

SISTEMAS OPERATIVOS

INFORME DE PRÁCTICA DE CONTROL

- › FCFS
- › SEGÚN NECESIDADES
- › MEMORIA NO CONTINUA - NO REUBICABLE

AUTOR: Iván Ruiz Gázquez

ÍNDICE

INTRODUCCIÓN	4
PASÁNDO DE LA TEORIA AL ALGORITMO.....	5
EJEMPLOS.....	6
EJEMPLO 1. FCFS SENCILLO. PROCESOS NO PARTICIONADOS.....	6
Tamaño de Memoria: 12.....	6
Procesos.....	6
T=0	6
T=6	7
T=9	7
T=10.....	8
T=11.....	8
T=17	8
T=18	8
T=21	9
T=31	9
T=32	10
T=36.....	11
EJEMPLO 1 EN SCRIPT. FCFS SENCILLO. PROCESOS NO PARTICIONADOS.....	12
EJEMPLO 2. FCFS COMPLEJO. PROCESOS MULTIPARTICIONADOS	18
Tamaño de Memoria: 23	18
Procesos.....	18
T=14	18
T=75.....	19
T=78.....	19
EJEMPLO 2 EN SCRIPT. FCFS COMPLEJO.	20
PROCESOS MULTIPARTICIONADOS.....	20
CAMBIOS Y MEJORAS	22
PÁGINA DE INICIO	22
CONTROL DE ERRORES.....	24
ENTRADA MANUAL	26
EJECUCIÓN DEL ALGORITMO	28
INFORME DE ALGORITMO	29
ENTRADA POR ARCHIVO	30

ENTRADA AUTOMÁTICA	30
CAMBIOS INTERNOS	30
AYUDA AL PROGRAMADOR	30
ARCHIVO CONFIG.....	31
DOCUMENTACIÓN GENERADA.....	32
GENERACIÓN DE DOCUMENTOS CON ZSDOC.....	32
ADOC DEL SCRIPT	33
INFORMACIÓN	33
FUNCTIONS.....	33
DETAILS.....	33
Script Body	33
asignarColores	34
asignarDatosInicial	34
asignarDesdeArchivo.....	34
asignarEstadosSegunInstante.....	34
asignarEstiloGeneral.....	35
asignarManual	35
asignarValoresAleatorios	35
avanzarAlgoritmo	35
calcularCambiosCPU	36
calcularCambiosMemoria.....	36
calcularLongitud	36
calcularMemoriaRestante	36
cc	36
centrarEnPantalla.....	36
colocarNombreAProcesos	37
comprobarProcesosEjecutando	37
copiarArray	37
elegirTipoDeEntrada.....	37
eliminarProcesosNoValidos	38
extraerDeConfig	38
fc.....	38
imprimirAyuda.....	38
imprimirCuadro	39
imprimirLineaProcesos	39
imprimirMemoria	39

imprimirTabla.....	39
main	40
ordenarArray.....	40
procesosHanTerminado.....	40
recibirEntrada.....	40
sacarHaciaArchivo.....	40
COMO GENERAR LA DOCUMENTACIÓN.....	41
HERRAMIENTAS	42
CONTROL DE VERSIONES:	42
IDE:	42
DEBUGGER:	42
SISTEMA OPERATIVO:.....	42
FORMATTER:.....	42

INTRODUCCIÓN

Cuando hablamos de algoritmos en sistemas operativos tenemos varias opciones:

- > Algoritmos de planificación por lotes:
 - FCFS
 - SRJ
 - SRTF
- > Algoritmos de planificación en sistemas interactivos
 - Round Robin
 - Por Prioridad
 - Múltiples Colas
 - Proceso Más Corto a Continuación
 - Planificación Garantizada
 - Planificación por Sorteo
 - Planificación por Partes Equitativas
- > Algoritmos de planificación en tiempo real

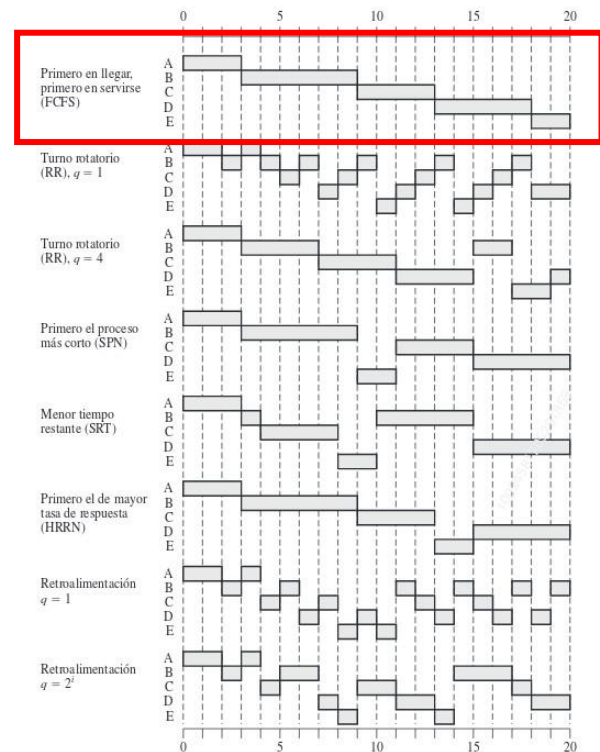


Figura 9.5. Comparación de políticas de planificación.

En nuestro caso, el que nos importa es el primero, FCFS. El algoritmo de FCFS (First Come, First Served), o FIFO (First Input, Last Output), es el más sencillo de todos. Los procesos entrarán a la CPU por orden de llegada, sin importar el tamaño de estos ni cualquier otra característica. De este modo se mantendrán en ejecución sin ser interrumpidos hasta que el tiempo restante de ejecución llegue a cero. En ese momento saldrán finalizados y el siguiente proceso en la lista entrará a ejecutarse. Haremos esto hasta llegar al final de la lista. Por esta razón, como veremos más adelante, los procesos de más atrás tardan mucho en ejecutarse, y puede que solo necesitaran unas pocas unidades de tiempo de la CPU. Al contrario, en algoritmos como RR (Round Robin), los procesos se van ejecutando en intervalos de ocupación de la CPU, llamados Quantum. De este modo el último proceso comienza a ejecutarse antes, en contraposición haciendo que todos terminen más tarde. Ahora bien, para ejecutar cualquiera de estos algoritmos, se necesita una estructura mínima compuesta de CPU y memoria. En la memoria es donde almacenamos los procesos que van a ser ejecutados. En este caso dicha memoria es no contigua y no reubicable. ¿Qué significa esto? Bien, que una memoria sea no reubicable significa que una vez el proceso que la ocupa haya terminado, el hueco restante no será rellenado por los otros procesos que la ocupan de manera simultánea; en el siguiente momento de CPU. Por otro

lado, que la memoria sea no continua nos dice que un mismo proceso puede ocupar cachos de memoria no contiguos. Por ejemplo, una memoria de 16 huecos cuyos 5 primeros huecos los ocupa el proceso 1, los siguientes el proceso 2, y de nuevo los siguientes el proceso 1. Veremos varios ejemplos en los siguientes puntos.

PASÁNDO DE LA TEORIA AL ALGORITMO

Vale, una vez sabemos las partes que componen la estructura que hace posible la ejecución de este algoritmo, tenemos que pasarlo al script de la manera más inteligente posible. No tenemos que tratarlo como dos algoritmos distintitos, el de memoria y el de CPU, si no que, para una mejor ejecución, tienen que trabajar entre ellos, comunicarse.

En los siguientes puntos vamos a explicar paso a paso el algoritmo de FCFS con varios ejemplos y después comprobaremos como los resuelve el programa paso a paso. Tras ello repasaremos todas las características adicionales implementadas en el programa y destacaremos las nuevas características añadidas respecto a versiones anteriores.

Tras esto haremos un análisis estructural del programa y por último un apartado para ayudar a los que vengan, ver que se puede mejorar. No olvidarnos de las herramientas, consejos y las fuentes de información. Esperamos que esta guía al algoritmo implementado sea útil y con el paso de los años se convierta en un programa útil y casi perfecto. Esperamos también que entiendas el código y... ¡Qué no se convierta en Legacy!

EJEMPLOS

EJEMPLO 1. FCFS SENCILLO. PROCESOS NO PARTICIONADOS

Vamos a coger un ejemplo sencillo de FCFS y ha hacerlo a mano paso a paso. Después veremos como lo hace el programa. Con los siguientes datos vamos a ver cada instante en el pasa algo como avanza y que uso de memoria hace.

Tamaño de Memoria: 12

Procesos

Nombre del proceso	Tiempo de Llegada	Tiempo de Ejecución	Tamaño en Memoria
P01	6	3	1
P04	10	11	8
P02	11	10	8
P05	17	1	4
P03	18	4	9

En la tabla anterior tenemos los procesos ordenados por tiempo de llegada.

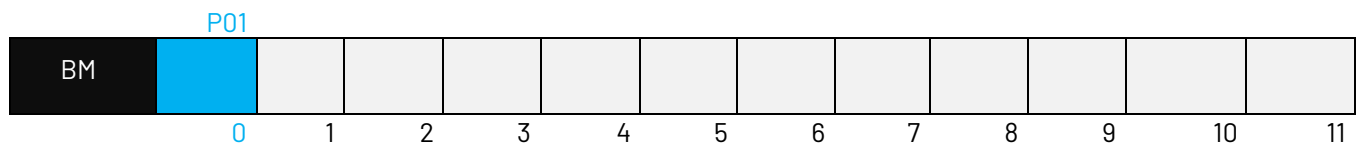
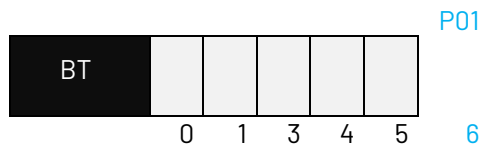
T=0

BT

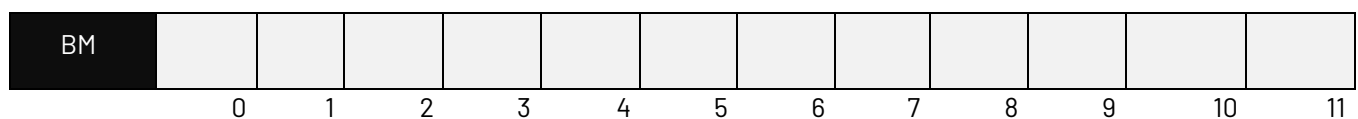
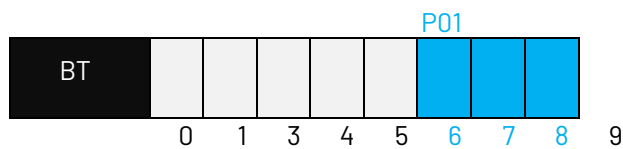
0

BM	0	1	2	3	4	5	6	7	8	9	10	11

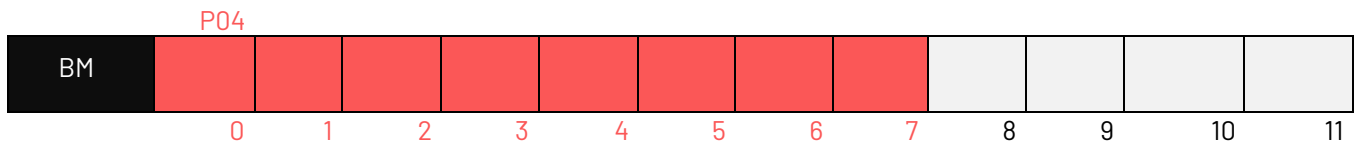
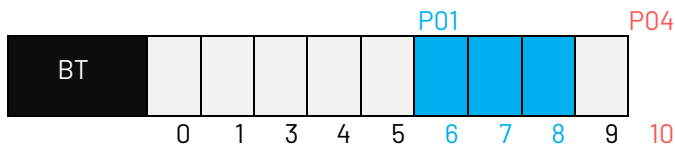
En tiempo T=0, ninguno de los procesos entra en memoria ni comienza a ejecutarse, ya que ninguno tiene tiempo de llegada 0. Por lo tanto, la banda de tiempo, donde vamos poniendo los procesos según se ejecutan, está vacía.

