Prácticas de Sistemas Operativos

Informe de prácticas

Práctica de Control.

Código de grupo: 1-Jueves

Alumnos:

1.-José Luis Garrido Labrador

2.-Luis Pedrosa Ruiz

Indice

1 Presentación del tema

El algoritmo escogido fue Round Robin con gestión de memoria de particiones dinámicas ajustada al mejor, contigua y reubicale. La memoria al ser dinámica era inutil el ajuste al mejor al no existir particiones predefinidas por lo que a la hora de implementar el sistema de gestión de memoria fue orientado a buscar el menor hueco posible para evitar una excesiva fragmentación por lo que seguimos con el propio trabajo a pesar de no seguir exactamente el enunciado que tiene un parámetro incorrecto.

A la hora de implementar un script que tuviese una sinergia lo más optima entre la gestión de procesos y gestión de memoria hemos supuestos varias premisas

- La memoria del sistema operativo está bloqueada en una partición no contemplada por lo que la memoria en la que se realiza la práctica solo contempla la memoria de la que dispone el usuario.
- No se han conteplado tiempos de escritura y borrado de memoria así como cambios de contexto.
- Hemos tratado el problema como una simulación por lo que partes de su código tienen más una finalidad de cara al usuario que de cara al funcionamiento interno provocando que algunos fragmentos no tengan sentido en una implementación real como es el caso de la lista de procesos o los tres vectores de memoria

2 Documentación de los algoritmos

El script fue realizado completamente desde cero. Esta decisión fue tomada debido a que el script contenía también implementado el algoritmo Round Robin Virtual y para aumentar la comprensión de la gestión Round Robin de manera particular mediante una implementación directa. Además la generación de la lista era incorrecta y la documentación interna como externa insuficiente para su comprensión.

El código se divide de manera general en cuatro partes, la captura de datos, vectores de información, la gestión de memoria y la gestión de procesos. Se ha utilizado un paradigma de programación modular y por tanto existen varias funciones tanto para la gestión de la memoria como para control de errores.

 CC-BY-SA / GPLv3

2.a Captura de datos

En está parte el usuario introduce los datos de cantidad de procesos, quantum, cantidad de memoria y los datos de nombre de proceso, ráfaga de CPU, momento de llegada y cantidad de memoria necesaria. Estos datos anteriores solamente si se elije la introducción manual, existe la posibilidad de hacer una introducción automática desde un fichero en el que hay 10 procesos introducidos. Todos los datos de los procesos se guardan en vectores de información de dimensión variable según la cantidad de procesos introducidos y su posición va indicada por el orden en el que el usuario los ha añadido. Esto tiene una finalidad puramente orientativa a la hora de que el programador sepa en cada momento que proceso es sin la necesidad de saber que datos ha introducido.

Algunos datos tienen valores restringidos como que cada proceso tenga un nombre distinto o que la ráfaga o la memoria de un proceso no pueden ser menores o iguales que 0. En el primer caso está orientado únicamete para una mejor comprensión de los resultados pero en los otros dos para evitar situaciones imposibles. Se recomienda también que los procesos tengan nombre cortos para una mejor visualización de las gráficas y de la memoria ya que los nombres tomarán un papel crucial a la hora de visualizar la información pero no tienen un papel importante de cara a la programación.

2.b Vectores de información

La mayoría de los datos que usamos a la hora de programar los dos algoritmos fueron almacenados en vectores con unas características muy específicas, vamos a explicarlos uno por uno para una mejor comprensión, además en el propio código existe una propia documentación interna.

- mem: el vector memoria con tantas casillas como megas de memoria introduzca el usuario, este vector tendrá el papel de memoria pero a nivel informativo, no real
- mem_dir: este vector es la memoria propiamente dicha, es igual a mem pero a diferencia de este la información que almacena mem_dir es el valor identificativo de cada proceso no su nombre y nunca se visualiza, es puramente interno para el funcionamiento correcto del algoritmo. Su valor es -1 si se está libre
- proc_name: vector con los nombres de cada proceso
- proc_arr: vector con el turno de llegada de cada proceso
- proc_exe: tiempo de ejecución restante de cada proceso

- proc_mem: memoria necesaria de cada proceso
- proc_order: este vector es el orden en el que los procesos llegan, sus valores se le asignan mediante un algoritmo de ordenación implementado por nosotros, está enfocado a la simulación pero no tiene implementación real
- list: este es el vector que tiene la lista de ejecución de los procesos, la posición 0 es la que contiene el proceso que se ha de ejecutar y funciona con dos dimensiones, una real (su tamaño) y lógica (las posiciones que realmtene contienen información útil), esto es gracias al funcionamiento de los vectores en bash y que estos son dinámicos, el tamaño lógico del vector lo da la variable listTam que comienza en 0 y va creciendo cuando llegan procesos a la memoria y decreciendo cuando se liberan.

  Como el funcionamiento de la lista requiere un estudio más individual vamos a explicarlo detenidamente. Su funcionamiento lo hacemos mediante la función lista:

```
function lista {
	local cont
	local aux
	local fin
	proceso=${list[0]}
	if [ ${proc_exe[$proceso]} -eq 0 ];then
		let listTam--
		for (( cont=0;cont<$listTam;cont++ ))
		do
			list[$cont]=${list[$(expr $cont + 1)]}
		done
	else
		aux=${list[0]}
		fin=$(expr $listTam - 1)
		for (( cont=0;cont<$listTam;cont++ ))
		do
			case $cont in
				$fin)
					list[$cont]=$aux
					;;
				*)
					list[$cont]=${list[$(expr $cont + 1)]}
					;;
			esac
		done
	fi
}
```

Siendo breves cuando un proceso termina desplazamos todas los procesos hacia delante (el valor de listTam se reduce previamente) y si no ha terminado se guarda en una variable auxiliar y se pone al final de la lista.

Su valor aumenta cuando un proceso entra en memoria:

```
#Metemos el procenso en la cola de ejecución
list[$listTam]=$zed
let listTam++
let total++
```

La variable total representa la cantidad de procesos que han entrado en memoria

- proc_memI: palabra de memoria inicial del proceso
- proc_memF: palabra de memoria final del proceso
- proc_waitA: tiempo que el proceso espera (más el tiempo desde 0)
- proc_waitR: tiempo real de espera (valor desde el que se hace la media)
- proc_ret: tiempo de salida del proceso
- proc_ret: tiempo de retorno, (tiempo desde que entró hasta que salió)
- partition: cantidad de memoria libre contigua, en cada casilla se guarda la cantidad de Megabyte libres
- part_init: este vector almacena el punto inicial donde empieza un conjunto de espacio vacío
- mem_print: vector clon de la memoria para almacenar la misma sin valores de colores con el fin de direccionar correctamente a un fichero

2.c Gestión de procesos

Al ser un algoritmo Round Robin hay un cambio de contexto cada vez que se agota el quantum, un proceso se bloquea o este se termina. En nuestra implementación no tenemos entrada y salida ni procesos de varios hilos por lo que los procesos nunca padecen bloqueos y por tanto el caso no ha sido contemplado. El gestor de procesos ejecuta un proceso mientras el quantum disponible no sea 0 o el programa no tenga una ejecución igual a cero, es decir, haya acabado. Solo ejecuta los procesos que están en en la lista y como se mencionó antes entran en ella cuando entran en memoria.

