TP1 - SO

Santiago Cremón

September 2025

Acerca de la notación de los diagramas: Para indicar que un proceso comienza (es aceptado en el scheduler) se utilizará una regla vertical con el nombre del mismo y una *C*. A su vez las etiquetas *IO S*, *IO D* y *T* se utilizarán para indicar que el proceso que se estaba ejecutando solicito/terminó entrada/salida o ha terminado respectivamente.

1 Generación aleatoria

En este ejercicio se pide generar dos problemas con datos aleatorios deshabilitando completamente E/S para los trabajos.

1.1 Primer Ejemplo

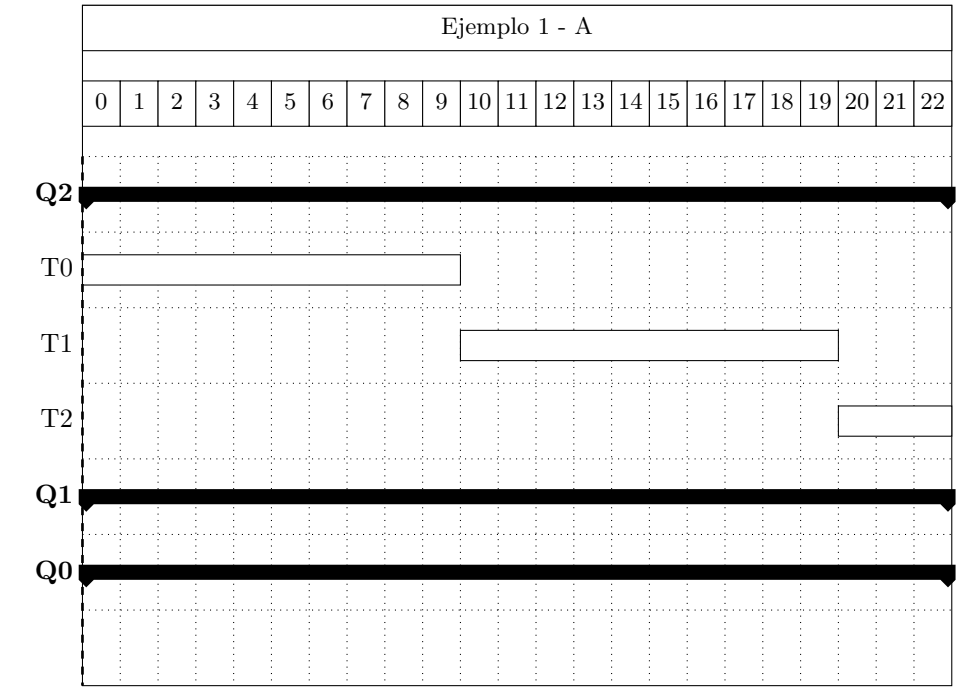
En el primer caso, el programa `mlfq.py` fue llamado con los siguientes flags:

```
python3 mlfq.py -m 25 -M 0 -i 0 -c
```