T=6

En tiempo $T=6$ llega el proceso P01 al sistema, se comprueba que este va a entrar en memoria. Por lo que cambiamos su estado a "En Memoria". Tras esto, como la CPU no está ocupada con ningún otro proceso, el estado de este pasará a ser "En Ejecución". Al ser un algoritmo FCFS, este se ejecutará desde $T=2$ hasta $T=2+(\text{tiempo de ejecución})$; es decir $T=2+3=5$.

T=9

En tiempo $T=9$ el proceso ha terminado de ejecutarse, por lo que pasa al estado "Finalizado". En este momento la CPU queda libre para el siguiente proceso en memoria.

T=10

En tiempo $T=10$ entra el proceso P04 entra a la CPU tras un instante sin ejecuciones y ocupa 8 huecos de memoria. Se prepara para ejecutarse desde $T=10$ hasta $T=10+11=21$. Cualquier proceso que venga detrás tendrá que esperar en memoria o fuera de ella si no entra, hasta que este proceso deje de usar la CPU. Esto, como hemos comentado antes, aumenta mucho el tiempo medio de espera de todos los procesos.

T=11

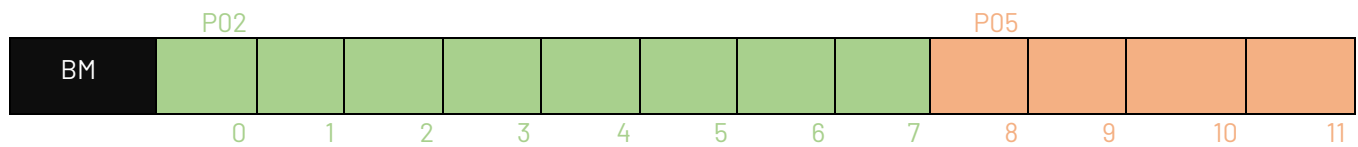
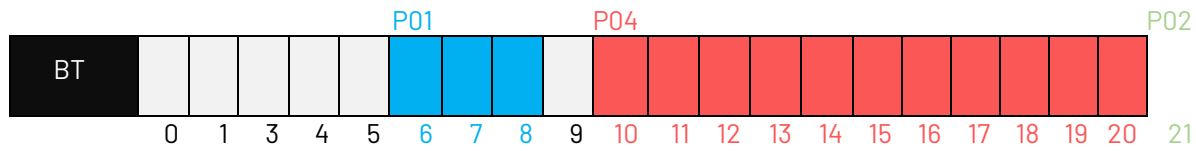
El proceso P04 comienza a ejecutarse mientras el proceso P02 se coloca en espera, ya que no entra en memoria. (huecos libres: 4, huecos necesarios: 8). Como este proceso está en espera, cualquiera de los procesos que van detrás (tiempo de llegada mayor que él), tendrán que esperar a que este entre en memoria y después se ejecute; aunque alguno de ellos si entre en memoria.

T=17

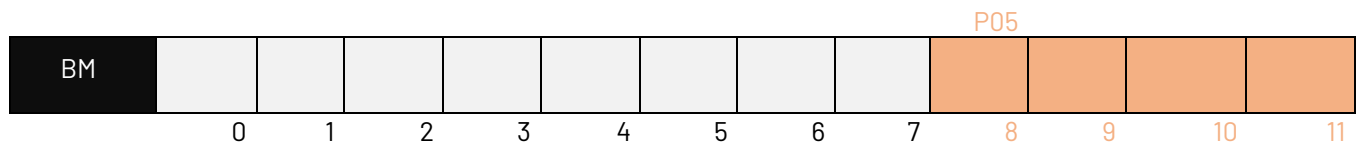
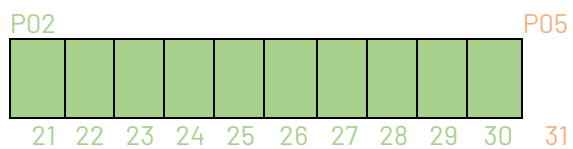
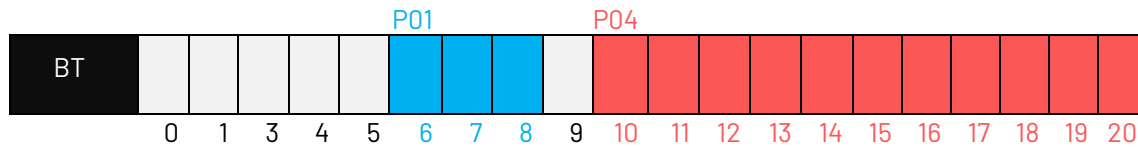
El proceso con tiempo de llegada 17, P05, se coloca en espera, a pesar de que entra en memoria (huecos libres: 4, huecos necesarios: 4). Por lo tanto, ambos deben esperar a que el proceso P04 acabe de ejecutarse. Esto ocurrirá en $T=21$.

T=18

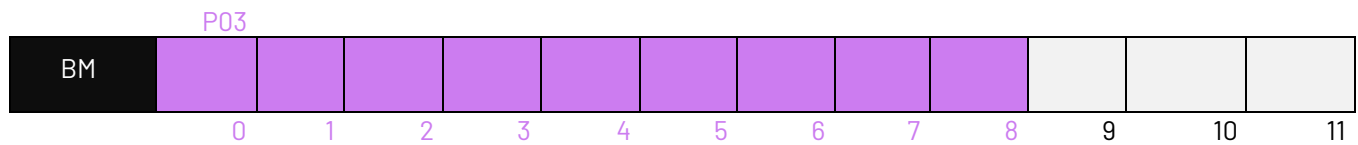
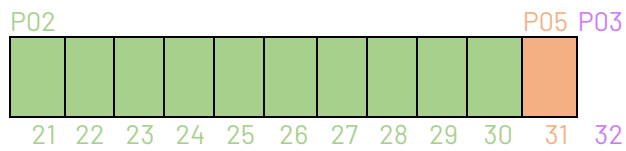
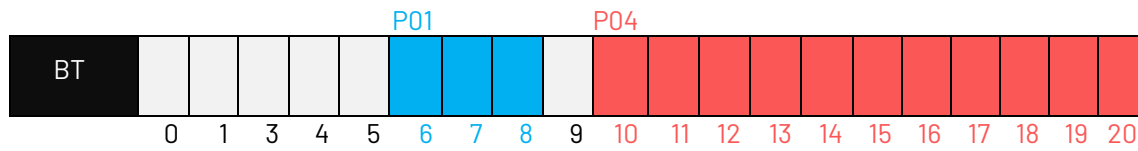
En $T=18$ llegará el proceso P03 y se colocará a la espera como los anteriores procesos. En este momento el P04 estará a 3 instantes restantes de la completa ejecución.

T=21

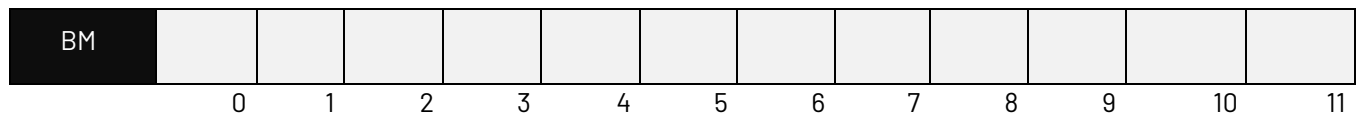
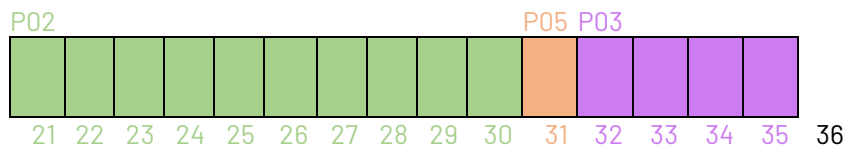
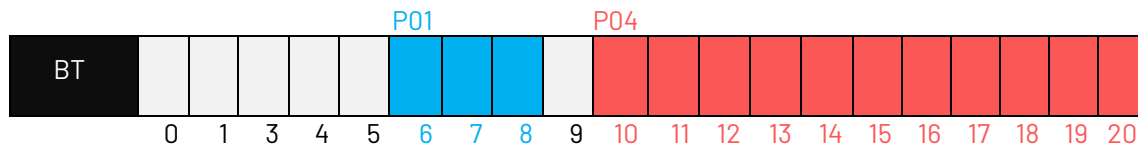
En tiempo $T=21$ el proceso P04 ha finalizado, por lo que deja paso al P02. Este entra en memoria y como deja hueco para el siguiente proceso en espera (P05), también entra este. P02 comienza a ejecutarse desde $T=21$ hasta $T=21+10=31$

T=31

En tiempo $T=31$, el proceso P02 ha finalizado y libera la memoria para el siguiente proceso que quiera entrar, el P03. El proceso que ya estaba en memoria comienza su ejecución desde $T=31$ hasta $T=31+(\text{tiempo ejecución P05})=31+1=32$

T=32

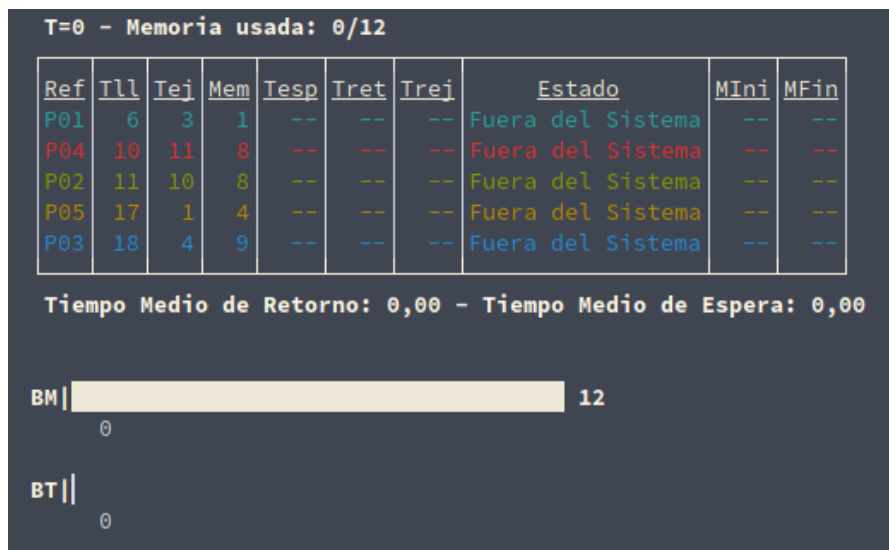
En tiempo $T=32$, el proceso P03 entra en memoria, cuando el proceso anterior, P05, al terminar la ejecución, le ha dejado hueco suficiente (necesitaba 9 huecos, tenía 8). Este proceso se ejecutará hasta el instante 36, pues su tiempo de ejecución es 4.