Cada vez que un proceso termina (o agota su quantum) la lista cambia y hay un cambio de contexto, si el tamaño lógico de la lista fuera 0 pero el total de procesos que se introdujeron en memoria no es igual a la cantidad de procesos introducidos por el usuario habrá un avanze rápido hacie el siguiente tiempo en el que entran más procesos.

Los procesos entran en memoria según su momento de llegada y la posición de la cola, esto se determina gracias a la función que ordena en un vector auxiliar los procesos según su llegada.

```
#Algoritmo de orden
for (( i=$(expr $proc-1); i>=0; i-- ))
do
        max=0
        for (( j=0; j<$proc; j++ ))
        do
                for (( z=$i, coin=0; z<=$(expr $proc-1); z++ ))
                do
                        if [ $j -eq "${proc_order[$z]}" ];then
                                coin=1
                        fi
                done
        if [ $coin -eq 0 ];then
                if [ "${proc_arr[$j]}" -ge $max ];then
                        aux=$j
                        max="${proc_arr[$j]}"
                fi
        fi
        done
        proc_order[$i]=$aux
done
```

Está adjuntado un diagrama de flujo de esta función.

2.d Gestión de memoria

Cada vez que un proceso nuevo llega se le asigna memoria, si necesita más memoria de la disponible actualmente quedará en espera hasta que algún proceso libere su parte y si lo que pasa es que hay suficiente memoria disponible pero no hay suficiente memoria contigua se reubica toda la tabla de particiones desplazandose hacia la izquierda mediante esta función:

```
function reubicar {
        before=0
        local aux
        local aux2=0
        local ret
        for (( w=0; w<$mem_total; w++))
        do
                if [ ${mem_dir[$w]} -eq -1 -a $before -eq 0 ];then
                                before=1
                                aux_init=$w
                elif [ $before -eq 1 -a ${mem_dir[$w]} -ne -1 ];then
                                aux=${mem_dir[$w]}
                                aux2=1
                                DesOcuMem ${proc_memI[$aux]} $
{proc_memF[$aux]}
                                proc_memI[$aux]=$aux_init
```

```
                                    let proc_memF[$aux]=proc_memI[$aux]
+proc_mem[$aux]
                                    let proc_memF[$aux]=proc_memF[$aux]-1
                                    OcuMem ${proc_name[$aux]} $
{proc_memI[$aux]} ${proc_memF[$aux]} $aux
                                    before=0
                                    w=proc_memF[$aux]
                fi
        done
        if [ $aux2 -eq 1 ];then
                echo -e "${inverted}La memoria se ha reubicado${NC}"
                echo "La memoria se ha reubicado" >> informe.txt
        fi
        let ret=${proc_memF[$aux]}+1
        return $ret
}
```

Una explicación breve del funcionamiento sería: primero determinamos si la palabra de memoria actual está vacía y no lo estaba antes (before -eq 0), si ese es el caso significa que estamos en el principio de una partición vacía y esta posición se almacenará como el punto principal donde comenzará el primer proceso encontrado en el recorrido, cuando avanzando en la memoria encontremos una palabra no vacía entonces asignaremos la memoria del proceso encontrado desde la posición inicial hasta la palabra determinada por su tamaño, el recorrido de la memoria se contemplará ahora desde la siguiente palabra a la última asignada.

El funcionamiento de este módulo no es aplicable a una implementación real de reubicación de memoria pero si es válido para la resolución de ejercicios y simulación.

Otro detalle importante es que aunque las particiones dinámicas no tienes ajustes nosotros a la hora de plantear el problema entendimos el ajuste al mejor como la posición de la memoria en el hueco vacío más pequeño posible en el que cupiese el proceso implementando la función *PartFree* para buscar todos los huecos vacíos de la memoria por lo que a pesar de que las particiones dinámicas no tienen ajuste nosotros no cambiamos de trabajo.

```
function PartFree {
      for (( y=0; y<$MAX ; y++ ))
      do
            partition[$y]=0
      done
      part=0
      h=0
      for (( y=0; y<${#mem[@]}; y++ ))
      do
            value=${mem[$y]}
            if [ $value == $Li ];then
                  if [ $h -eq 0 ];then
                        part_init[$part]=$y
                        h=1
                  fi
                  let partition[$part]=partition[$part]+1
            else
                  if [ $h -eq 1 ];then
                        h=0
                        let part=part+1
                  fi
            fi
      done
}
```

El funcionamiento de PartFree es determinar los huecos libres de la memoria para esto vemos lo que hay en value, si es la variable Li (libre) entonces comprubo que sea la primera vez que he entrado entrado en la partición libre, si es cierto (h = 0) entonces guardamos en part_init la posición donde entra, y luego independientemente del valor de h aumentamos el valor de partition, esto luego de estudiará para ver cual es la más pequeña. En el caso de que no sea libre miramos a ver si h es 1 y en caso afirmativo la ponemos a 0 y aumentamos part

Cuando un proceso no puede entrar en el momento dado todos los demás se quedan en cola.

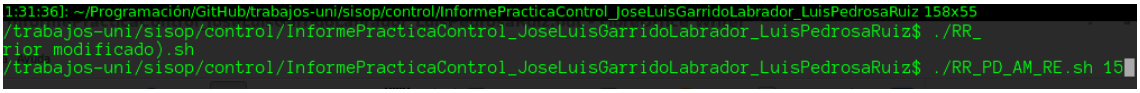
2.e Otros detalle importantes

A la hora de mostrar la información real implementamos dos maneras, una automática y otra manual. La manera manual es dependiente de que el usuario pulse intro cada vez que un turno del reloj muestra toda la información, en la versión automática hay un delay de 5 segundos (introducimos tambien un modo completamente automático que sin esperar manda todo al fichero de salida. Aunque en un tiempo no

pase nada se mostrará la información del mismo con los datos de los procesos. Los únicos casos en los que podrá haber saltos del reloj será o bien cuando en un periodo de tiempo no haya ningún proceso en cola (no será un salto propiamente dicho, será una carrera casi impersectible) y por la información dada por el usuario sabémos que llegará o el primer proceso en entrar no empieza en el turno 0.

