de una posición concreta del array:

- Comprueba que la posición indicada se encuentre dentro de los límites del array y que esté ocupada por un elemento válido.
- Verifica que el nuevo valor a asignar sea positivo.
- Sustituye el valor almacenado en la posición indicada y actualiza el campo **arrAdd** para mantener la suma correcta.

Finalmente devuelve el valor solicitado o -1 en caso de error

Código:

```
int setElement(arrayLength_t *p, int position, int value) {
    // 1. Comprobar límites
    if (position < 0 || position >= 10) {
        return -1;
    }

    // 2. Posición debe estar ocupada
    if (position >= p->arrSize) {
        return -1;
    }

    // 3. Nuevo valor positivo
    if (value < 0) {
        return -1;
    }

    // 4. Actualizar arrAdd
    p->arrAdd = (p->arrAdd - p->arrInt[position]) + value;

    // 5. Sustituir valor
    p->arrInt[position] = value;

    return 0;
}
```

2.5. Apartado 5

resetElement():

La función **resetArr** restablece la estructura a su **estado inicial**:

- Reutiliza la función **initArray** para asignar el valor **-1** a todas las posiciones del array.
- Restablece los campos **arrSize** y **arrAdd** a **0**.

Código:

```
int resetArr(arrayLength_t *p) {
    return initArray(p);
}
```

2.6. Apartado 6

En este apartado se define el main que se utiliza para comprobar si las anteriores funciones están bien declaradas:

- Crea dos estructuras de tipo arrayLength_t y las inicializa.
- Rellena la estructura al1 con múltiplos de 10 y muestra su contenido.
- Actualiza las posiciones impares de al1 con valores consecutivos impares y vuelve esta
- Copia los elementos de las posiciones pares de al1 en la estructura al2, almacenándolos de forma consecutiva.
- Finalmente los valores de 0 a 4 en al2 y muestra su contenido final.

El resultado de la ejecución confirma que las funciones implementadas funcionan correctamente :

```
● (base) martinagarciagonzalez@MacBook-Air-de-Martina-2 so_feedback % ./ej1
{[0, 10, 20, 30, 40, 50, 60, 70, 80, 90], 10, 450}
{[0, 1, 20, 3, 40, 5, 60, 7, 80, 9], 10, 225}
{[0, 20, 40, 60, 80, 0, 1, 2, 3, 4], 10, 210}
```

3. Solución Ejercicio 2

3.1. Apartado 1

potencia_t y potenciaP_t

En este apartado se definen los elementos básicos del ejercicio:

- **potencia_t** : una **estructura** que contiene los campos necesarios para representar una potencia: **base, exponente y resultado**.
- **potenciaP_t** : un **puntero que apunta a la dirección de memoria** de una **estructura potencia_t**, permitiendo acceder y modificar sus valores mediante funciones.

Código:

```
#define SIZE 10

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

typedef potencia_t* potenciaP_t;
```

setBaseExp():

Esta función asigna a la estructura los **valores de la base** y el **exponente de la potencia**, dejando el campo potencia a -1 para indicar que el resultado aún no ha sido calculado.

Código:

```
void setBaseExp(potencia_t *p, int base, int exp) {
    p->base = base;
    p->exp = exp;
    p->potencia = -1;
}
```

getPotencia() y setPotenciaEst():

La función **getPotencia** calcula el valor de una potencia a partir de una base y un exponente:

- Comprueba si el exponente es negativo y devuelve **-1** en ese caso.
- Si el exponente es cero, devuelve **1**.
- En el caso general, calcula la potencia mediante un bucle multiplicando la base tantas veces como indique el exponente.

La función **setPotenciaEst** calcula el valor de la potencia asociada a una estructura **potencia_t** y almacena el resultado en su campo **potencia**,

Código:

```
int getPotencia(int base, int exp) {
    // 1. Exponente negativo
    if (exp < 0) {
        return -1;
    }
    // 2. Exponente cero
    if (exp == 0) {
        return 1;
    }
    // 3. Caso Normal
    int resultado = 1;
    for (int i = 1; i <= exp; i++) {
        resultado = resultado * base;
    }
    return resultado;
}

void setPotenciaEst(potencia_t *p) {
    p->potencia = getPotencia(p->base, p->exp);
}
```

3.2. Apartado 2

initArrayEst() y printArrayEst():

- **initArrayEst**: inicializa el array de estructuras **potencia_t** mediante un bucle que recorre todos sus elementos, asignando a cada uno una base igual al índice más uno, el exponente a cero y la potencia a uno.
- **printArrayEst**: recorre el array y muestra por pantalla el contenido de cada estructura, indicando su índice, base, exponente y potencia.