T=36

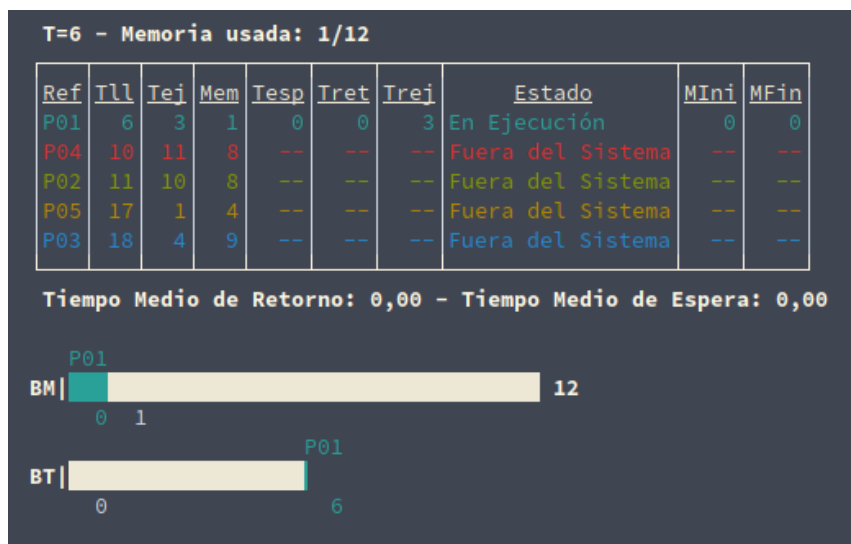
En tiempo T=36, podemos ver que la memoria se ha vaciado por completo. Esto se debe a que no queda ningún proceso restante en espera o que todos lo que lo hacen no caben en memoria y no sin ni contemplados. Podemos ver ahora que la línea de tiempo llega hasta 36 instantes de CPU.

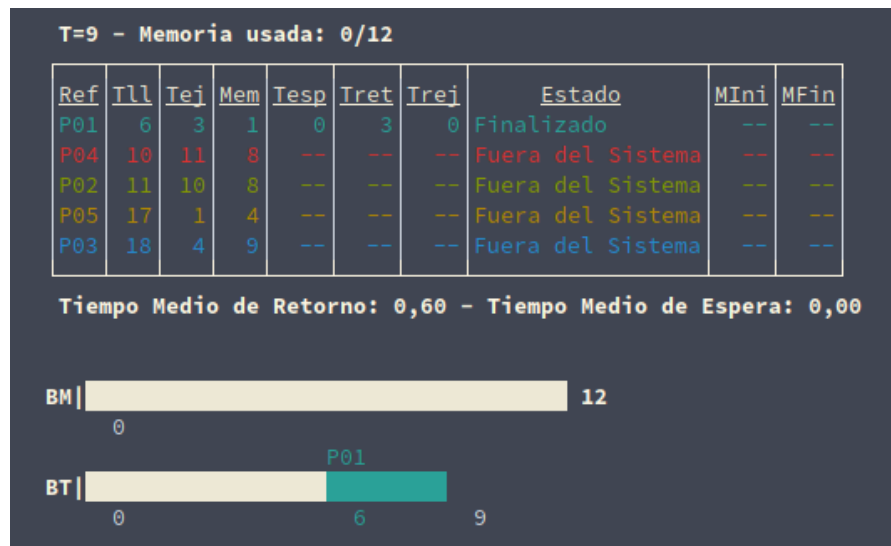
Ahora veamos como empleamos este mismo ejemplo en el script.

EJEMPLO 1 EN SCRIPT. FCFS SENCILLO. PROCESOS NO PARTICIONADOS

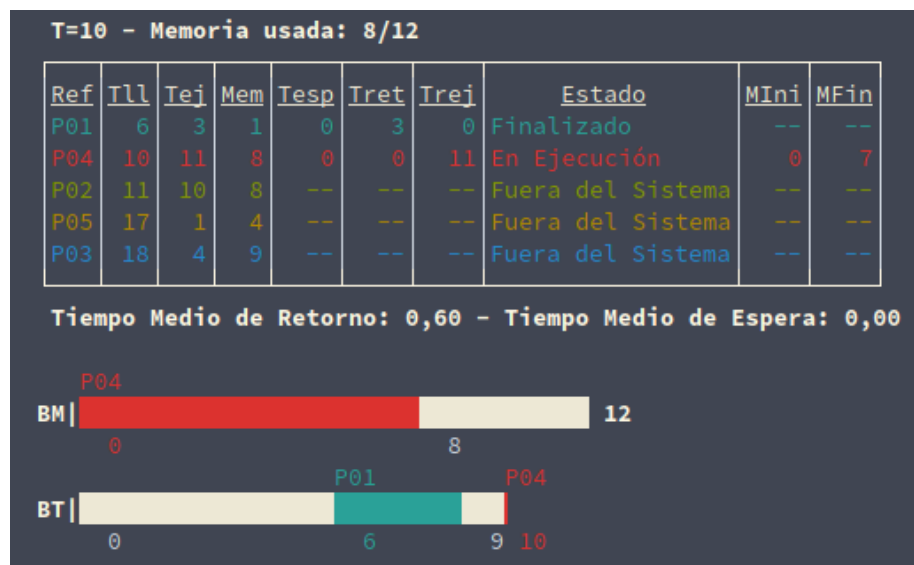


Como podemos ver, en la versión del script, podemos hacer más cálculos en cada interacción. En este caso vemos la columna del estado de los procesos. También vemos el tiempo de espera y de llegada, así como el tiempo restante de ejecución de cada uno. En las dos últimas columnas tenemos información sobre las posiciones que ocupan en memoria los procesos. Como el algoritmo es de memoria no continua y no reubicable los procesos pueden ocupar espacios no contiguos de la memoria. Si esto sucede, al proceso en cuestión se le otorgará una línea adicional y en estas columnas se indicará cada hueco ocupado en la memoria. Veremos esto en el siguiente ejemplo.

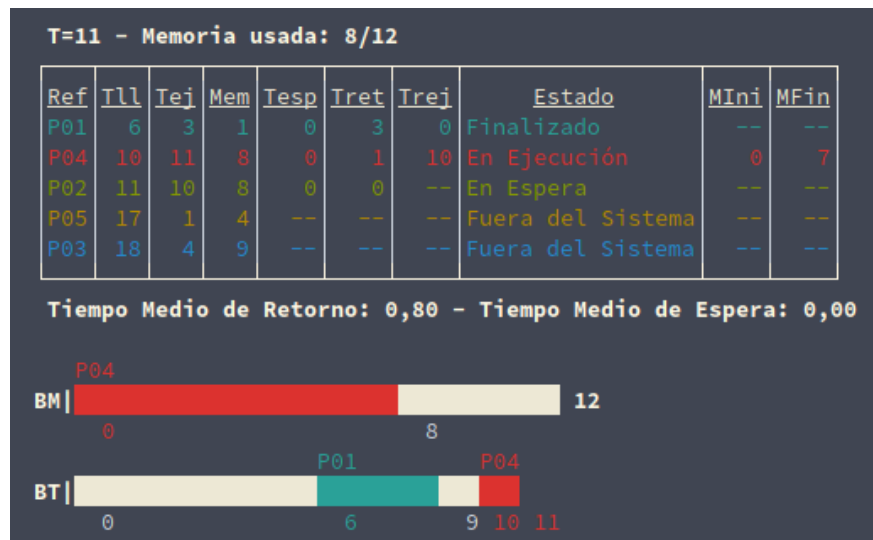




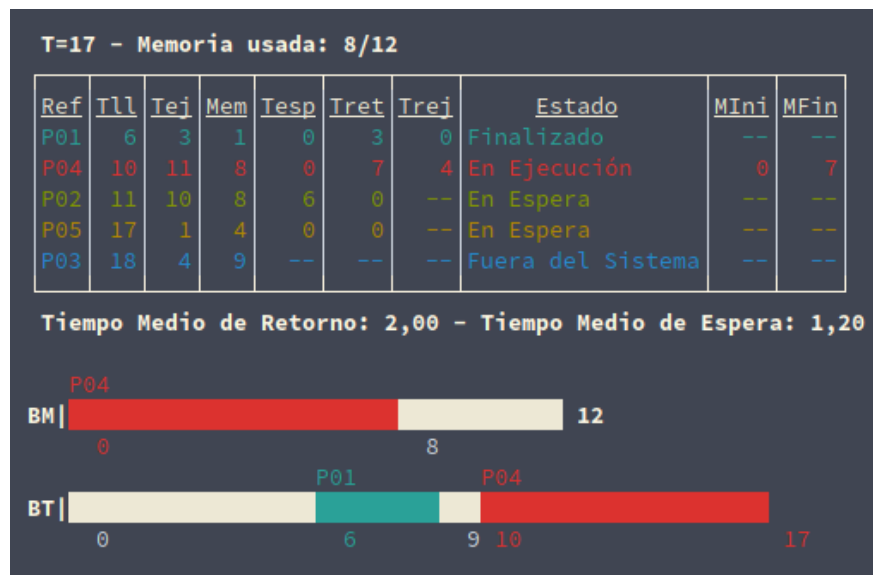
Podemos ver como en T=9 el proceso P01 ha pasado a estado finalizado y la memoria ha sido liberada. Por otro lado, vemos como el tiempo medio de respuesta ha sido actualizado. El de espera no parece haberlo hecho, pero eso es porque el proceso no ha tenido que esperar para empezar a ejecutarse, dando una media de 0.