Los tiempos de espera contemplados fueron dos, el tiempo de espera acumulado que cuenta desde el tiempo 0 todos los turnos en los que el proceso no se estuvo ejecutando (y no había retornado), el tiempo de espera real es el mismo tiempo pero desde que el proceso llegó (no que entró en memoria si no simplemente estaba esperando). En cuanto a la salida tenemos dos valores igualmente, el valor "Salida" que indica el tiempo en el que el proceso retornó mientras que el retorno real es el valor en el que el proceso ha estado desde su entrada hasta su salida. Los valores medios calculados son a partir de ambos valores reales.

Se ha añadido un modo para truncar la memoria en bloques separados por lineas evitando así cortes al final de la pantalla, por defecto esta opción está quitada pero si se quiere activar solo hace falta poner el n.º de bloques que aparecerán en cada linea como parámetro.



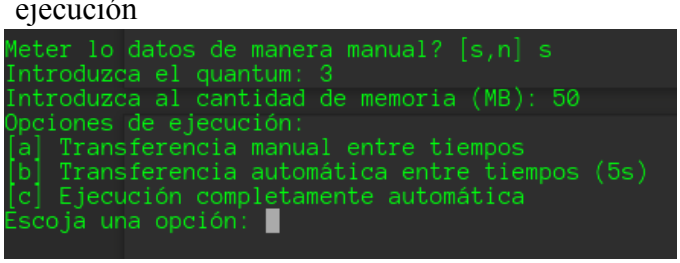
3 Archivos e interpretación de los datos

Los datos de entrada pueden ser manuales o no, en este caso será con un fichero que funciona de la siguiente manera:

```
100
3
p1;3;2;25
p2;0;2;15
p3;1;6;25
p4;2;8;65
p5;3;4;20
p6;0;4;35
p7;3;4;5
p8;4;5;22
p9;6;2;9
p10;4;1;3
```

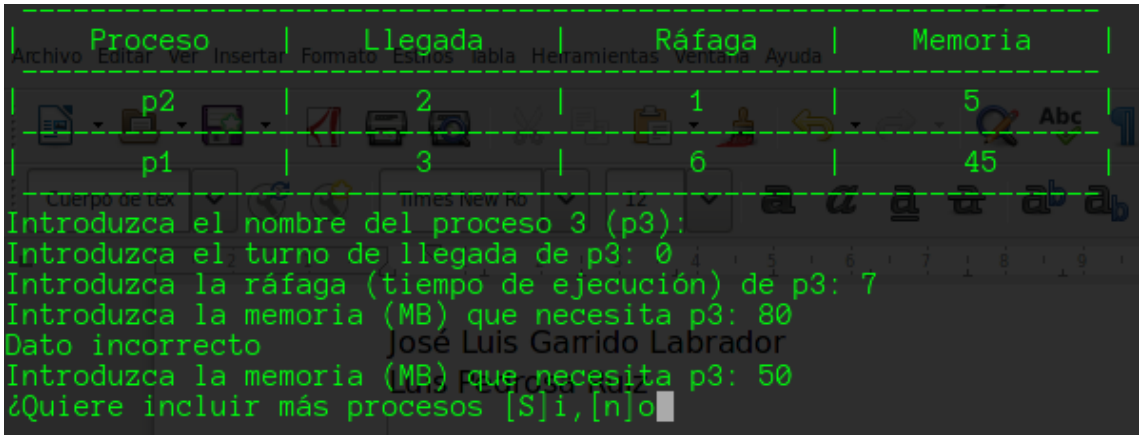
El primer valor determina la memoria, el segundo el quantum de ejecución y los demás los datos de cada proceso siendo en orden el nombre; e tiempo de llegada, el tiempo de ejecución y la memoria. Es importante que estos datos pueston con anterioridad sean correctos ya que se presupone que son correctos. Si hubiera alguna carencia en el fichero se rescribirá con los valores de default.txt con permisos 444. Si el

usuario introduce los datos manualmente estos datos serán los datos de la proxima ejecución



La captura de los datos se hace con la función Fichero

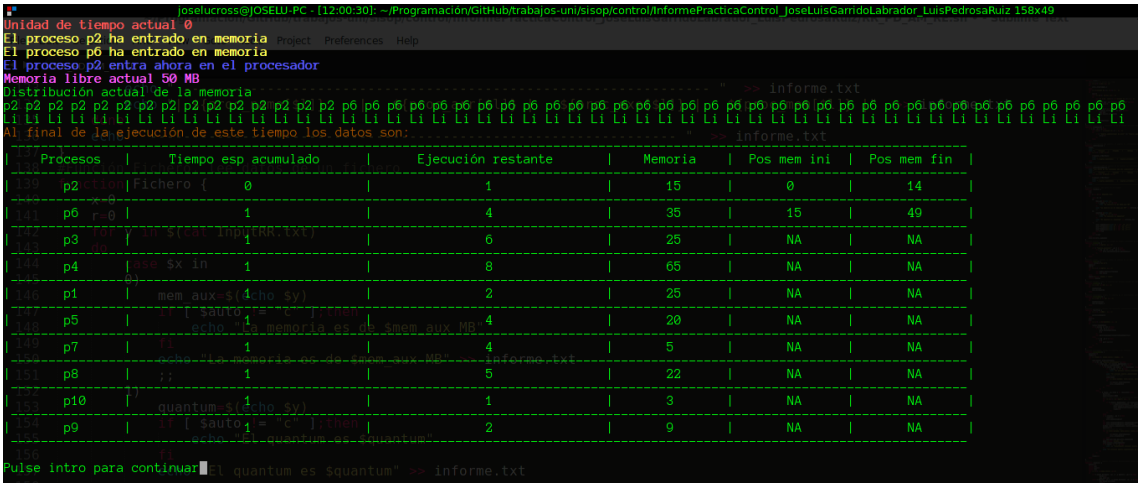
```
function Fichero {
	x=0
	r=0
	for y in $(cat InputRR.txt)
	do
		case $x in
		0)
			mem_aux=$(echo $y)
			if [ $auto != "c" ];then
				echo "La memoria es de $mem_aux MB"
			fi
			echo "La memoria es de $mem_aux MB" >> informe.txt
			;;
		1)
			quantum=$(echo $y)
			if [ $auto != "c" ];then
				echo "El quantum es $quantum"
			fi
			echo "El quantum es $quantum" >> informe.txt
			;;
		*)
			proc_name[$r]=$(echo $y | cut -f1 -d";")
			proc_arr[$r]=$(echo $y | cut -f2 -d";")
			proc_exe[$r]=$(echo $y | cut -f3 -d";")
			proc_mem[$r]=$(echo $y | cut -f4 -d";")
			let r=r+1
		esac
		let x=x+1
	done
	proc=${#proc_name[@]}
}
```