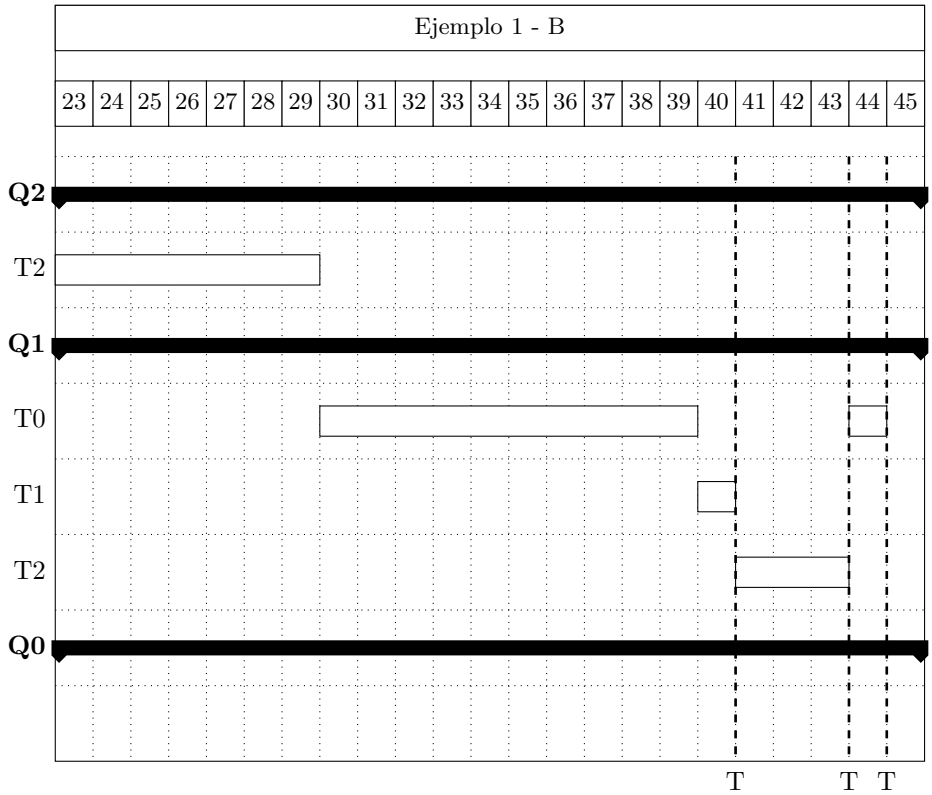
Dando como resultado la siguiente lista de trabajos con sus opciones:

```
Here is the list of inputs:
OPTIONS jobs 3
OPTIONS queues 3
OPTIONS allotments for queue  2 is   1
OPTIONS quantum length for queue  2 is  10
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 0
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime  21 - ioFreq   0
  Job  1: startTime   0 - runTime  11 - ioFreq   0
  Job  2: startTime   0 - runTime  13 - ioFreq   0
```

Podemos ver los datos plasmados en el siguiente diagrama de Gantt:



T0, T1, T2 C



En este caso poseemos tres procesos que requieren de un tiempo relativamente moderado de CPU y de tres colas de prioridad las cuales poseen el mismo valor de allotment y quantum. Como no se realizan E/S y la flag de -S (stay) está desactivada los tres procesos se irán ejecutando y bajando puestos en las colas de prioridad hasta llegar a la última sin alterar el orden de ejecución

dispuesto en un principio (T0 -¿ T1 -¿ T2). Por este mismo motivo, el orden de la ejecución de estos se da de manera muy similar a la dispuesta por un scheduler programado con un algoritmo RR a pesar de ser un MLFQ.

Todos los datos obtenidos de la traza de ejecución pueden encontrarse en el archivo `ej1_ejemplo1.txt`.

1.2 Segundo Ejemplo

En el segundo caso el programa fue llamado con los siguientes flags:

```
python3 mlfq.py -s 7 -m 25 -M 0 -i 0 -c
```

Teniendo en cuenta que limitamos MAXIO e IOTIME a cero para evitar cualquier tipo de E/S, obtenemos la siguiente lista de trabajos con una configuración determinada:

```
Here is the list of inputs:
OPTIONS jobs 3
OPTIONS queues 3
OPTIONS allotments for queue  2 is   1
OPTIONS quantum length for queue  2 is  10
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 0
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime   8 - ioFreq   0
  Job  1: startTime   0 - runTime  16 - ioFreq   0
  Job  2: startTime   0 - runTime  13 - ioFreq   0
```

Siendo las estadísticas finales las siguientes:

```
Final statistics:
Job  0: startTime 0 - response 0 - turnaround 8 - waitingTime: 0
Job  1: startTime 0 - response 8 - turnaround 34 - waitingTime: 18
Job  2: startTime 0 - response 18 - turnaround 37 - waitingTime: 24

Avg  2: startTime n/a - response 8.67 - turnaround 26.33 - waitingTime 14.00
```

Si vemos la traza obtenida en el archivo `ej1_ejemplo2.txt` podemos ver que el comportamiento de esta ejecución es muy similar a la del **Ejemplo 1**

2 Extensión de funcionalidades

Se solicita extender la funcionalidad del programa `mlfq.py` para que también admita un registro de **Throughput** y **Waiting Time**.

El código de la implementación se encuentran delimitados por comentarios en el archivo mlfq.qy

En esta sección se procede a describir la implementación de las nuevas funcionalidades del programa.