Código:

```

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;
    }
}

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);
    }
}

```

initArrayEst() y printArrayEst():

En este apartado se emplean **hebras** (threads), que **permiten ejecutar varias tareas de forma concurrente dentro de un mismo proceso**, compartiendo memoria pero disponiendo cada una de su propia pila de ejecución.

- **calcPotHeb:** función utilizada por las hebras, que recibe un puntero genérico, lo convierte a **potencia_t** y calcula la potencia correspondiente, almacenando el resultado en el campo **potencia** de la estructura

Código:

```

void *calcPotHeb(void *arg)
{
    // convertir un puntero cualquiera a un puntero de tipo
    potencia_t
    potencia_t *p = (potencia_t *) arg;

    // calcular la potencia y actualizar el valor en la estructura
    p->potencia = getPotencia(p->base, p->exp);

    return NULL;
}

```

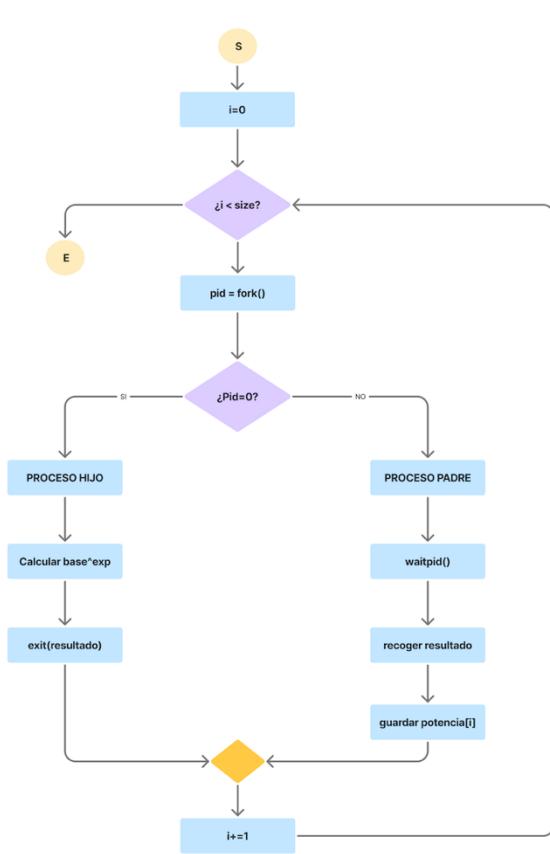
3.3. Apartado 3

Función Main:

1. Se declaran e inicializan los arrays de estructuras potencia_t, preparando los valores necesarios para el cálculo de las potencias.
2. Se crean procesos hijos mediante fork, donde cada proceso hijo calcula la potencia correspondiente y devuelve el resultado al proceso padre.
3. El proceso padre espera la finalización de cada hijo, recoge los valores calculados y los almacena en las estructuras correspondientes.
4. Finalmente, se repite el cálculo utilizando hebras sobre un segundo array y se muestran los resultados obtenidos.

Diagrama de flujo:

A través del diagrama de flujo se ha querido reflejar cómo, mediante la llamada al sistema fork, se crean procesos concurrentes. Esta llamada devuelve el valor 0 al proceso hijo y el PID del hijo al proceso padre, lo que permite distinguir su ejecución y



hacer que el proceso hijo calcule la potencia mientras el proceso padre espera y recoge el resultado obtenido.

Resultado:

A continuación, se presenta el resultado de ejecución del apartado anterior.

```
Arr1 inicializado:
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 tras actualizar base y exp:
arr[0]: base: 0 exp: 2 potencia -1
arr[1]: base: 1 exp: 2 potencia -1
arr[2]: base: 2 exp: 2 potencia -1
arr[3]: base: 3 exp: 2 potencia -1
arr[4]: base: 4 exp: 2 potencia -1
arr[5]: base: 5 exp: 2 potencia -1
arr[6]: base: 6 exp: 2 potencia -1
arr[7]: base: 7 exp: 2 potencia -1
arr[8]: base: 8 exp: 2 potencia -1
arr[9]: base: 9 exp: 2 potencia -1

Arr1 tras calcular las potencias:
arr[0]: base: 0 exp: 2 potencia 0
arr[1]: base: 1 exp: 2 potencia 1
arr[2]: base: 2 exp: 2 potencia 4
arr[3]: base: 3 exp: 2 potencia 9
arr[4]: base: 4 exp: 2 potencia 16
arr[5]: base: 5 exp: 2 potencia 25
arr[6]: base: 6 exp: 2 potencia 36
arr[7]: base: 7 exp: 2 potencia 49
arr[8]: base: 8 exp: 2 potencia 64
arr[9]: base: 9 exp: 2 potencia 81
```

```
Arr2 inicializado:
Arr2 inicializado:
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 tras actualizar base y exp:
arr[0]: base: 0 exp: 2 potencia -1
arr[1]: base: 1 exp: 2 potencia -1
arr[2]: base: 2 exp: 2 potencia -1
arr[3]: base: 3 exp: 2 potencia -1
arr[4]: base: 4 exp: 2 potencia -1
arr[5]: base: 5 exp: 2 potencia -1
arr[6]: base: 6 exp: 2 potencia -1
arr[7]: base: 7 exp: 2 potencia -1
arr[8]: base: 8 exp: 2 potencia -1
arr[9]: base: 9 exp: 2 potencia -1

Arr2 tras calcular las potencias:
arr[0]: base: 0 exp: 2 potencia 0
arr[1]: base: 1 exp: 2 potencia 1
arr[2]: base: 2 exp: 2 potencia 4
arr[3]: base: 3 exp: 2 potencia 9
arr[4]: base: 4 exp: 2 potencia 16
arr[5]: base: 5 exp: 2 potencia 25
arr[6]: base: 6 exp: 2 potencia 36
arr[7]: base: 7 exp: 2 potencia 49
arr[8]: base: 8 exp: 2 potencia 64
arr[9]: base: 9 exp: 2 potencia 81
```