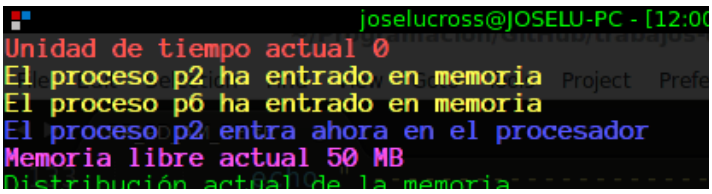
Los datos se introducen de manera individual hasta que se diga que no se quieren introducir más procesos. Como se puede observar si la memoria que se introduce es mayor a la total devuelve error y pide el dato nuevamente.

Al final de la introducción de datos se nos informa de los datos introducidos ordenados por tiempo de llegada

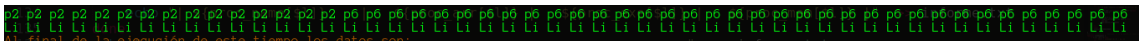
Esta es la muestra de un tiempo dado, vamos a explicarlo parte a parte.



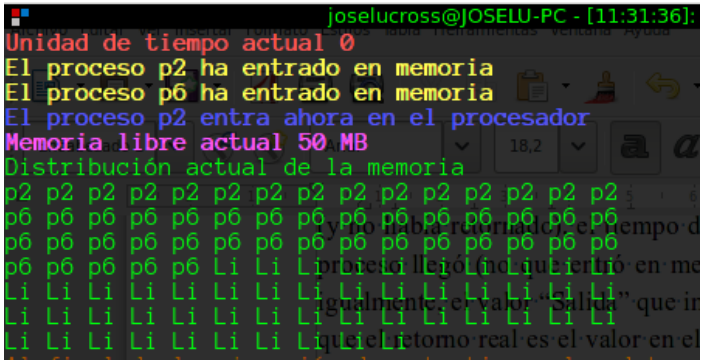
En rojo tenemos el tiempo actual (se ha cambiado a verde pero el resultado es el mismo), al principio de ese tiempo muestra los procesos que han entrado en memoria.



Posteriormente determina que proceso entra en el procesador para ser ejecutado. Al final nos muestra la memoria libre y la distribución de la memoria.



Si se trunca la memoria con 15 el resultado sería mas o menos este

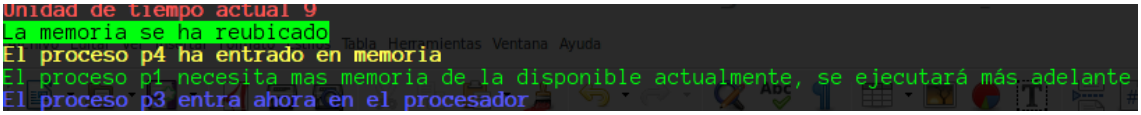


Posteriormente tenemos un resumen de todos los procesos

Procesos	Tiempo esp acumulado	Ejecución restante	Memoria	Pos mem ini	Pos mem fin
p2	0	1	15	0	14
p6	1	4	35	15	49
p3	1	6	25	NA	NA
p4	1	8	65	NA	NA
p1	1	2	25	NA	NA
p5	1	4	20	NA	NA
p7	1	4	5	NA	NA
p8	1	5	22	NA	NA
p10	1	1	3	NA	NA
p9	1	2	9	NA	NA

Aquí podemos ver el tiempo de espera que lleva acumulado y el tiempo restante, también aparece la memoria y sus respectivas posiciones. Además si la memoria no ha sido asignada al proceso se determina como “NA” (No asignado) y si el proceso ha terminado en *Ejecución restante*, en *Pos mem ini* y en *Pos mem fin* aparecerá como “END”. Además los valores de estos dos ultimos campos pueden cambiar si ha habido reubicación.

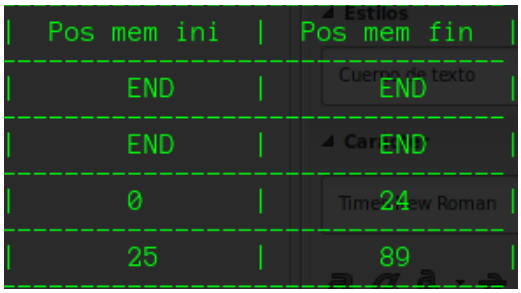
Cuando la memoria se reubica se muestra al principio del tiempo, y cuando un proceso no puede entrar por falta de memoria actualmente se muestra el mensaje de aviso.



Memoria en el turno anterior a la reubicación:

Pos mem ini	Pos mem fin
END	END
END	END
50	74
NA	NA
NA	NA

Y despues de esta



Pos mem ini	Pos mem fin
END	END
END	END
0	24
25	89

Cuando un proceso termina su ejecución o agota su quantum, es decir, en el turno anterior a un cambio de contexto aparece el siguiente mensaje:

En el subrayado azul aparecen los siguientes datos: nombre, el momento de entrada en la CPU y las ráfagas restantes.

Para finalizar con esta sección en el fichero de salida *informe.txt* está volcada toda la información de la misma manera que se muestra por pantalla (a excepción de los colores). Este fichero se sobrescribe cada vez que se ejecuta el script por lo que es sumamente importante cambiarle de nombre o moverlo a otro directorio si no se quiere perder.

4 Análisis de scripts y mejoras

El script RR_RRV.sh no funcionaba correctamente dado que no generaba correctamente la lista de ejecución además de que el código carecia de la suficiente documentación, al prever que la interpretación del propio código y los posteriores arreglos fuesen a ocupar una gran cantidad de tiempo optamos por hacer una implementación personal ajena al RR_RRV.sh, a parte del script principal con el trabajo completo también adjuntamos la gestión de procesos RR de manera independiente en el programa RR.sh. Aunque no utilizamos el código cedido si añadimos que los procesos se añadiesen en el acto sin una previa asignación al igual que en nuestra implementación.