En el caso del cálculo de **throughput**, cómo este se refiere a la cantidad de procesos terminados por unidad de tiempo, debemos de imprimir cada dos ticks el cálculo de *procesosTerminados / tickActual*. Para esto, simplemente agregamos una impresión en pantalla del cálculo con las variables que ya nos proporciona el script. Sólo hemos de cerciorarnos de que el mensaje en terminal se envíe cada dos ticks y esto es fácilmente configurable con un condicional dentro de la sección de la traza de ejecución.

Por otro lado, para calcular el **waiting time** individual de cada proceso será necesario obtener información de el estado de este durante toda la ejecución, ya que debemos de calcular la cantidad de tiempo en el que cada proceso pasa en estado *ready* sin ser ejecutado. En particular haremos uso de la estructura proporcionada en el script donde se almacena el estado de I/O, el momento en el que el proceso inicio y el momento en el que este terminó. También realizaremos una modificación sobre esta, agregando como **key** el string *waitingTime* para almacenar el valor en cada trabajo.

Particularmente el algoritmo se basará en revisar por cada tick si la tarea se está ejecutando y, si no lo esta haciendo, revisar si la misma se encuentra despachada y esperando a ser ejecutada con un condicional. En caso de que lo anteriormente mencionado sea afirmativo, se procede a actualizar el valor de waiting time en la estructura de dicha tarea sumando un tick. Una vez finalizada la traza, se procede a recorrer cada trabajo, ingresando a su estructura de datos correspondiente e imprimiendo el valor acumulado para waiting time.

En el caso de necesitar el cálculo para el promedio de el dato, tan solo es necesario imprimir la suma de todos los tiempos de espera dividido por la cantidad de trabajos. Todo el cálculo para trabajos individuales se implementa en la sección de traza, mientras que el promedio es implementado con una pequeña modificación a la impresión de pantalla en la sección de estadísticas finales.

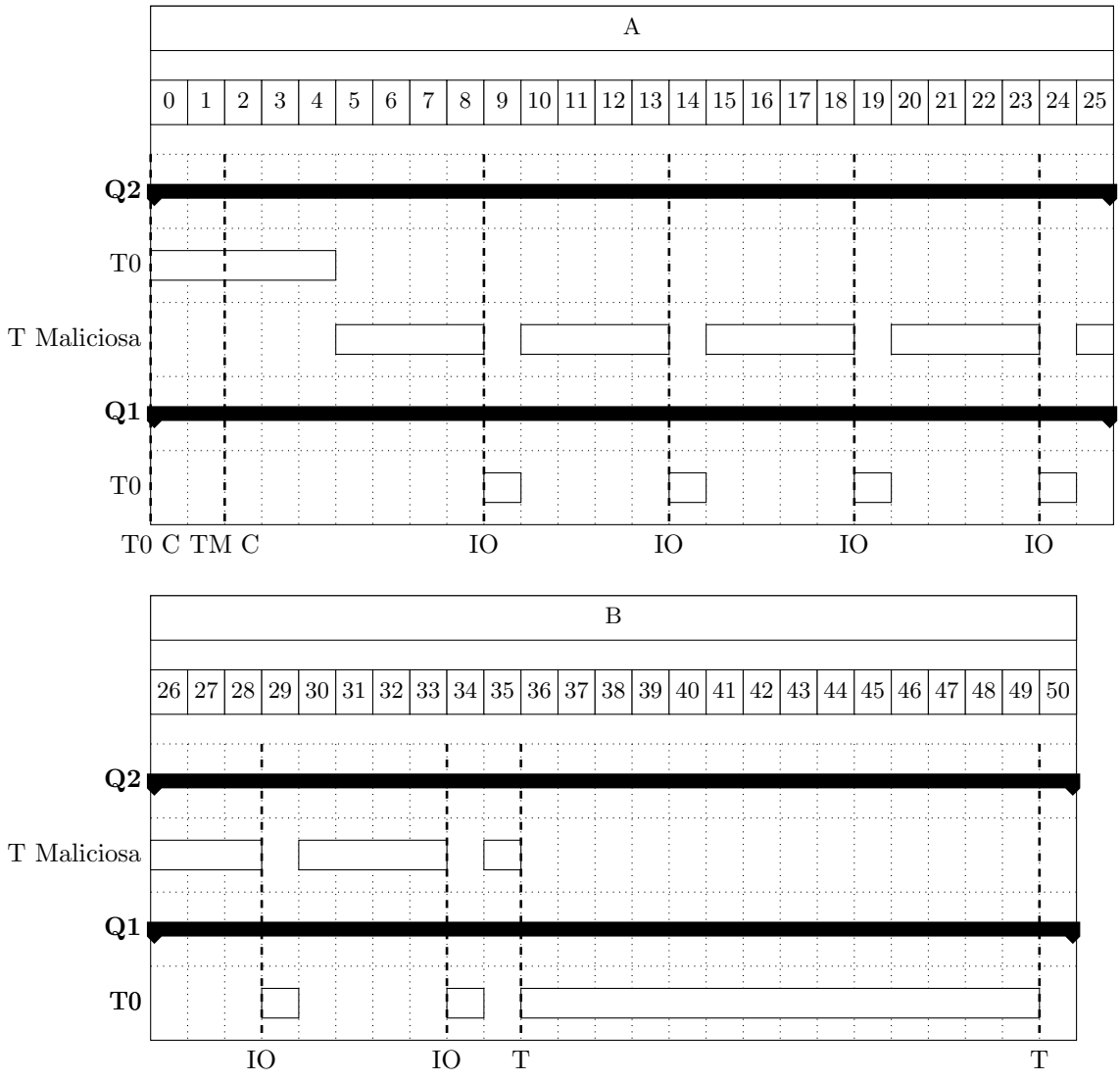
El comportamiento de estas nuevas utilidades puede verse en los resultados guardados en todos aquellos archivos que comparten nombre `ej2_test*.txt`.