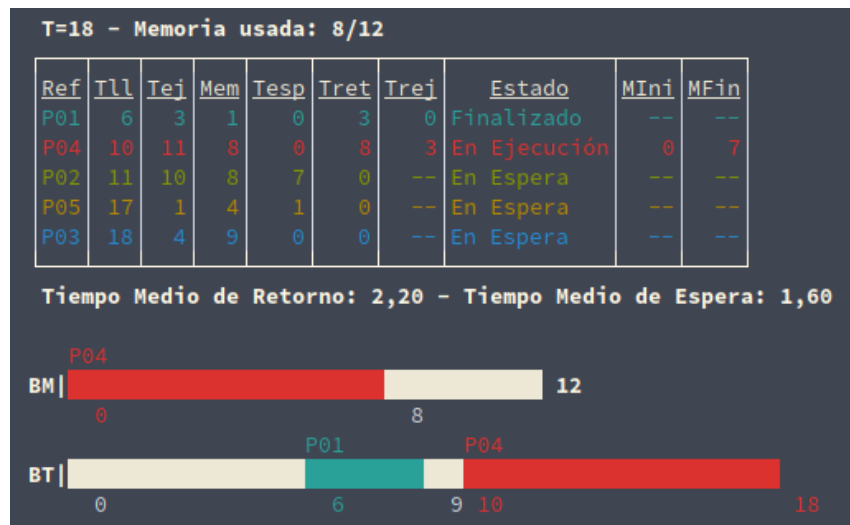
En T=10, podemos ver como el P04 entra en memoria, desde la posición 0 a la 7, como vemos en las dos últimas columnas. Su estado pasa a "En Ejecución", listo para ejecutarse en los siguientes instantes.



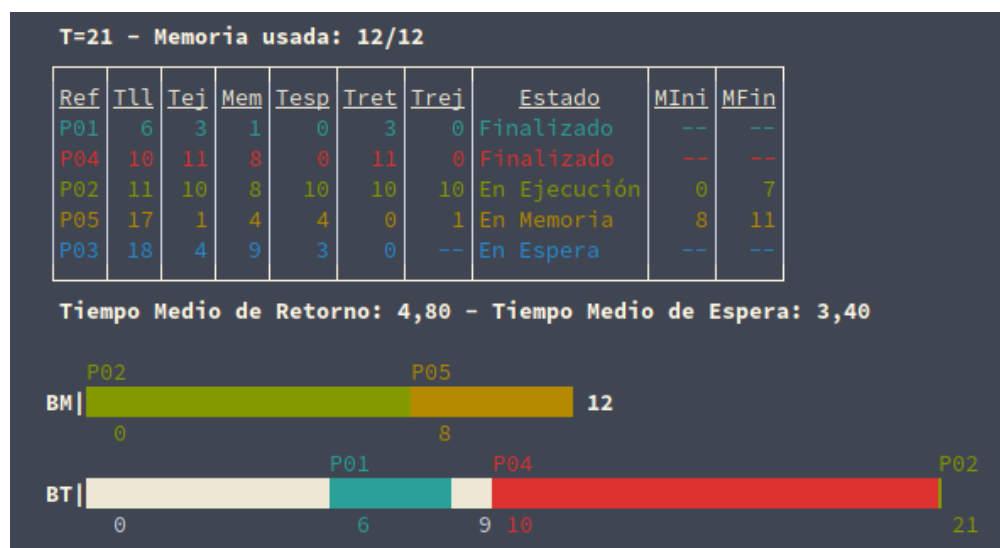
Solo ha pasado un instante, pero el programa nos indica que uno de los procesos se ha colocado "En Espera". Desde ese momento su tiempo medio de espera va a aumentar, aumentando por lo tanto el tiempo medio de todos los procesos.



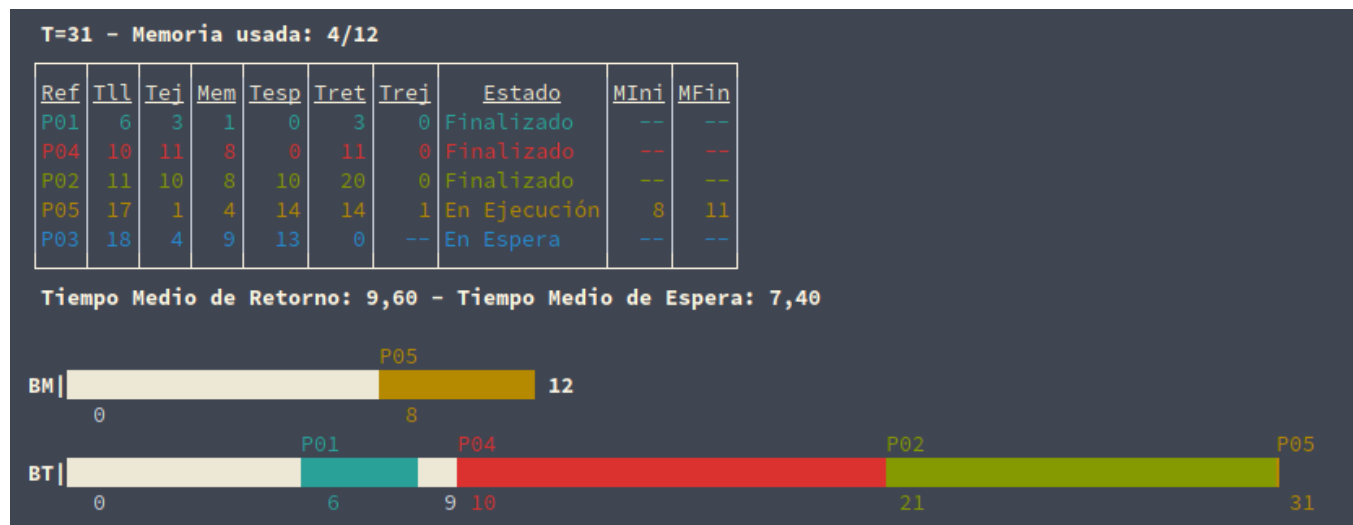
Todavía está ejecutándose el proceso P04, pero el programa para en el instante 17 para enseñarnos que el siguiente proceso P05, aun cabiendo en memoria, se ha colocado en espera. Vemos también como el tiempo restante nos indica que el P04 acabará de ejecutarse en el instante +4 al actual, es decir en el 21.



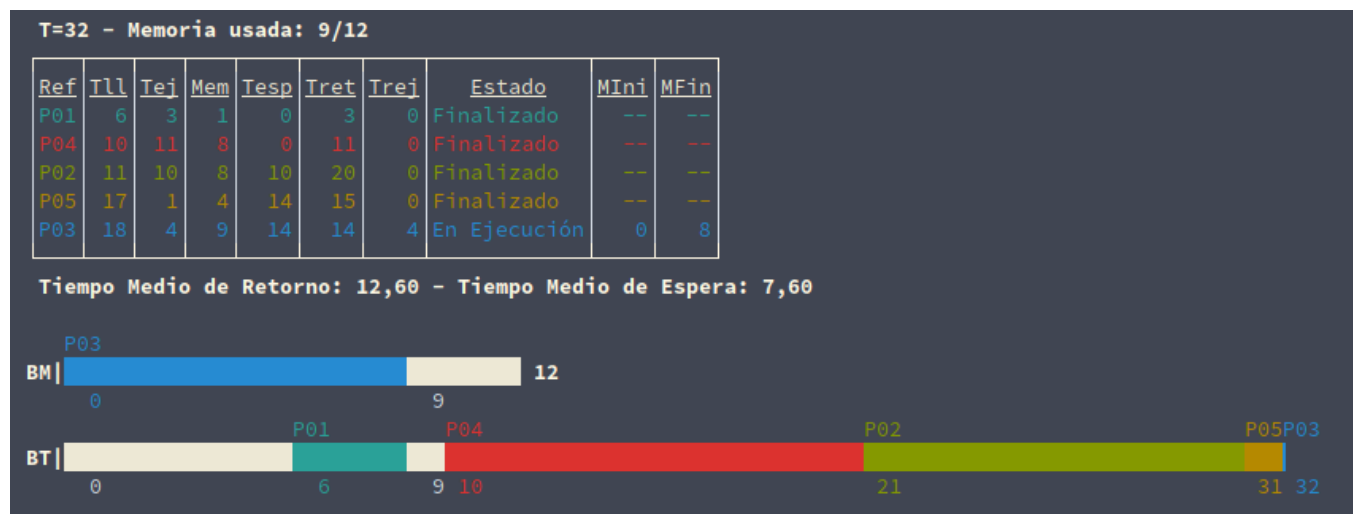
De nuevo vemos como el último proceso se coloca en espera.



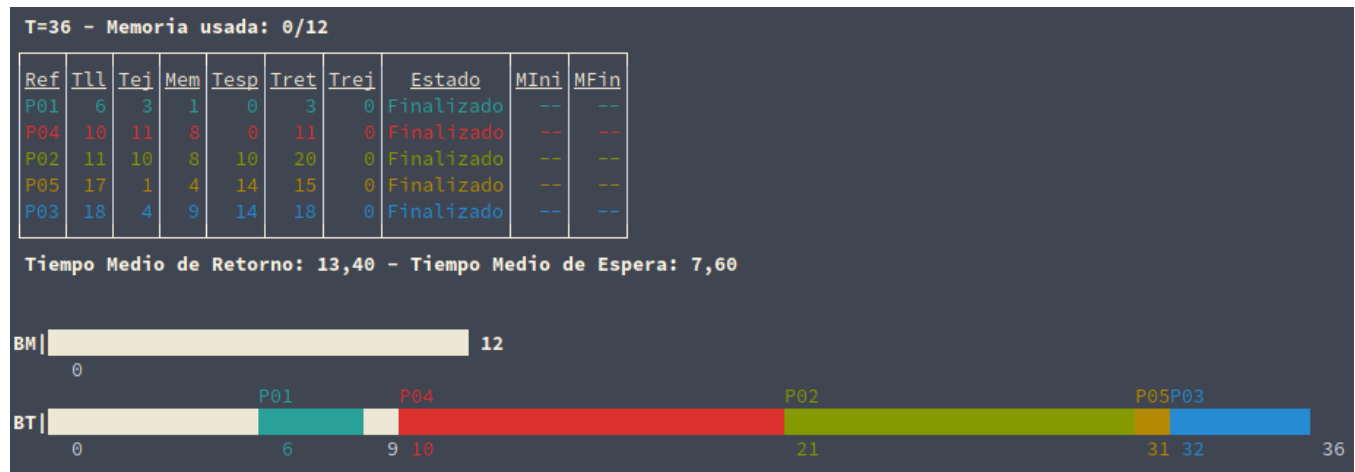
El proceso P05 termina y podemos ver que el siguiente en ejecutarse es el P02. Este ha entrado en memoria e incluso hay hueco para el siguiente, por lo tanto, ambos ponen su estado en “En Memoria”. Lo que pasa con el P02 es que no Oslo ha entrado en memoria, si no que también pasa a “En Ejecución”, sobrescribiendo el anterior. Podemos ver como las dos columnas finales nos indican las posiciones de todos los procesos en memoria.