A continuación el ejemplo con el que se observa el error en la lista de procesos

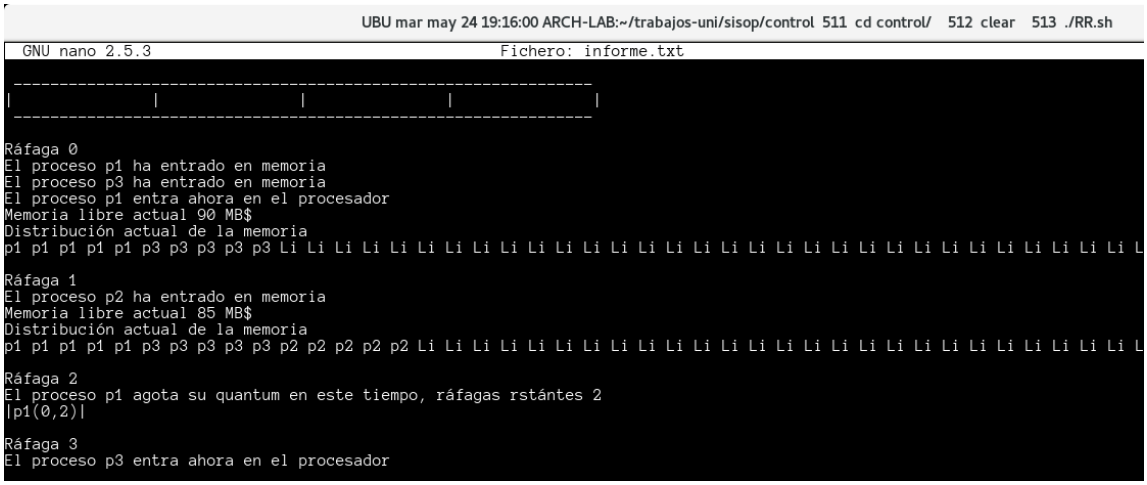
Los datos son

```
>> Quantum de tiempo: 3
>> 3 procesos.
>> Procesos y sus datos:
-------------------------
  PRO| LLEGADA | RAFAGA
```

```
--------------------------
  1    | 0        | 5
--------------------------
  2    | 1        | 7
--------------------------
  3    | 0        | 3
--------------------------
```

Y el resultado era este

Como se puede ver el proceso 2 se ejecuta posteriormente a 1 aunque 3 llegase despues de 1 y antes que 2, por lo que le tocaría a él ejecutarse. En nuestra implementación los mismos datos se ejecutan de esta manera.



Como puede observarse (y se puede replicar en caso de que la calidad de la imágen sea demadiado precaria) el proceso 3 que llega antes que 2 y despues que 1 se ejecuta entre estos.

5 Ejemplos a mano

5.a Ejercicio 1

Para el primer ejemplo usaremos 5 procesos y una memoria de 250 megabytes y un quantum de 5.

Nombre	Llegada	Ráfaga	Memoria
1	3	13	90
2	0	17	125
3	2	8	23
4	0	2	15
5	2	15	5

Tiempo 0

El proceso 2 entrará en memoria y ocupara desde la palabra 0 a la palabra 124, el proceso 4 entrará en memoria de manera posterior y entrará desde la posición 125 hasta la 129. El proceso 2 entrará en la CPU ahora hasta el turno 4 donde agotará su quantum y le quedarán por ejecutarse 12 ráfagas

Tiempo 2

En este tiempo el proceso 2 sigue ejecutandose y entra en memoria el proceso 3 en las palabras 130 a 152 y el proceso 5 desde la 153 a la 167.

Tiempo 3

El proceso 1 llega pero necesita ocupar 90 espacios en la memoria y solo se disponen de 87 megabytes, tendrá que esperar

Tiempo 4

El proceso 2 abandona la CPU al terminar este tiempo quedándole 12 ráfagas aún. Todos los demás procesos a excepción del 2 han acumulado 5 tiempos de espera

Tiempo 5

El proceso 4 entra en la CPU

Tiempo 6

El proceso 4 termina su ejecución al finalizar este turno y por tanto se libera su memoria, ahora hay 102 megabyte libres en dos particiones distintas. Todos los proceso a excepción de 4 sumań 2 tiempos en su espera acumulada, cuatro termina con una espera de 5 (real y acumulada) y un retorno (real) en 6

Tiempo 7

El proceo 1 ya puede entrar en memoria pero para eso es necesario reubicar la memoria, ahora el proceso 3 está en las palabras 125 a 147, el proceso 5 en las palabras 148 a 152. El proceso 1 estaría en 153 hasta 242. El proceso 3 entra ahora

Tiempo 11

El proceso 3 ya sale del procesador por agotar su quantum, le quedan 3 ráfagas más, el tiempo de espera de los demás procesos aun no terminados aumenta en 5

Tiempo 12

El proceso 5 entra en el procesador

Tiempo 16

El proceso 5 sale del procesador y le quedan aún 10 ráfagas más, el resto de procesos aumenta su espera en 5

Tiempo 17

El proceso 1 entra en el procesador

Tiempo 21

El proceso 1 sale del procesador y le quedan 8 ráfagas, los demás procesos suman 5 en su tiempo de espera acumulado

Tiempo 22

El proceso 2 entra en el procesador.

Tiempo 26

El proceso 2 abandona la CPU con una ejecución restante de 7 ráfagas. Los demás procesos suman 5 a su espera.

Tiempo 27

El proceso 3 entra en la CPU

Tiempo 29

El proceso 3 abandona la CPU y termina su ejecución, su memoria asignada se libera. Los demás procesos aumentan su espera en 3. Al salir tiene una espera acumulada de 22 y real de 20. Su retorno ha sido de 27.

Tiempo 30

El proceso 5 entra en la CPU

Tiempo 34

El proceso 5 abandona la CPU al finalizar este tiempo, aun tiene unas ráfagas pendientes de 5, los demás procesos acumulan 5 de espera.

Tiempo 35

El proceso 1 entra en la CPU

Tiempo 39

El proceso 1 abandona la CPU y solo tiene 3 ráfaga pendientes, los demás procesos aumetan en 5 su espera.

Tiempo 40

El proceso 2 entra en la CPU

Tiempo 44

El proceso 2 abandona la CPU con 2 ráfagas restantes, los demás procesos aumentan en 5 su espera.

Tiempo 45

El proceso 5 entra en la CPU

Tiempo 49

El proceso 5 agota su quantum pero tambíen termina la ejecución, la memoria asignada para él se vacía. Los demás procesos auméntan en 5 su tiempo de espera. Al salir tiene un tiempo de espera acumulado de 35, real de 33 y un retorno de 47.

Tiempo 50

El proceso 1 entra nuevamente en la CPU

Tiempo 52

El proceso 1 termina la ejecución retorna con una espera acumulada de 40, 37 real y retorna en 49. La espera acumulada aumenta en 3 a todos los demás procesos no terminados. r

Tiempo 53

El proceso 2 entra en la CPU

Tiempo 54

El proceso 2 termina su ejecución con una espera acumulada de 38 al igual que la espera real, el retorno es de 54. Al terminar este proceso se termina la ejecución de todos los procesos introducidos.