3 Programa Malicioso

Debido a que el diseño del scheduler afecta directamente a la seguridad del sistema, podemos suponer que uno que carezca de ciertas precauciones puede conceder a ciertos programas maliciosos la capacidad de acaparar tiempo de CPU y dejar fuera de ejecución a otros procesos del sistema.

En este caso vamos a suponer un programa intensivo en E/S (malicioso) y uno extensivo en uso de CPU no-interactivo. También vamos a permitir que el scheduler carezca de un sistema de priority boost para evitar que todos los procesos vuelvan, cada cierto periodo, a la cola de mas alta prioridad. Además vamos a hacer uso de las flags *–stay* e *–iobump* para que aquellos procesos interactivos que sean intensivos en entrada y salida sean beneficiados.

Esta configuración, suponemos beneficiosa para el software malicioso, se ve plasmada en el siguiente diagrama de Gantt:



Nota: En este caso, debido a las limitaciones del script, la E/S de ambas tareas se establece a un intervalo de tiempo corto (un tick), pero en un escenario real sólo la tarea maliciosa intentaría aprovecharse de lapsos ínfimos de E/S para mantenerse en alta prioridad. Otros procesos pueden variar en su tiempo de espera.

Como podemos ver en las estadísticas finales

```
Final statistics:
Job 0: startTime 0 - response 0 - turnaround 50 - waitingTime: 25
Job  1: startTime 2 - response 3 - turnaround 34 - waitingTime: 3
```

Una MLFQ sin un priority boost puede ser "engañada" por una tarea al abusar de las llamadas de E/S para mantenerse ejecutando por tiempos indefinidos.

En este caso **T Maliciosa** al ser admitida por el scheduler aparta totalmente de la CPU a **T0** cuando la segunda agota su quantum por primera vez. A partir de este momento sólo se le permitirá ejecutarse cuando la primera entra en el ciclo de E/S necesario para renovar las prioridades.

La idea detrás de este *exploit* es que la tarea maliciosa aproveche gran parte de su *allotment* y, antes de ser arrebatada de la CPU, haga E/S por un tiempo relativamente corto.

Es necesario que la MLFQ tenga implementada en su algoritmo una regla de renovación de prioridades al realizar una llamada de E/S pero, de ser este el caso, la tarea maliciosa se encontraría permanentemente renovando su posición en la cola de mayor prioridad por sobre todas los demás procesos causando inanición en estos.