Como todos los procesos han llegado al sistema, el programa ahora solo mostrará los instantes en los que cualquiera de los procesos termine su ejecución. En este caso vemos que el P02 ha terminado y el siguiente se coloca en “En Ejecución”. El proceso inmediatamente posterior (P03) no puede entrar en memoria porque no cabe.



El proceso P05 termina de ejecutarse y es el turno del último proceso.



En el instante 36 todos los estados de los procesos se marcan como "Finalizado" y la memoria la vemos vacía. La línea de procesos muestra todos los instantes en los que la CPU ha estado en uso, o en desuso (huecos blancos).

EJEMPLO 2. FCFS COMPLEJO. PROCESOS MULTIPARTICIONADOS

Para este ejemplo 2, solo mostraremos los instantes en los que se produce la partición de un proceso en varios cachos dentro de la memoria. La tabla de datos es la siguiente.

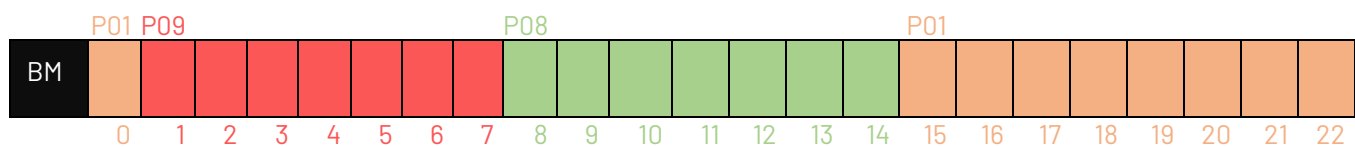
Tamaño de Memoria: 23

Procesos

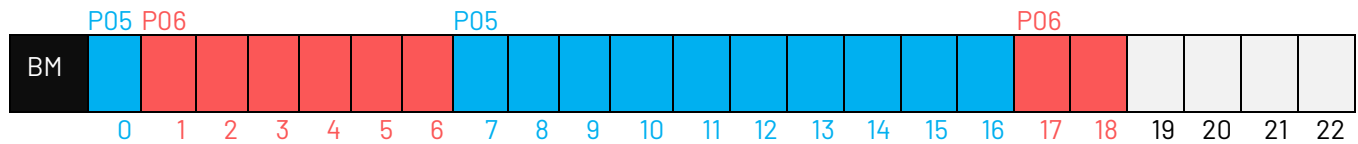
Nombre del proceso	Tiempo de Llegada	Tiempo de Ejecución	Tamaño en Memoria
P03	0	14	10
P09	0	14	7
P08	3	19	7
P01	4	15	9
P04	5	13	6
P05	8	3	11
P06	9	7	8
P07	10	18	14
P02	12	25	1

Avanzamos en la ejecución del algoritmo de la misma manera que el anterior. La línea de CPU no cambia, pero la banda memoria si tiene características diferentes. Avanzamos hasta el instante 14.

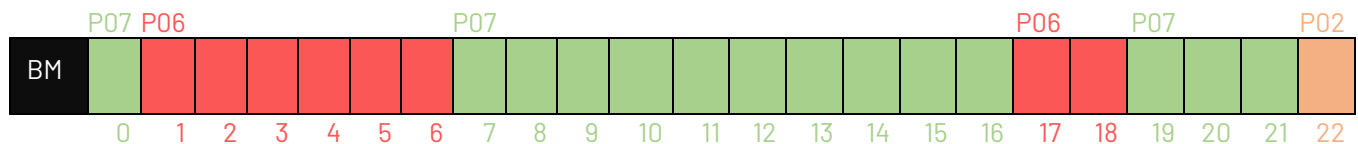
T=14



En tiempo T=14, vemos que el proceso P01, que es siguiente en la cola de espera, tiene hueco en memoria. Como la memoria es no continua, entra y se distribuye en los huecos que tiene partiéndose en dos y ocupando la primera posición de memoria y las últimas 8 [15-22].

T=75

Debido a que el tiempo de ejecución está alrededor de 14 instantes, llegamos al instante 75 y podemos ver que no solo un proceso ocupa varios huecos separados de la memoria, si no que dos procesos hacen esto de forma intercalada. El proceso P05 en azul ocupa las posiciones 0 y [7-16]. Por otro lado, el proceso P06, en rojo, ocupa las posiciones [1-6] y [17-18].

T=78

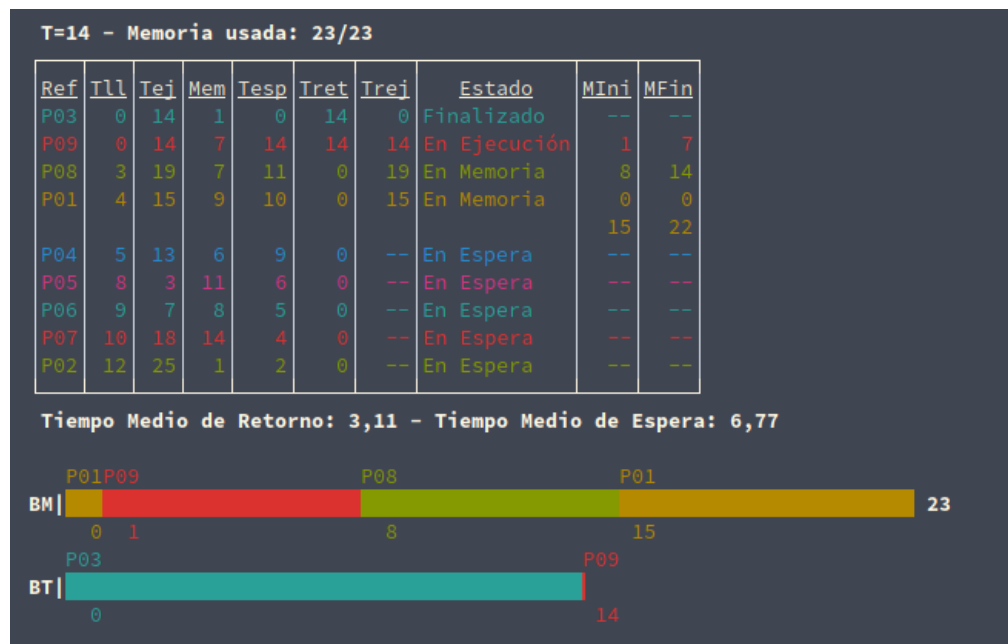
Por último, tenemos la situación de un proceso, que no está partido en dos, si no que se aloja en la memoria en tres huecos no contiguos entre sí. En este caso el proceso P07 se aloja en los huecos que han quedado tras la finalización del proceso P05.

Veámoslo desde el script.

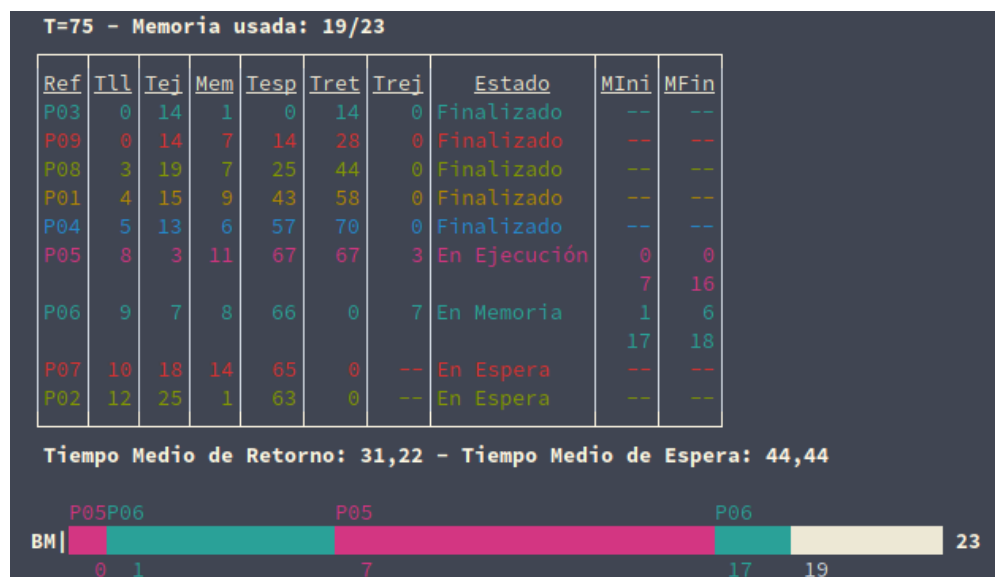
EJEMPLO 2 EN SCRIPT. FCFS COMPLEJO.

PROCESOS MULTIPARTICIONADOS

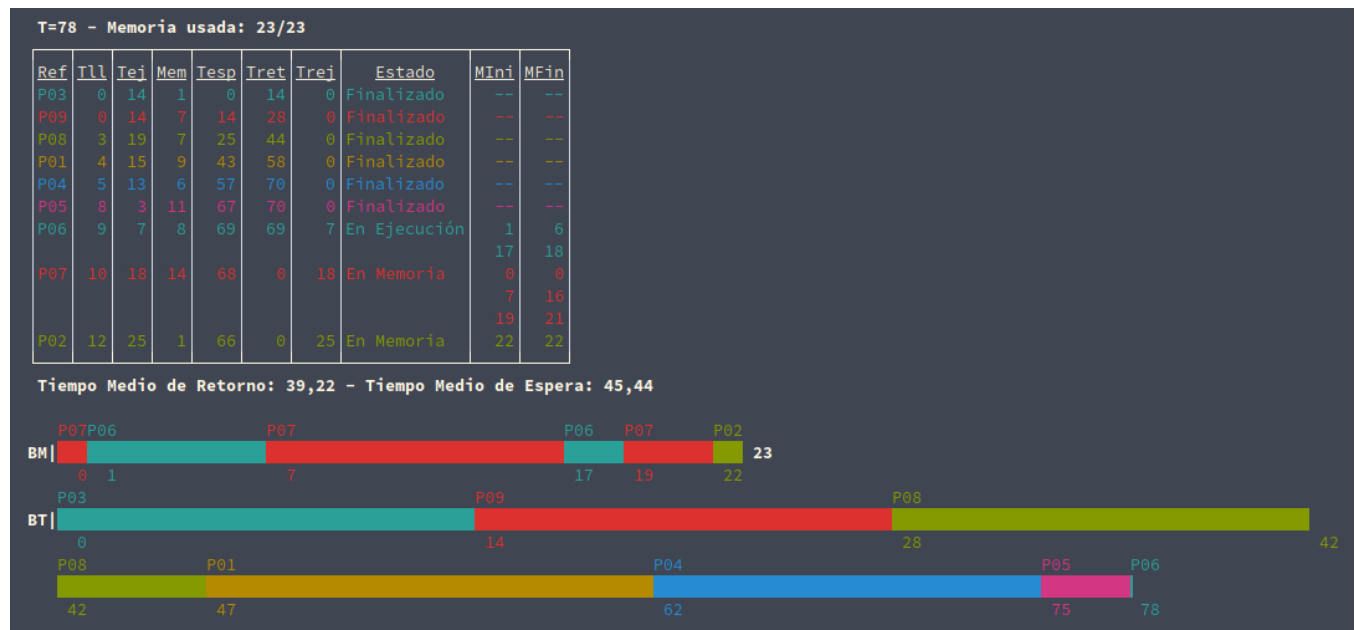
De nuevo, vamos a destacar los instantes en los que los procesos ocupan de la memoria de forma no contigua.