4. Solución Ejercicio 3

Antes de analizar los resultados, se describe el formato utilizado para representar la evolución del sistema. Para cada proceso se ha representado su estado a lo largo del tiempo mediante un código de colores: **rojo** indica que el proceso se encuentra en **ejecución**, **amarillo** representa el estado de **bloqueo** debido a una operación de entrada/salida, **azul** corresponde al estado **listo**, en el que el proceso espera a ser asignado al procesador, y **verde** señala el **instante de finalización** del proceso.

Para cada uno de los apartados se ha construido un **cronograma de ejecución**, en el que se indica qué proceso está siendo ejecutado en cada instante de tiempo. Además, en la parte superior del cronograma se muestran las **tres colas de prioridad** del sistema, lo que permite visualizar con mayor detalle cómo evolucionan los procesos dentro de cada cola y cómo influyen las políticas de planificación, las expulsiones y los cambios dinámicos de prioridad a lo largo del tiempo. Con el objetivo de facilitar la comprensión del comportamiento del planificador en instantes concretos, se han añadido **comentarios explicativos** en determinadas celdas del cronograma.

Finalmente, para cada proceso se han calculado el **tiempo de retorno**, el **tiempo de ejecución** y el **tiempo de espera**, y a partir de ellos se ha obtenido el **tiempo medio de retorno**, que proporciona una medida global del rendimiento del sistema y permite comparar el comportamiento del planificador en los distintos escenarios analizados.

4.1. Solución Apartado 1

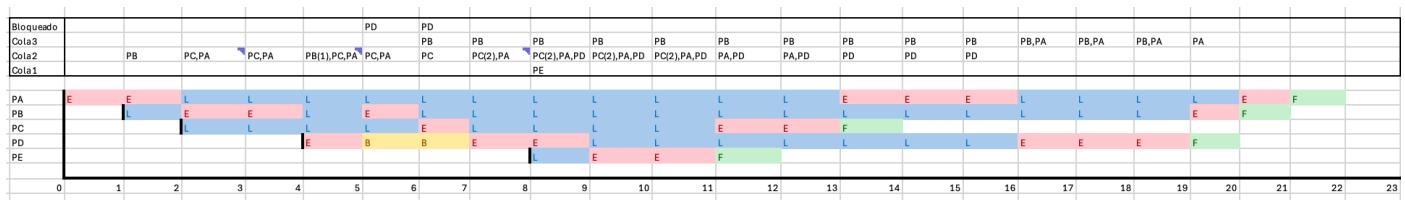
Cronograma

A continuación se presenta el **cronograma de ejecución correspondiente a este apartado**, en el que los procesos son ordenados en **colas de prioridad** y gestionados mediante el algoritmo **Round Robin**, aplicando una planificación **expulsiva**. En cada instante de tiempo se ejecuta el proceso perteneciente a la cola de mayor prioridad no vacía, respetando el cuanto asignado a cada nivel.

Los únicos instantes que se consideran relevantes de destacar son los siguientes:

- $t = 2$: el proceso **PA** agota su cuanto en la cola de prioridad 1 y pasa a la **cola 2**. En ese mismo instante llega el proceso **PC** a dicha cola, situándose por delante de PA al ser el proceso que llevaba más tiempo sin ejecutar el procesador.
- $t = 4$: el proceso **PD**, de prioridad 1, interrumpe la ejecución de **PB**. Al tratarse de una expulsión, PB vuelve al **frente de la cola 2**, conservando un cuanto restante de **1 unidad de tiempo**, ya que había consumido previamente dos unidades.

