Tp Scheduling

Lautaro Manuel Orsi

Segundo Cuatrimestre 2025

1 Ejercicio 1

Tomemos los siguientes comandos:
A) ./mlfq.py -s 10 -m 25 -M 0 -c
B) ./mlfq.py -s 1250 -m 25 -M 0 -c

Notemos que tanto para A como para B vamos a estar **limitando la duración** de los procesos a 25 milisegundos y **desactivando E/S** (poniendo la frecuencia en 0 ms), luego usaremos para ambos una **semilla generadora** de parametros al azar para el resto de opciones dentro del simulador (semilla 10 y semilla 5 respectivamente).

1.1 Comportamiento de A

Para esta semilla se generarán 3 procesos:
-P0, con start time 0, duración de 14 ciclos de scheduler y sin E/S
-P1, con start time 0, duración de 14 ciclos de scheduler y sin E/S
-P2, con start time 0, duración de 20 ciclos de scheduler y sin E/S

Ejecución:
Los 3 procesos seran habilitados a ser ejecutados por el scheduler pero comenzará la ejecución de P0 *(es importante notar que los 3 procesos tienen prioridad 2 al comenzar, pero debe elegirse uno arbitrario que vaya primero)*.
Lo que vemos es que los 3 procesos se van a ejecutar por la totalidad del quantum que le otorga el scheduler (en este caso 10 ticks), dado que la duración de los tres procesos es mayor al quantum.
Una vez terminada la primer ronda de ejecuciones observamos que P0 pasará a tener prioridad 1 ya que el allotment es de 1 y será ejecutado por los 4 ciclos que le faltaban para completar su ejecución, lo mismo ocurrirá con P1 y por ultimo P2 se ejecutará por otros 10 ciclos que resultara en su finalización.

Estadisticas finales: Para cualquiera de los 3 procesos podemos observar
- Response time: La suma de tiempo de ejecución de los procesos anteriores (en este caso 10 ticks por cada proceso). - Turnaround: La cantidad de ticks desde el start time hasta que finalizaron.

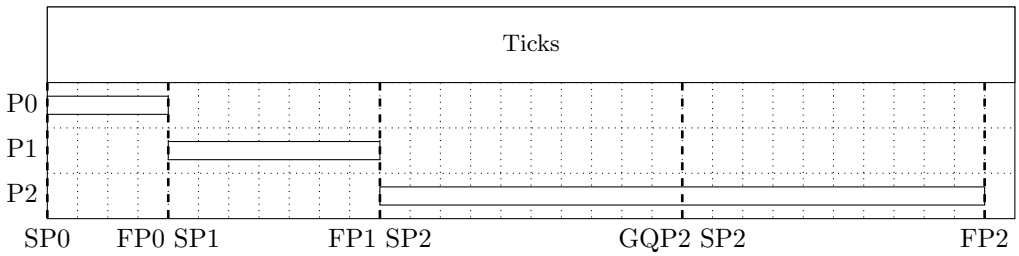
1.2 Comportamiento de B

Para esta semilla se generarán 3 procesos:
-P0, con start time 0, duración de 4 ciclos de scheduler y sin E/S
-P1, con start time 0, duración de 7 ciclos de scheduler y sin E/S
-P2, con start time 0, duración de 20 ciclos de scheduler y sin E/S

Ejecución:
Nuevamente, los 3 procesos serán habilitados a ser ejecutados por el scheduler pero comenzará la ejecución de P0.
Vemos que en este caso P0 tiene u tiempo de ejecución (4 ticks) menor al quantum otorgado (10 ticks), con P1 ocurre algo similar dado que su *runtime* es de 7 ticks pero para el caso de P2 veremos algo parecido al caso A ya que necesitará dos rondas de quantum para poder terminar, pero al haber finalizado los otros dos procesos este retoma el uso de la CPU instantaneamente.
Si bien tener procesos con *runtimes* menores al quantum otorgado permite terminar rapidamente su ejecución, no afectará al resto de procesos dado que el quantum sobrante no será agregado al que lo sigue, simplemente se reseteará al valor a 10 ticks y comienza con el siguiente proceso.