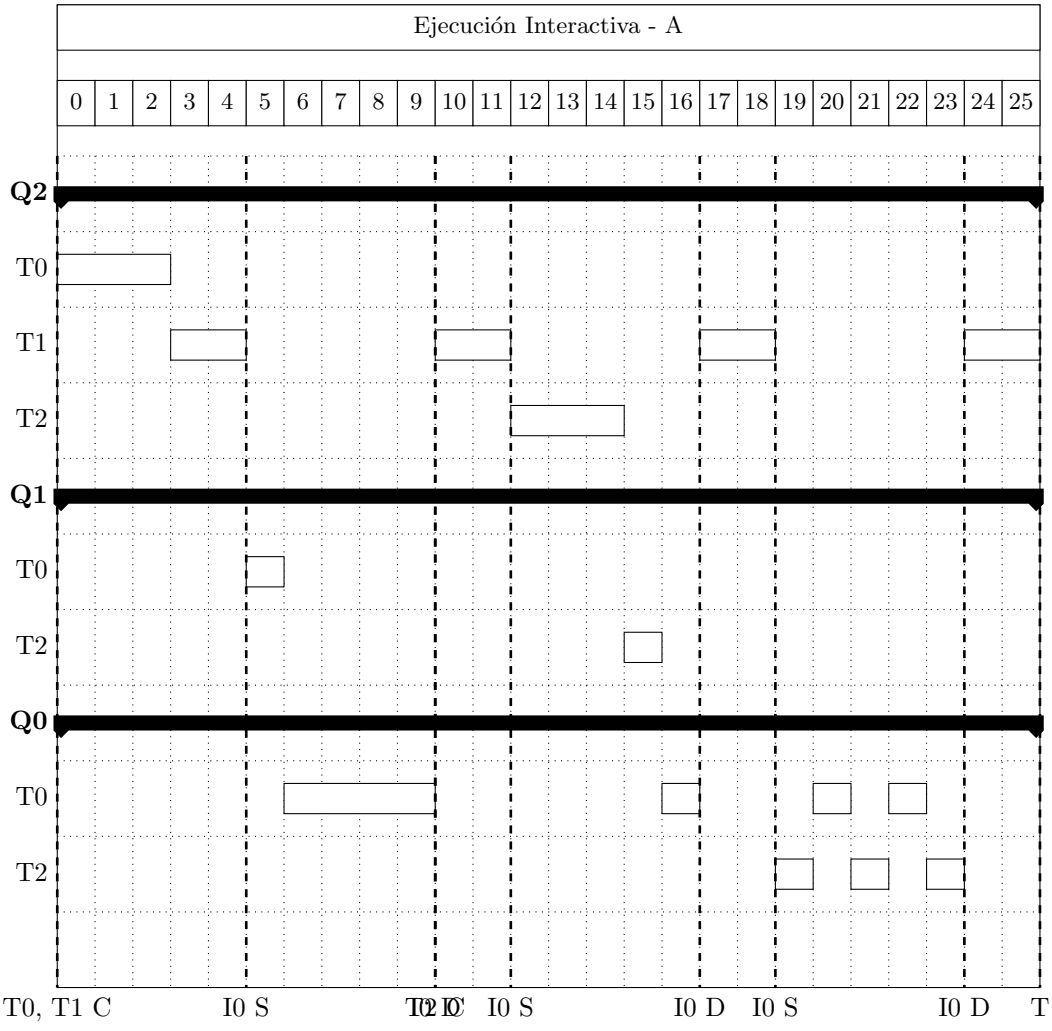
4 Simulación de tareas

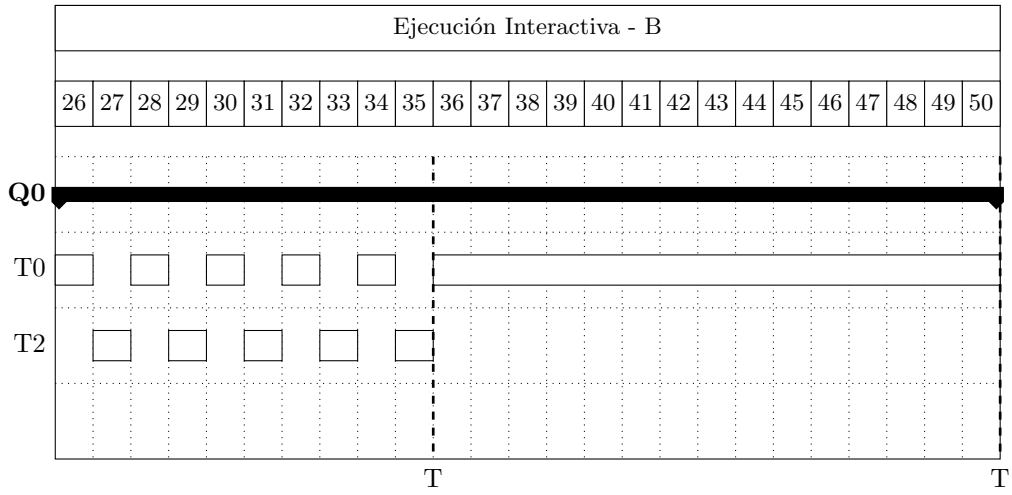
4.1 Ejecución beneficiosa para tarea interactiva

En este caso, además de utilizar una lista de trabajos dada, ejecutamos `mlfq.py` con la configuración siguiente:

```
Here is the list of inputs:
OPTIONS jobs 3
OPTIONS queues 3
OPTIONS allotments for queue  2 is   1
OPTIONS quantum length for queue  2 is   3
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is   1
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is   1
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO True
OPTIONS iobump True
```

Los resultados de la traza del programa volcados en el archivo `ej4_interactiva.txt` se ven plasmados en el siguiente diagrama de Gantt:





Podemos ver que, al igual que en el caso de la tarea maliciosa, una configuración que de **scheduler** que establezca las flags de *–stay* e *–iobump* también beneficiará a las tareas interactivas. Esto es debido a la renovación tanto de allotment como de prioridad en la cola que estas opciones dan a las tareas que realizar E/S. A su vez, intensificamos este efecto al no activar la opción de *–boost*, la cual permitiría cada cierto período de tiempo que todas las tareas vuelvan a la cola de mayor prioridad y, por lo tanto, ejecutarse momentáneamente.