- $t = 7$: el proceso **PC** es expulsado por el proceso **PD**, de mayor prioridad, y se coloca nuevamente al **frente de la cola 2**, manteniendo un cuanto restante de $q = 2$ unidades de tiempo.



Tabla

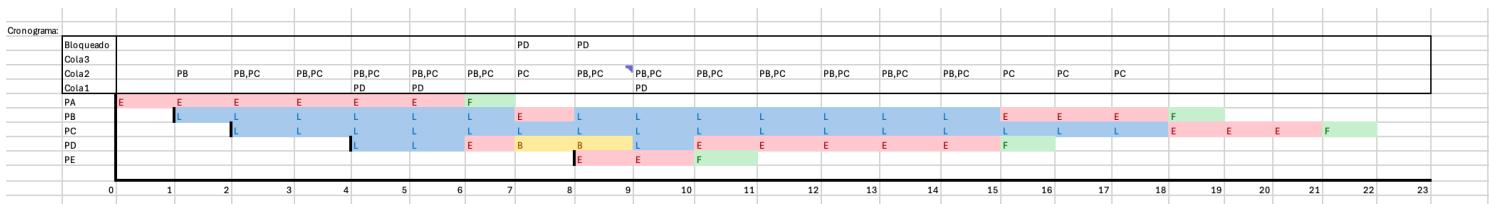
La tabla presentada a continuación, muestra los tiempos de retorno, espera y retorno normalizado de cada proceso, así como los valores medios del sistema. Se observa que **PA** y **PB** presentan los **mayores** tiempos de retorno y espera, lo que indica que son los procesos más penalizados por su mayor tiempo de servicio y por las interrupciones derivadas de la planificación expulsiva. Por el contrario, **PE** es el proceso más favorecido, con valores **bajos** de retorno y espera, al finalizar rápidamente gracias a su corta duración y alta prioridad. El **valor medio del tiempo de retorno normalizado**, moderadamente elevado, refleja el impacto global del planificador sobre el conjunto de procesos.

	TR	TE	TRN (TR/TS)
PA	21	15	3,5
PB	19	15	4,75
PC	11	8	3,666666667
PD	15	12	2,5
PE	3	1	1,5
MEDIA	13,8	10,2	3,183333333

4.2. Solución Apartado 2

En este apartado se presenta el cronograma correspondiente a un sistema de planificación por **colas de prioridad**, donde los procesos de mayor prioridad son claramente favorecidos. Tal y como se observa en el cronograma, una vez que un proceso de **prioridad más alta** entra en el procesador, **no puede ser expulsado por procesos de menor prioridad**, lo que provoca que estos últimos acumulen mayores tiempos de espera.

Además, al no existir un cuanto que limite estrictamente la ejecución de los procesos de mayor prioridad, estos pueden permanecer más tiempo en el procesador, retrasando la ejecución de los procesos de colas inferiores. Este comportamiento explica la concentración de ejecución en las colas de mayor prioridad y el aumento del tiempo de espera de los procesos menos prioritarios, tal y como se refleja en el cronograma.



Tabla

Al comparar las tablas de resultados, se observa que en el **apartado 2** los **tiempos medios de retorno y espera son menores** que en el apartado 1, lo que indica una **mejora global del rendimiento del sistema**. Esto se debe a que el planificador utilizado es de tipo **FIFO por colas de prioridad**, no expansivo, de modo que los procesos de mayor prioridad, una vez que acceden al procesador, lo utilizan sin interrupciones.

Este comportamiento beneficia especialmente a procesos como **PA** y **PE**, que presentan tiempos de retorno y espera **nulos o muy bajos**, al ejecutarse de forma continua desde su llegada. Sin embargo, esta mejora global se consigue a costa de penalizar a los procesos de menor prioridad, como **PC**, que presenta los **valores más altos** de tiempo de retorno y retorno normalizado, al quedar bloqueado durante largos períodos mientras se ejecutan procesos más prioritarios.

En consecuencia, aunque el apartado 2 ofrece **mejores valores medios**, el reparto del tiempo de procesador es **menos equitativo**, favoreciendo claramente a los procesos de mayor prioridad frente a los de menor prioridad.

	TR	TE	TRN (TR/TS)
PA	6	0	0
PB	17	13	4,25
PC	19	16	6,333333333
PD	11	5	1,833333333
PE	2	0	0
MEDIA	11	6,8	2,483333333

5. Anexo 1: Código de la solución

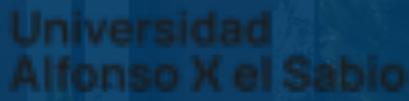
- Github: https://github.com/martinagg7/so_feedback.git
- Cronogramas y tablas : ejercicio_3.xlsx



**WELCOME
TO
UAX**



UAX



Universidad
Alfonso X el Sabio



GRACIAS



UAX.COM