Como hemos indicado anteriormente, el proceso P01 está dividido en dos. Vemos como en la tabla de procesos el P01 tiene dos filas, cuyas dos últimas columnas marcan las posiciones de inicio y final de los huecos que ocupa en memoria.



En esta imagen podemos ver como ambos procesos, en azul y rosa se intercalan ocupando la mayoría de la memoria. En sus respectivas filas en la tabla de procesos vemos que se indican el comienzo y el final de los 4 huecos que ocupan.



Por último, podemos observar como el P07 ocupa tres partes no continuas de la memoria, y de nuevo vemos como ahora tiene hasta tres filas dedicadas para el en las que ponemos los inicios y finales de los tres huecos ocupan. 0, [7-16] y [19-21]. Esto se puede ampliar a todas las particiones del proceso que ocurran. Por cada nuevo hueco que ocupe en memoria una nueva línea se añadirá con la información de comienzo y final de dicho hueco.

CAMBIOS Y MEJORAS

Aun siendo uno de los algoritmos más fáciles de implementar, no les quita importancia a las mejoras implementadas respecto a previas versiones. En este caso esto ocurre más pues justo la anterior versión estaba falta de muchas de las mejores que otras ya implementaban. Echemos un rápido vistazo al programa del que hemos partido para apreciar un poco más todas las mejoras añadidas.

PÁGINA DE INICIO

Al ejecutar el programa anterior, nos recibía el siguiente mensaje:

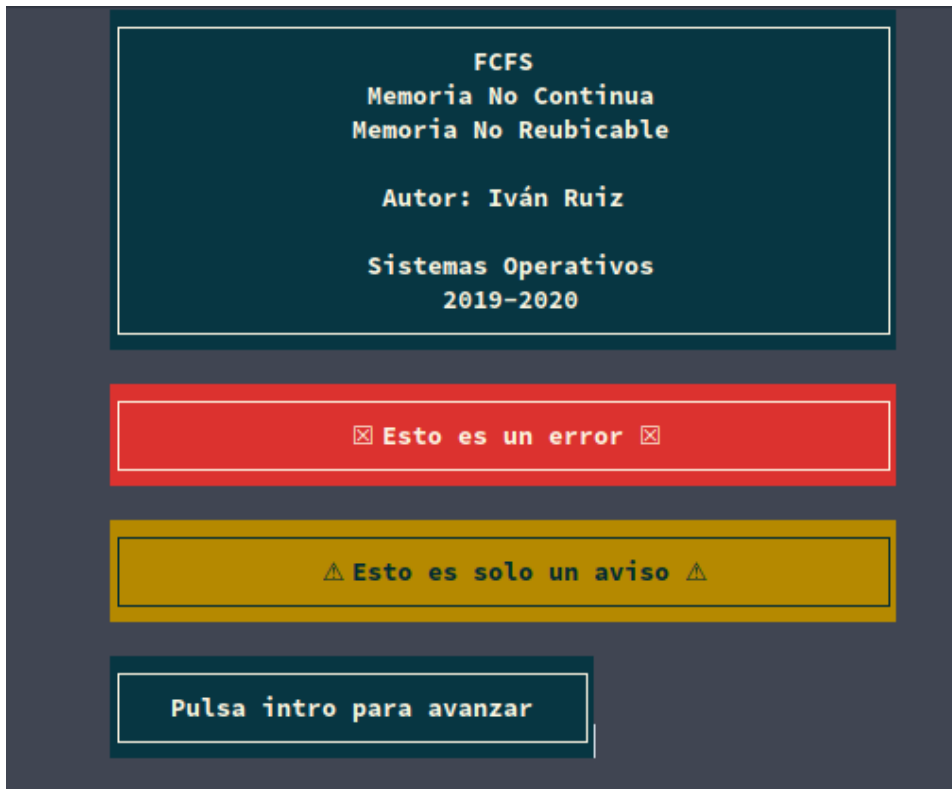
```
#####
#                               #
#      Creative Commons         #
#                               #
#      BY - Atribución (BY)     #
#      NC - No uso Comercial (NC) #
#      SA - Compartir Igual (SA) #
#####

-----
*                               *
*      Algoritmo FCFS          *
*      José Castañeda Palacín & Jesús Martínez Taboada      *
*      Versión Mayo 2017       *
*                               *
\-----/
Introduzca el numero de procesos a ejecutar
|
```

Para empezar, podemos observar dos cosas:

1. La falta de colores, aunque esto no es algo necesariamente malo.
2. La pregunta sobre el número de procesos a ejecutar. Al hacer esta pregunta no podemos meter procesos a gusto según los vamos introduciendo uno a uno.

Ambas cosas se arreglan en la nueva versión cuya entrada es la siguiente:



Podemos ver una cabecera mas sólida, con un borde continuo, así como un ejemplo de los posibles mensajes de advertencia y de error. Estos mensajes pueden saltar por una mala entrada de un dato o por la entrada de un dato que no encaje de forma correcta con la ejecución. Esta es otra mejora que se ha implementado, pues la anterior práctica, como veremos ahora, no tenía control de errores en algunos aspectos.

CONTROL DE ERRORES

Para comprobar este aspecto vamos a avanzar en el algoritmo al menú de cada uno y ver como gestiona una entrada errónea. En la versión anterior tenemos el siguiente menú:

```
Introduzca el numero de procesos a ejecutar
-2
Entrada no válida. Introduce un número de procesos entero:5
¿Desea introducir los datos de forma manual?(s/n):
f
Respuesta introducida no válida
Introduce una respuesta que sea s/n
s
```

No es como tal un menú, es una pregunta de si/no sobre si quieres introducir los datos de forma manual. Si introducimos un dato mal, podemos ver que nos repite la pregunta. Es algo confuso pues no se ve bien donde está el error y que falla. Veamos ahora como se ve el menú en la nueva versión:

MENÚ PRINCIPAL

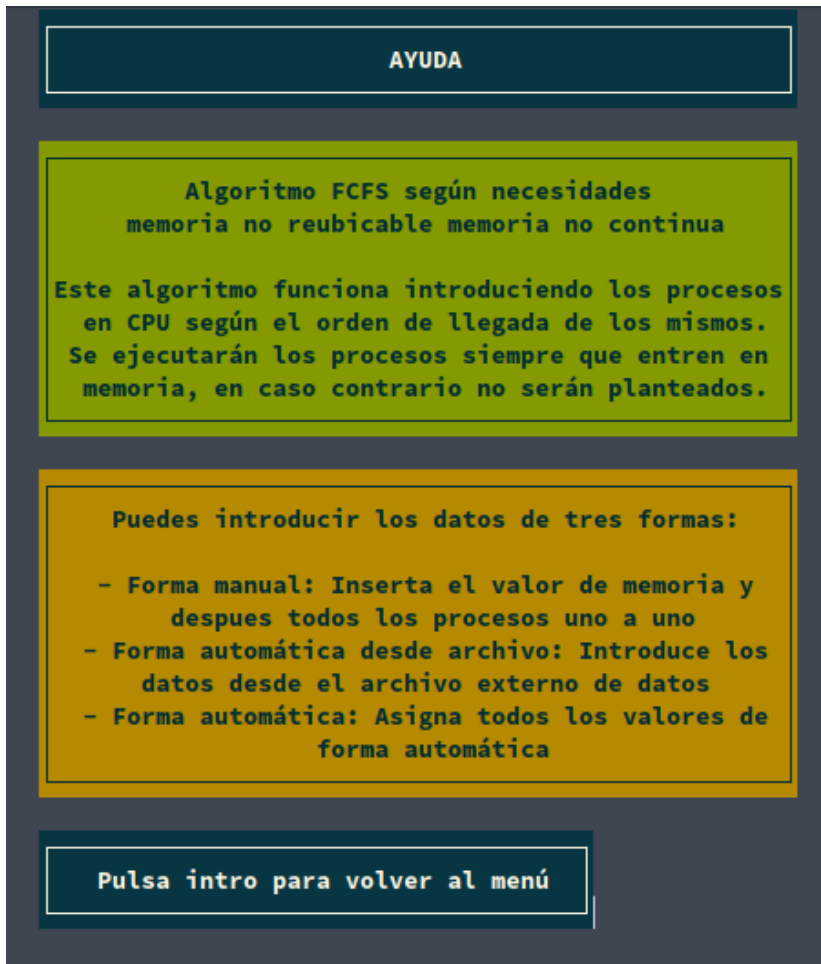
- 1.- Entrada manual por teclado
- 2.- Entrada automática por archivo
- 3.- Entrada automática con valores aleatorios
- 4.- Ayuda
- 0.- Salir

Respuesta: -

Inserta un valor numérico entre 0 y 4

Respuesta: 1|

Podemos elegir entre tres formas de entradas de datos. Manual, por archivo y aleatoriamente. También podemos ver una ayuda que nos explicará un poco el funcionamiento del programa. Por último, podemos salir del script.



Podemos volver al menú principal pulsando cualquier tecla. Vamos a elegir la entrada manual en ambos scripts.

ENTRADA MANUAL

Veamos la entrada manual que tenemos en script antes de ser mejorado.

Proceso	T. Llegada	T.Ejecución
proc1	2	3
pr2	24	5
p3	0	0
p4	2	2
p6	1	1

¿Tiempo de Espera Acumulado[A] o Real[R]?

Podemos destacar varias cosas de la entrada manual de este script:

1. Nos va preguntando el nombre, el tiempo de llegada y el tiempo de ejecución de cada proceso. Esto genera un problema, y es que como se puede ver en la imagen de arriba, saltamos del proceso 4 al proceso 6. El no poner los nombres de forma automática genera muchos problemas de comprensión posteriores.
2. No hay colores por ningún lado. El uso de un color único para cada proceso hace que los identifiquemos de forma rápida en cada uno de los lugares en los que les representamos: línea de memoria, línea de CPU o tabla de procesos.
3. Los procesos no se van ordenando según los metemos: Como podemos ver, los procesos no están en orden por tiempo de llegada
4. El aspecto de la tabla puede mejorarse sustancialmente, así como la justificación de los números a derecha y el texto a izquierda.
5. No pregunta el tamaño de la memoria en la que almacena los procesos. Veremos más tarde por qué.