Un **scheduler** que priorice respuesta de las tareas ante los usuarios debe de tener en cuenta que se le debe dar otorgar especial atención a las tareas que sean intensivas en E/S (siempre asumiendo que estas son las tareas interactivas con el usuario).

En este caso, y con esta configuración, las tareas no-interactivas verán afectado su tiempo de ejecución negativamente debido a que no pueden hacer uso de las flags anteriormente mencionadas. De esta manera, se verán relegadas a esperar a que **T1** (la tarea interactiva) termine su ejecución antes de ser priorizadas por el **scheduler**.

Las estadísticas finales, en las cuales podemos ver la diferencia de tiempo de esperas entre los distintos tipos de tareas, son:

```
Final statistics:
Job 0:startTime 0 - response 0 - turnaround 50 waitingTime: 20
Job 1:startTime 0 - response 3 - turnaround 26 - waitingTime: 3
Job 2:startTime 10 - response 2 - turnaround 26 - waitingTime: 14

Avg  2: startTime n/a - response 1.67 - turnaround 34.00 - waitingTime 12.33
```

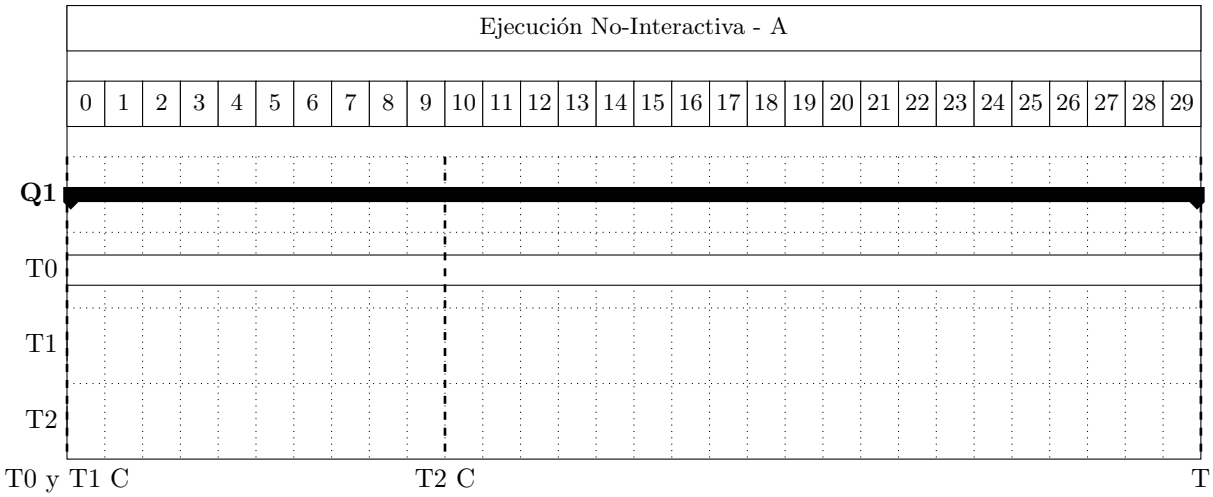
4.2 Ejecución beneficiosa para tareas no-interactivas