Los valores medios reales son: Espera=26, Retorno=36

La resolución con el programa de este mismo problema se encuentra en el video adjunto y los datos del informa estan en informe_ejercicio1.txt.

5.b Ejercicio 2

Para este segundo ejercicio vamos a usar los siguientes datos

Nombre	Llegada	Ráfaga	Memoria
p1	3	5	25
p2	0	2	45
p3	1	3	65
p4	5	3	45

El quantum será de 3 y la memoria de 140

Tiempo 0

En este tiempo entra p2, ocupa de la posición 0 a la 44 dejando 95 Megas libres, el finalizar el tiempo actual la ráfaga de p2 es 1

Tiempo 1

El proceso p3 entra ahora en memoria y se coloca desde la 45 a la 109 dejando 30 MB libres. El proceso 2 terminará de ejecutarse ahora, su memoria se liberará dejando libre 75 MB libres en dos particiones de 45 de 0 a 44 y una de 30 de 110 a 139. La espera el proceso 2 al terminar es de 0

Tiempo 2

El proceso p3 empieza a ejecutarse

Tiempo 3

El proceso p1 entra en memoria ocupando de la posicion 110 a la 134 ya que es la partición vacia más pequeña donde cabe.

Tiempo 4

p3 termina de ejecutarse (además agota su quantum), libera la memoria y los demás procesos no terminados aumentan su espera a 3, el proceso 3 tiene un retorno de 3 y una espera real de 1.

Tiempo 5

El proceso p4 entra en memoria ocupando de la posición 0 a 44, el proceso p1 se empieza a ejecutar

Tiempo 7

El proceso p1 agota su quantum, le quedan dos ráfagas de ejecución, la espera acumulada aumenta 3 en los demás procesos no terminados.

Tiempo 8

El proceso p4 entra a ejecutarse

Tiempo 10

El proceso p4 termina de ejecutarse con un retorno de 5 y una espera de 2, la espera del proceso 1 aumenta en 3

Tiempo 11

El proceso p1 entra a ejecutarse

Tiempo 12

El proceso p1 termina de ejecutarse, el retorno es de 9 y la espera real de 4

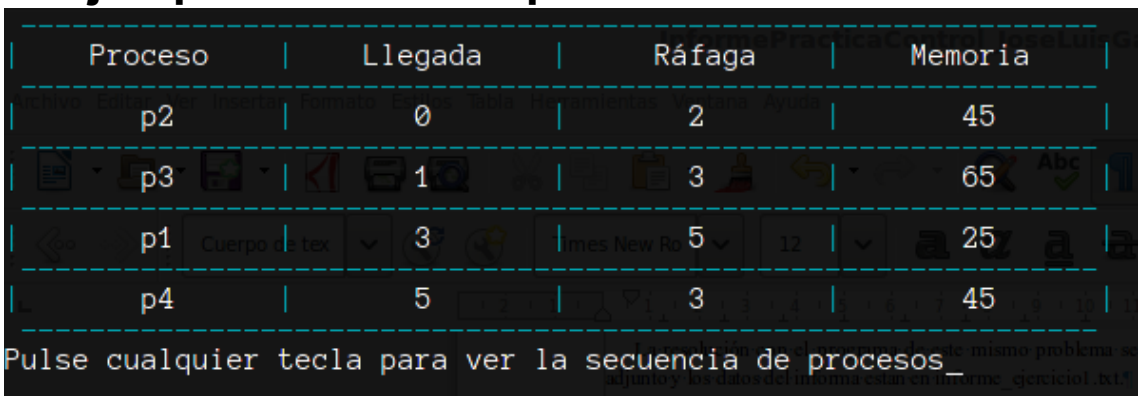
Los valores medios son de retorno:

(9+5+3+1)/4 es aproximadamente 4

La espera media es:

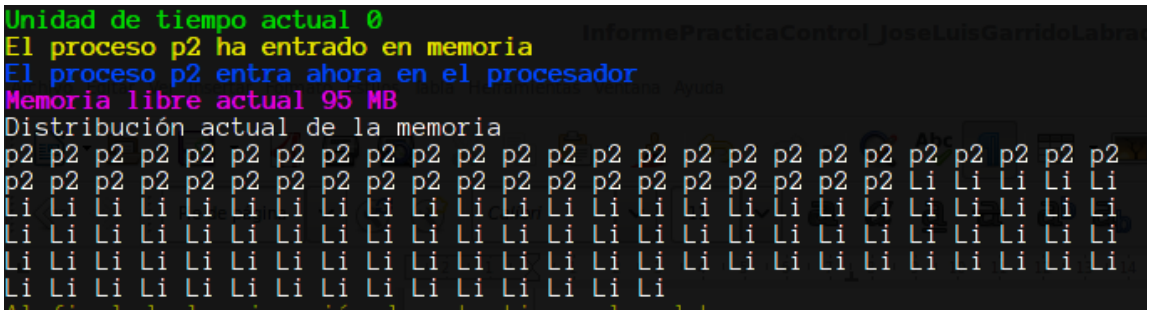
(4+2+0+3)/4 es aproximadamente 2

6 Ejemplos con el script

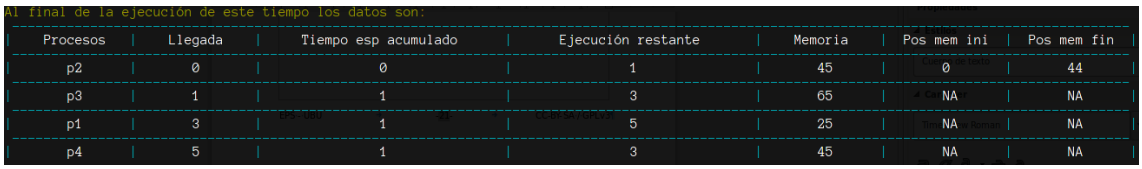


Proceso	Llegada	Ráfaga	Memoria
p2	0	2	45
p3	1	3	65
p1	3	5	25
p4	5	3	45

Pulse cualquier tecla para ver la secuencia de procesos_

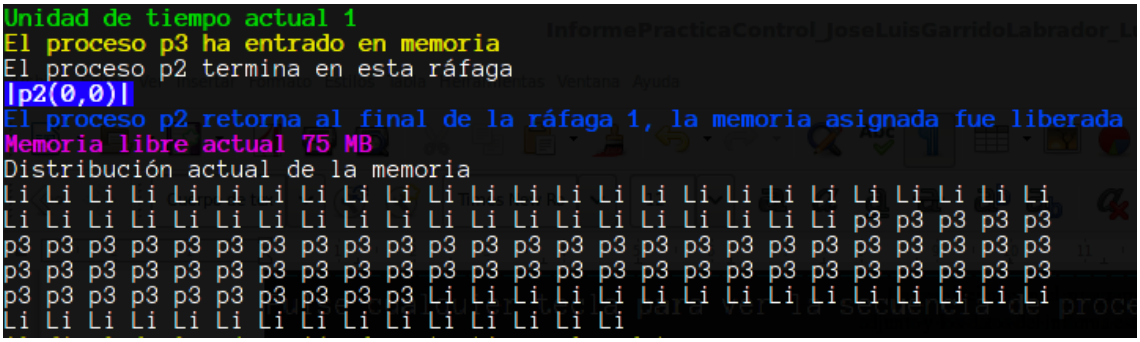


```
Unidad de tiempo actual 0
El proceso p2 ha entrado en memoria
El proceso p2 entra ahora en el procesador
Memoria libre actual 95 MB
Distribución actual de la memoria
p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2
p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 p2 Li Li Li Li Li
Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li
Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li
Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li
Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li
```

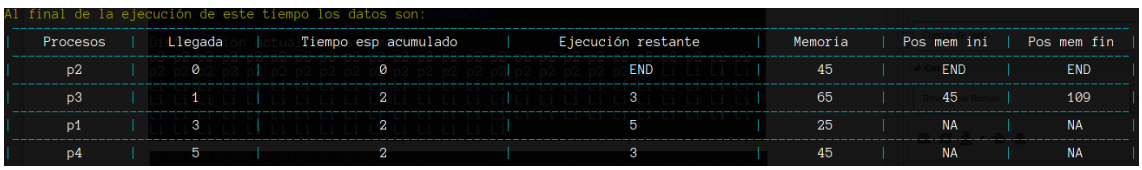


Al final de la ejecución de este tiempo los datos son:

Procesos	Llegada	Tiempo esp acumulado	Ejecución restante	Memoria	Pos mem ini	Pos mem fin
p2	0	0	1	45	0	44
p3	1	1	3	65	NA	NA
p1	3	1	5	25	NA	NA
p4	5	1	3	45	NA	NA

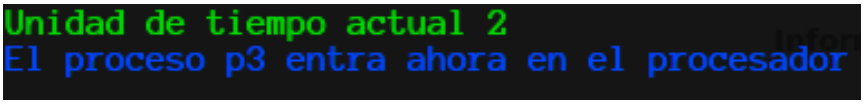


```
Unidad de tiempo actual 1
El proceso p3 ha entrado en memoria
El proceso p2 termina en esta ráfaga
|p2(0,0)|
El proceso p2 retorna al final de la ráfaga 1, la memoria asignada fue liberada
Memoria libre actual 75 MB
Distribución actual de la memoria
Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li
Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li p3 p3 p3 p3 p3
p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3
p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3
p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 p3
p3 p3 p3 p3 p3 p3 p3 p3 p3 p3 Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li
Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li Li
```



Al final de la ejecución de este tiempo los datos son:

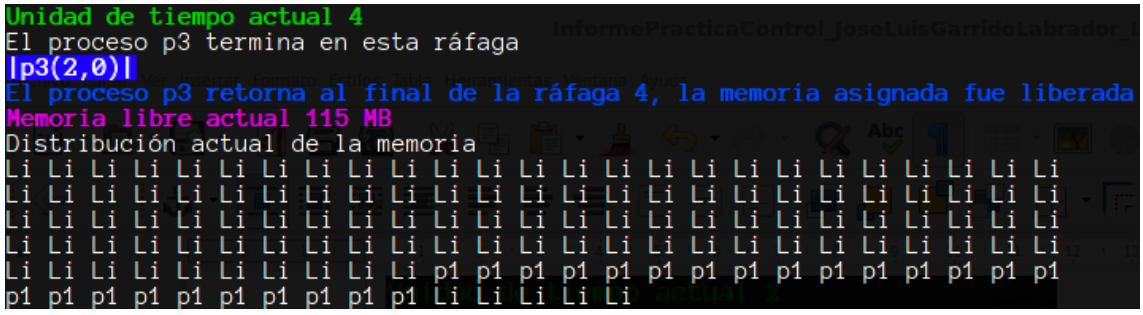
Procesos	Llegada	Tiempo esp acumulado	Ejecución restante	Memoria	Pos mem ini	Pos mem fin
p2	0	0	END	45	END	END
p3	1	2	3	65	45	109
p1	3	2	5	25	NA	NA
p4	5	2	3	45	NA	NA



```
Unidad de tiempo actual 2
El proceso p3 entra ahora en el procesador
```

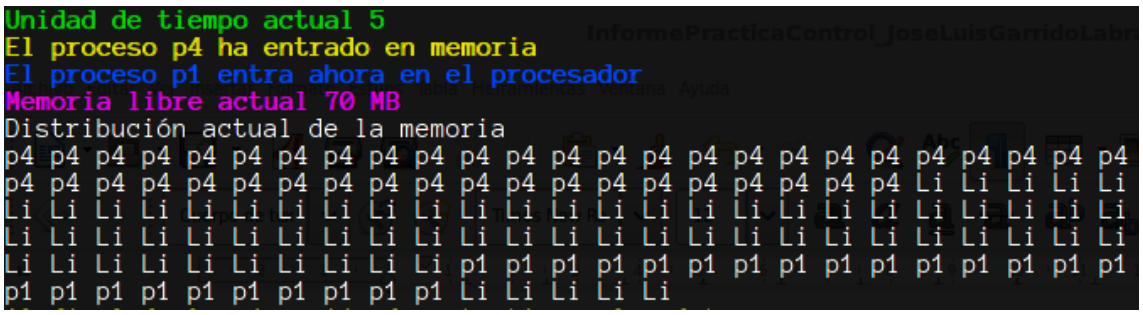
Al final de la ejecución de este tiempo los datos son:

Procesos	Llegada	Tiempo esp acumulado	Ejecución restante	Memoria	Pos mem ini	Pos mem fin
p2	0	0	END	45	END	END
p3	1	2	1	65	45	109
p1	3	4	5	25	110	134
p4	5	4	3	45	NA	NA



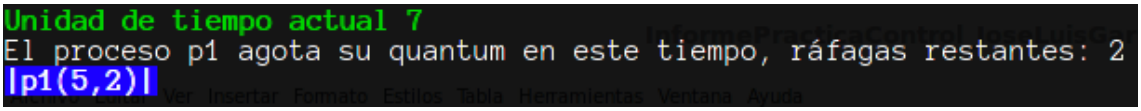
Al final de la ejecución de este tiempo los datos son:

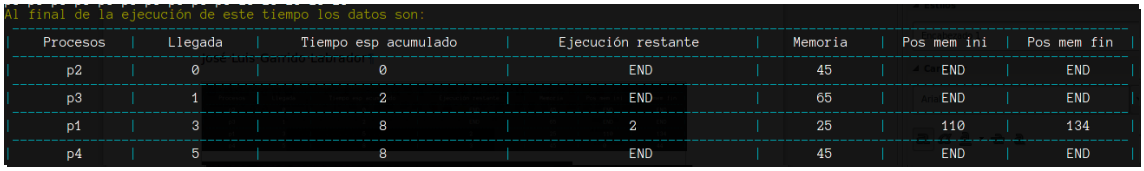
Procesos	Llegada	Tiempo esp acumulado	Ejecución restante	Memoria	Pos mem ini	Pos mem fin
p2	0	0	END	45	END	END
p3	1	2	END	65	END	END
p1	3	5	5	25	110	134
p4	5	5	3	45	NA	NA

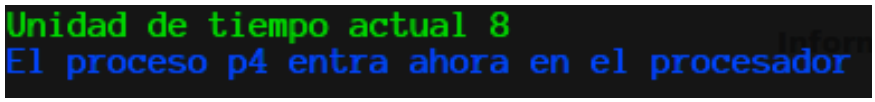


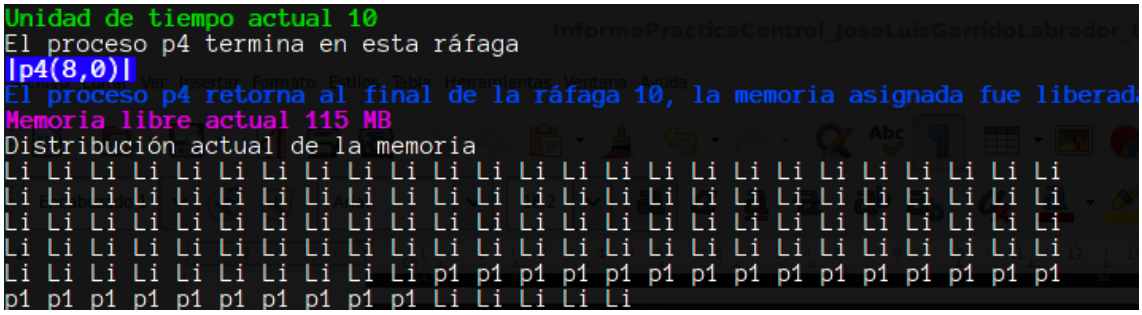
Al final de la ejecución de este tiempo los datos son:

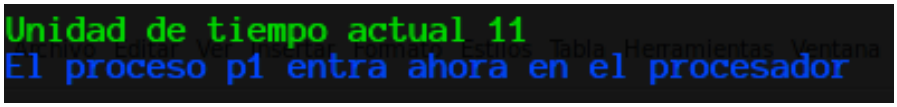
Procesos	Llegada	Tiempo esp acumulado	Ejecución restante	Memoria	Pos mem ini	Pos mem fin
p2	0	0	END	45	END	END
p3	1	2	END	65	END	END
p1	3	5	4	25	110	134
p4	5	6	3	45	0	44

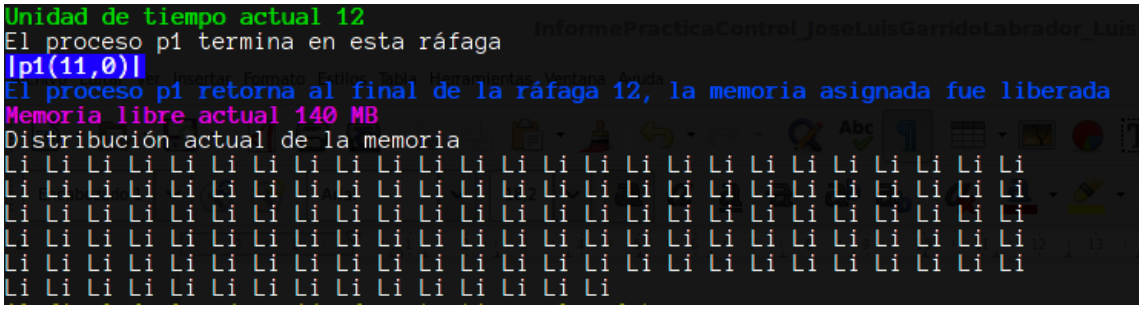


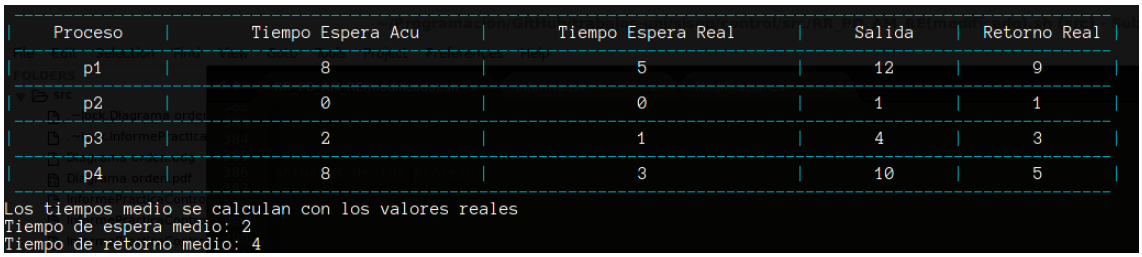












Como se puede observar los resultados son los mismos que los del ejercicio 2 hecho a mano.

7 Concluciones finales

En referencia a la gestión de procesos Round Robin es un sistema que realmente optimiza la realización multitarea que aunque puede realentizar todo el sistema dando valores de retorno individual superior al propio de cada proceso permite un retorno lo

bastante óptimo como para que ningún proceso tarde demasiado en ejecutarse completamente. El principal reto lo tiene cuando es la memoria la que realentiza el sistema al impedir que procesos se ejecuten antes si esta es inferior, sin embargo una memoria dinámica reubicable dentro de su limitud permite que muchos más procesos estén en la lista de ejecución al carecer de particionamiento y pérdidas de espacio.

8 Bibliografía y referencias

Omar Santos Bernabé
Documentación Round Robin – 2014/2015
Arrays en Bash
http://castilloinformatica.com/wiki/index.php?title=Arrays_en_Bash
José Manuel Sáiz Diez
Planificación de Procesos, Gestión de Memoria