Antes de ver si el nuevo script implementa las nuevas mejoras, podemos destacar el hecho de que este algoritmo tiene algo que el nuevo no, y esto es la opción de dar a elegir al usuario si quieren tiempo acumulado o tiempo real. Como es algo no muy importante, hemos decidido dejarlo fuera en las nuevas versiones. A parte de que tenía algún que otro fallo:

```

gTiempo de Espera Acumulado[A] o Real[R]?

./FCFS.sh: línea 248: [: demasiados argumentos

Introduce el nombre del proceso_6:
1
Introduce tamaño del proceso_6:
3
./FCFS.sh: línea 482: [: ==: se esperaba un operador unario
./FCFS.sh: línea 482: [: ==: se esperaba un operador unario
expr: falta un operando
Pruebe 'expr --help' para más información.
expr: syntax error: missing argument after «+»
expr: syntax error: unexpected argument «5»

```

En la anterior imagen podemos ver como la entrada de la pregunta de tipo de tiempo de espera no tiene control de errores y produce un fallo. También vemos como cuando respondemos el programa nos pregunta por otro proceso del que no debería preguntarnos; lo cual genera aún más errores. Ahora vemos como este fallo afecta más adelante. Comprobemos ahora la entrada de datos de la nueva versión:

Podemos observar que lo primero que se pregunta es el tamaño de la memoria. Tras esto pasamos a preguntar el

Tamaño de la memoria

Respuesta: 14

Tamaño del proceso 1 - (Memoria: 14)

Ref	Tll	Tej	Mem
P01	3	5	

Respuesta: 15

¡El tamaño del proceso es mayor que la memoria! No podrá ejecutarse

Respuesta: -2

Valor de tiempo de tamaño no válido, entero mayor que 0

Respuesta:

tiempo de llegada, el tiempo de ejecución y por último el tamaño de cada proceso. Esto último no puede ser mayor que el tamaño de memoria introducido en la ventana anterior. Como se puede observar hay un control de errores para las distintas entradas, como un tamaño mayor o una entrada negativa. Por otro lado, según vamos metiendo datos, estos van recibiendo un nombre de forma automática. También se ordenan según el orden de llegada para que el usuario se entere bien de todos los procesos que está introduciendo. Vemos también como se organizan en una pequeña tabla

Ref	Tll	Tej	Mem
P02	1	4	3
P03	2	2	2
P01	3	5	7

limpia y con sus respectivos colores por proceso.

EJECUCIÓN DEL ALGORITMO

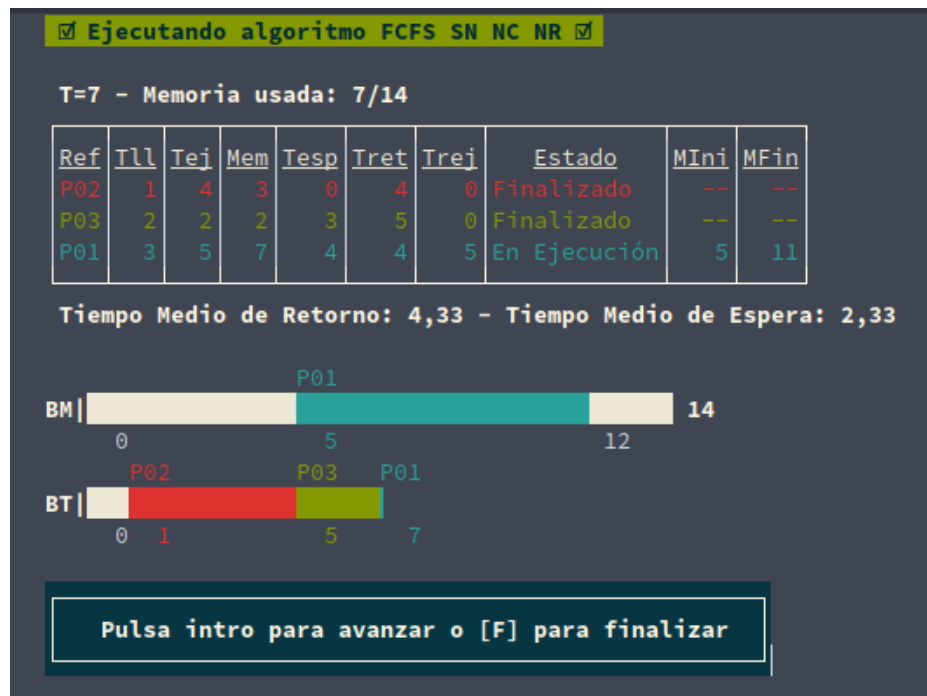
Antes de analizar el resto de los tipos de entradas implementadas, vamos a analizar la chica de la práctica, la ejecución. Siento decir que, en la anterior versión, simplemente, no hay. El algoritmo salta directamente hasta el final, donde nos pregunta si queremos ver el informe.

Proceso	Llegada	Ejecución	Espera	Respuesta
* P02	* 1	* 4	* 0	* 4
* P03	* 2	* 2	* 3	* 5
* P01	* 3	* 5	* 4	* 9

* T.espera medio: 2 - * T.retorno medio: 6

¿Quieres abrir el informe? ([s],n): |

Podemos observar columnas adicionales para la espera y para la respuesta. Veamos la última versión:



Para empezar, podemos observar varias diferencias. Se ha implementado una cabecera del programa, así como la línea con el instante en el que nos encontramos. Vemos varias columnas más, como la columna de estados de procesos, la de tiempo restante de ejecución, la de memoria, que no estaba implementada en el anterior. Por lo tanto, tampoco tenemos en la versión anterior las columnas de inicio y final de memoria. Lo que sí tenemos en ambos en la línea con el tiempo de espera y retorno medio. Tampoco observamos ninguna línea de tiempo o memoria. Así como color alguno.

INFORME DE ALGORITMO

Bien, tras ejecutar el algoritmo, extraemos la salida a varios informes, para que el usuario guarde el registro del script. En este caso, el algoritmo ya lo tenía implementado, solo que algo básico y lleno de errores. Para empezar uno básico y es que no usa un comando universal para todas las distribuciones. Te hace descargar un paquete para verlo desde el programa. Por otro lado, el informe generado es totalmente en blanco y negro. No es raro pues no se ha usado ni una sola vez un color de identificación de proceso.

```
¿Quieres abrir el informe? ([s],n): s
FCFS.sh: línea 572: gedit: orden no encontrada
```

Esto cambia en la nueva versión. Podemos ver un menú en el que podemos elegir la forma de ver el informe, en color, o en blanco y negro:

```

MENU DE INFORME

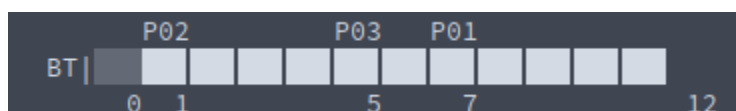
1.- Informe en color
2.- Informe en blanco y negro
0.- No sacar informe

Respuesta: -

Inserta un valor numérico entre 0 y 2

Respuesta: |
```

Como siempre, mantenemos un control de errores. Y no solo esto si no que la salida del algoritmo se ha hecho de tal forma que no pierda información en blanco y negro. Como dependemos de los colores para identificar los distintos procesos y las posiciones que ocupan en memoria, se ha optado por usar un carácter especial que ayude en esta tarea:



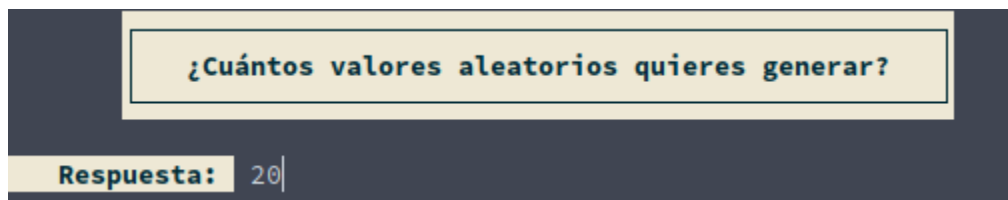
El cuadrado ayuda a identificar un instante de tiempo o un hueco de memoria.

ENTRADA POR ARCHIVO

La versión anterior no tiene ningún tipo de entrada por archivo, por lo que para repetir un programa tienes que meter los datos de nuevo. Esto ha sido implementado mediante un archivo CSV que guarda todos los datos necesarios de la manera más ordenada posible. No importa en que línea metas el nuevo proceso. Tampoco importa si cometes algún error en los números o si hay alguna línea vacía, el programa se encargará de eliminar todos los errores.

ENTRADA AUTOMÁTICA

Ambas versiones tienen una entrada de datos automática, que colocan datos aleatorios. Este es el único punto en el que pedir el número de procesos tiene sentido. Veamos la nueva versión:



A screenshot of a terminal window with a dark background. At the top, a light yellow box contains the prompt "¿Cuántos valores aleatorios quieres generar?". Below this, a light yellow box contains the text "Respuesta:" followed by the number "20" and a cursor.

CAMBIOS INTERNOS

Algo muy importante, a parte de todos los cambios estéticos, son los cambios internos en el programa. Es la parte invisible, pero es muy importante. No porque el programa corra y sea bonito significa que esta implementado de forma perfecta. Cosas que han sido mejoradas:

- Velocidad de ejecución. Mejores algoritmos.
- Limpieza de código inútil.
- Legibilidad y comentarios. Documentación.
- Mayor uso de funciones. Eliminación de código repetido.
- Reducción del uso de variables globales. Mayor control.

AYUDA AL PROGRAMADOR

Para ayudar al programador que venga después, se han dejado notas de como cambiar el código ligeramente para implementar distintas opciones de color, tamaño, bordes, estilos, etc...

Un ejemplo es esta función en la que se puede elegir el estilo de los bordes de las tablas y cuadros. Es algo trivial, pero hay más por todo el código. Si lo estás leyendo, ¡Espero que te ayuden a seguir mejorando este programa!

```
# Almacen de los estilos de las tablas con sus códigos ASCII
# @param Numero de estilo
# -----
function asignarEstiloGeneral() {
    local estilo1=( "=" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" )
    local estilo2=( "-" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" )
    local estilo3=( "-" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" )
    local estilo4=( "-" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" "┌" "┐" "└" "┘" )

    case "$1" in
        1)
            estiloGeneral=( "${estilo1[@]}" )
            ;;
        2)
            estiloGeneral=( "${estilo2[@]}" )
            ;;
        3)
            estiloGeneral=( "${estilo3[@]}" )
            ;;
        4)
            estiloGeneral=( "${estilo4[@]}" )
            ;;
        *)
            estiloGeneral=( "${estilo4[@]}" )
            ;;
    esac
}
```

La documentación encima de las funciones ayuda mucho en la generación de ADOC y otros documentos de registro del código. Veámoslo en el siguiente punto.

ARCHIVO CONFIG

Se proporciona un archivo config (toml) en el que el usuario puede cambiar fácilmente datos básicos del programa, como el nombre del algoritmo, el autor, fecha, etc. Puede ampliarse para cualquier variable global del programa.