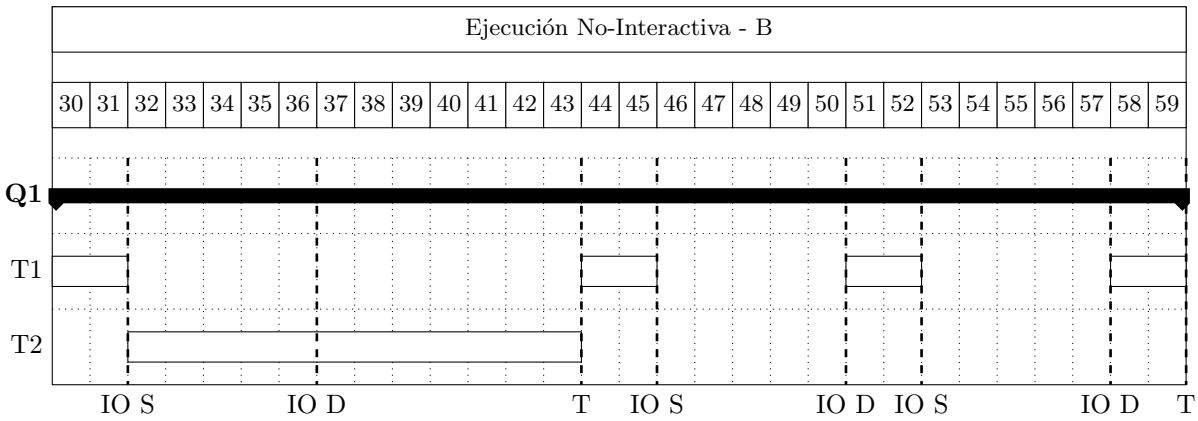
En el caso de beneficiar a las tareas no-interactivas, se procede a llamar a `mlfq.py` con la siguiente configuración:

```
Here is the list of inputs:
OPTIONS jobs 3
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  30
```

```
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is   1
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False
```

La traza de ejecución, por su parte se ve plasmada en el siguiente diagrama de Gantt:





Podemos ver que el uso de una sola cola emparejado con cantidades de quantum excesivas y el desuso de opciones como *–stay* e *–iobump* que permiten a las tareas interactivas recobrar prioridad en la cola acaban por beneficiar en sobremanera a aquellas tareas que requieren gran cantidad de tiempo en CPU y no requieren de eventos de E/S.

En este caso se trata de recrear un algoritmo de FIFO (esta es la razón de asignar el quantum a el mismo valor de tiempo que el requerido por la tarea más intensiva en cómputo). Pero además, el hecho de que la tarea interactiva requiera de esperar a que el evento de E/S finalice provoca que se vea relegada a esperar, cada vez que necesite realizar uno de estos eventos, que otra tarea finalice para poder reanudar con su ejecución.

Los resultados pueden verse claramente en las estadísticas finales:

```
Final statistics:
Job 0: startTime 0 - response 0 - turnaround 30 - waitingTime:    0
```

```
Job 1: startTime 0 - response 30 - turnaround 60 - waitingTime:  37
Job 2: startTime 10 - response 22 - turnaround 34 - waitingTime: 22

Avg 2: startTime n/a - response 17.33 - turnaround 41.33 - waitingTime 19.67
```

Nótese que el tiempo de respuesta y de espera son excesivos para una tarea que, se supone, interactúa con el usuario.