DOCUMENTACIÓN GENERADA

GENERACIÓN DE DOCUMENTOS CON ZSDOC

Es muy importante mantener el código ordenado, porque llega el momento de terminar el programa y generar la documentación; y como el código no esté debidamente señalado y colocado, la documentación generada puede ser un desastre. Si nosotros generamos la documentación de nuestro script, conseguimos un archivo ADOC con toda la información. Desde este archivo podemos generar otros como PDFs, HTML o DOCX. A continuación, vamos a insertar la documentación generada y después vamos a explicar un poco como usarla para generar documentación en las siguientes versiones del programa.

ADOC DEL SCRIPT

INFORMACIÓN

Script que simula la ejecución de un algoritmo FCFS en sistemas operativos

FUNCTIONS

```
asignarColores
asignarDatosInicial
asignarDesdeArchivo
asignarEstadosSegunInstante
asignarEstiloGeneral
asignarManual
asignarValoresAleatorios
avanzarAlgoritmo
calcularCambiosCPU
calcularCambiosMemoria
calcularLongitud
calcularMemoriaRestante
cc
centrarEnPantalla
colocarNombreAProcesos
comprobarProcesosEjecutando
copiarArray
elegirTipoDeEntrada
eliminarProcesosNoValidos
extraerDeConfig
fc
imprimirAyuda
imprimirCuadro
imprimirLineaProcesos
imprimirMemoria
imprimirTabla
main
ordenarArray
procesosHanTerminado
recibirEntrada
sacarHaciaArchivo
```

DETAILS

Script Body

Has 3 line(s). Calls functions:

Script-Body

```
`-- main
```

asignarColores

Asigna los colores usados en todo el algoritmo

@param Desde una posicion(n) o desde el principio("")

Has 12 line(s). Doesn't call other functions.

Called by:

asignarManual

elegirTipoDeEntrada

asignarDatosInicial

Asigna los estados de los procesos

Has 13 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. `command_not_found_handle` or called indirectly in other way).

asignarDesdeArchivo

Devuelve los datos del archivo de entrada en un array

Has 29 line(s). Calls functions:

asignarDesdeArchivo

```
|-- avanzarAlgoritmo
```

```
|-- centrarEnPantalla
```

```
|-- elegirTipoDeEntrada
```

```
|   |-- asignarColores
```

```
|   |-- asignarManual
```

```
|   |   |-- asignarColores
```

```
|   |   |-- ordenarArray
```

```
|   |-- asignarValoresAleatorios
```

```
|   |   |-- centrarEnPantalla
```

```
|   |-- centrarEnPantalla
```

```
|   |-- colocarNombreAProcesos
```

```
|   |-- eliminarProcesosNoValidos
```

```
|   |-- imprimirAyuda
```

```
|   |   |-- centrarEnPantalla
```

```
|   |   |-- sacarHaciaArchivo
```

```
|   |-- sacarHaciaArchivo
```

```
`-- sacarHaciaArchivo
```

Uses feature(s): *read*

Called by:

elegirTipoDeEntrada

asignarEstadosSegunInstante

Asigna los estados segun avanza el algoritmo

Has 51 line(s). Calls functions:

```

asignarEstadosSegunInstante
`-- calcularMemoriaRestante

```

Not called by script or any function (may be e.g. `command_not_found_handle` or called indirectly in other way).**asignarEstiloGeneral**

Almacen de los estilos de las tablas con sus códigos ASCII

@param Numero de estilo

Has 22 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. `command_not_found_handle` or called indirectly in other way).**asignarManual**

Asigna valores en el array de forma manual

Has 117 line(s). Calls functions:

```

asignarManual
|-- asignarColores
`-- ordenarArray

```

Called by:

elegirTipoDeEntrada

asignarValoresAleatorios

Crea un array con valores aleatorio para fase desarrollo o entrada de datos automatica, (ni manual ni por archivo)

@param Numero de filas a generar de manera aleatorio (num. procesos)

Has 29 line(s). Calls functions:

```

asignarValoresAleatorios
`-- centrarEnPantalla

```

Called by:

elegirTipoDeEntrada

avanzarAlgoritmo

Funcion basica de avance de algoritmo

Has 2 line(s). Doesn't call other functions.

Uses feature(s): *read*

Called by:

asignarDesdeArchivo

calcularCambiosCPU

Calcula la linea de cpu hasta el momento

Has 21 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. command_not_found_handle or called indirectly in other way).

calcularCambiosMemoria

3. ALGORITMO

Calcula los cambios en memoria para no hacerlo en la misma funcion de impresion

Has 46 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. command_not_found_handle or called indirectly in other way).

calcularLongitud

Devuelve la longitud del string, contando los patrones de colores.

@param String del que queremos calcular la longitud

Has 1 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. command_not_found_handle or called indirectly in other way).

calcularMemoriaRestante

Función que calcula la memoria restante

Has 7 line(s). Doesn't call other functions.

Called by:

asignarEstadosSegunInstante

cc

Devuelve la expresión completa de color, pasándole los parámetros que queremos en orden

@param tipoEspecial (Negrita=Neg, Subrayado=Sub, Normal=Nor, Parpadeo=Par)

@param (valor random, default, error, acierto, fg aleatorio sobre bg negro o lista de colores en orden)

Has 73 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. command_not_found_handle or called indirectly in other way).

centrarEnPantalla

Centra en pantalla el valor pasado, si es un string, divide por saltos de linea y coloca cada linea en el centro

@param String a centrar

@param Si se quiere un espacio al final

Has 41 line(s). Doesn't call other functions.

Called by:

asignarDesdeArchivo

asignarValoresAleatorios

elegirTipoDeEntrada

imprimirAyuda

colocarNombreAProcesos

Funcion que pone los nombres a los procesos con el estándar pedido

Has 7 line(s). Doesn't call other functions.

Called by:

elegirTipoDeEntrada

comprobarProcesosEjecutando

Comprueba los procesos que se están ejecutando

Has 5 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. command_not_found_handle or called indirectly in other way).

copiarArray

Hace una copia del array antes de cambiarlo en la ejecución

Has 5 line(s). Doesn't call other functions.

Called by:

imprimirTabla

elegirTipoDeEntrada

Funcion para elegir el tipo de entrada de datos

@param archivo externo para la opcion de archivo

Has 65 line(s). Calls functions:

elegirTipoDeEntrada

|-- asignarColores

|-- asignarDesdeArchivo

| |-- avanzarAlgoritmo

| |-- centrarEnPantalla

| `-- sacarHaciaArchivo

|-- asignarManual

| |-- asignarColores

| `-- ordenarArray

|-- asignarValoresAleatorios

| `-- centrarEnPantalla

|-- centrarEnPantalla

|-- colocarNombreAProcesos

```

|-- eliminarProcesosNoValidos
|-- imprimirAyuda
|   |-- centrarEnPantalla
|   `-- sacarHaciaArchivo
`-- sacarHaciaArchivo

```

Called by:

asignarDesdeArchivo

imprimirAyuda

eliminarProcesosNoValidos

Funcion que elimina las lineas de datos no válidas.

Estas son las que tienen procesos cuyo tamaño es mayor a la memoria.

Has 34 line(s). Doesn't call other functions.

Called by:

elegirTipoDeEntrada

extraerDeConfig

Devuelve los datos extraidos del archivo de configuracion

@param parametro a leer del config

Has 47 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. command_not_found_handle or called indirectly in other way).

fc

Finaliza el uso de colores

Has 1 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. command_not_found_handle or called indirectly in other way).

imprimirAyuda

Ayuda del algoritmo

Has 23 line(s). Calls functions:

imprimirAyuda

```

|-- centrarEnPantalla
|-- elegirTipoDeEntrada
|   |-- asignarColores
|   |-- asignarDesdeArchivo
|   |   |-- avanzarAlgoritmo
|   |   |-- centrarEnPantalla
|   |   `-- sacarHaciaArchivo
|   |-- asignarManual
|   `-- asignarColores

```

```

| | `-- ordenarArray
| | -- asignarValoresAleatorios
| | `-- centrarEnPantalla
| | -- centrarEnPantalla
| | -- colocarNombreAProcesos
| | -- eliminarProcesosNoValidos
| | `-- sacarHaciaArchivo
|-- sacarHaciaArchivo

```

Uses feature(s): *read*

Called by:

`elegirTipoDeEntrada`

imprimirCuadro

Imprime la introduccion del programa

@param Ancho del cuadro

@param Color

@param Array del contenido del cuadro

@param Tipo de texto

Has 64 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. `command_not_found_handle` or called indirectly in other way).

imprimirLineaProcesos

Imprime la linea de procesos de CPU

@param Instante actual

Has 82 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. `command_not_found_handle` or called indirectly in other way).

imprimirMemoria

Imprime el uso de la memoria según los procesos en ella

Has 76 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. `command_not_found_handle` or called indirectly in other way).

imprimirTabla

Imprime una tabla según el tamaño del array de datos

@param numeroFilasImprimir

@param numeroColumnasImprimir

@param numeroDeColumnaDelQueEmpezamos

Has 199 line(s). Calls functions:

`imprimirTabla`

``-- copiarArray`

Not called by script or any function (may be e.g. `command_not_found_handle` or called indirectly in other way).

main

```
##### Main
Main, eje central del algoritmo, única llamada en cuerpo
-----
```

Has 271 line(s). Doesn't call other functions.

Called by:

Script-Body

ordenarArray

```
Ordena el array según tiempo de llegada para mostrar la tabla.
@param col Movemos también los colores.
-----
```

Has 47 line(s). Doesn't call other functions.

Called by:

asignarManual

procesosHanTerminado

```
Comprueba si el programa ha acabado
-----
```

Has 9 line(s). Doesn't call other functions.

Not called by script or any function (may be e.g. `command_not_found_handle` or called indirectly in other way).

recibirEntrada

```
Funcion tipo de entrada de datos comun a todas las peticiones del programa
-----
```

Has 5 line(s). Doesn't call other functions.

Uses feature(s): *read*

Not called by script or any function (may be e.g. `command_not_found_handle` or called indirectly in other way).

sacarHaciaArchivo

```
Saca la información del comando que acompaña
@param "-a" para append
-----
```

Has 8 line(s). Doesn't call other functions.

Called by:

asignarDesdeArchivo

elegirTipoDeEntrada

imprimirAyuda

COMO GENERAR LA DOCUMENTACIÓN

Para ver como generar la documentación puedes ejecutar el script “documentar.sh” dentro de la carpeta Doc. Misma carpeta de este archivo. Cuando lo ejecutes te preguntará si quieres ver la ayuda de instalación. Una vez tengas todos los requisitos, pueden seguir las instrucciones en orden para ir generando los documentos por tu cuenta o puedes ejecutar el script y el mismo te generará una carpeta con la documentación del script pasado por parámetro, o del script por defeco si ningún parámetro es pasado. Por otro lado, podrás personalizar la cabecera de la documentación dentro de la carpeta info.

HERRAMIENTAS

CONTROL DE VERSIONES:

- GIT

IDE:

- VISUAL STUDIO CODE

DEBUGGER:

- SHELLCHECK

SISTEMA OPERATIVO:

- ARCOLINUX (ARCHLINUX)

FORMATTER:

- PRETTIER FORMATER