Estadisticas finales: Nuevamente en lineas generales para cualquiera de los 3 procesos podemos observar
- Response time: La suma de tiempo de ejecución de los procesos anteriores.
P0, 0 ticks
P1, 4 ticks
P2, 11 ticks
- Turnaround: La cantidad de ticks desde el start time hasta que finalizaron.
P0, 4 ticks
P1, 11 ticks
P2, 31 ticks

Diagrama de Gantt:



Nota:
SPX: comienza ejecución de PX
GQPX: Gasta su quantum PX
FPX: finaliza ejecución de PX

Figure 1: Diagrama de Gantt

2 Ejercicio 2

2.1 Waiting Time:

Para lo pedido basta con añadir un nuevo atributo *'waitingTime'* a los trabajos, luego chequeamos por cada unidad de tiempo que pase y para todos los procesos **salvo el que está siendo ejecutado actualmente**:
-Si su tiempo de comienzo de ejecución es menor o igual al actual (esto quiere decir que ya está listo para ser ejecutado)
-Si su tiempo de ejecución restante es mayor a 0 (esto quiere decir que no finalizó)
-No esté realizando operaciones I/O (pues en este caso el proceso no estaría **ready**)
Cuando se cumplen estas condiciones incrementamos el *waitingTime*.
Por ultimo, agregamos el atributo a la sección del codigo encargada de imprimir los otros valores individuales y luego la sumatoria de los mismos en la sección *Average*

2.2 Throughput:

En este caso no hace falta agregar ningun atributo gracias a que contamos con la variable *finishedJobs*, usando esto podemos verificar cada 2 unidades de tiempo (con un simple *currentTime* mod 2 == 0) la cantidad de procesos finalizados e imprimir el promedio por pantalla. Es importante notar que la impresión debe hacerse **luego** de haber chequeado si el proceso que se ejecutó en esta unidad finalizó, ya que hacerlo antes nos retornaría un número erróneo y excluir el caso currentTime = 0 para evitar la división por cero.

2.3 Ejemplo

Veamos un caso de ejecución, tomemos el comando **./mlfq.py -l 0,20,11:1,20,0 -q 10**, acá tenemos 2 procesos:

- **Proceso A**: Comienza en ms 0, 20ms de duración y operaciones I/O cada 11ms (con duración predeterminada de 5ms)
- **Proceso B**: Comienza en ms 1, 20ms de duración y sin operaciones I/O

Milisegundo	Estado A	Estado B	WaitingTime A	WaitingTime B
0	Running	Blocked	0	0
1	Running	Ready	0	1
...	Running	Ready	0	...
9	Running	Ready	0	9
10	Ready	Running	1	9
...	Ready	Running	...	9
19	Ready	Running	10	9
20	Running	Ready	10	10
21	Waiting	Running	10	10
...	Waiting	Running	10	10
25	Waiting	Running	10	10
26	Ready	Running	11	10
...	Ready	Running	...	10
30	Ready	Running	15	10
31	Running	Finished	15	10
...	Running	Finished	15	10
39	Finished	Finished	15	10

Podemos observar que corriendo la simulación en el programa obtenemos los mismos resultados planteados en el cuadro

```
Final statistics:
  Job  0: startTime   0 - response   0 - turnaround  40 - waitingTime  15
  Job  1: startTime   1 - response   9 - turnaround  30 - waitingTime  10
```

Figure 2: Resultado simulador

Por ultimo respecto al *Throughput* podemos observar lo siguiente, antes del tick 31 no habia terminado ningun proceso y por lo tanto nuestro *Throughput* era de 0 (0/cant ticks) pero a partir del 31 ya tenemos un proceso finalizado y por lo tanto el *Throughput* es de 1/32, este valor va a ir decreciendo a medida que

pase el tiempo pero eventualmente en el tick 40 al finalizar otro proceso vamos a llegar a un throughput final de $2/40 = 0.05$

```
[AGREGADO TP] THROUGHPUT: 0.0
[ time 30 ] Run JOB 1 at PRIORITY 1 [ TICKS 0 ALLOT 1 TIME 0 (of 20) ]
[ time 31 ] FINISHED JOB 1
[ time 31 ] Run JOB 0 at PRIORITY 1 [ TICKS 8 ALLOT 1 TIME 8 (of 20) ]
[AGREGADO TP] THROUGHPUT: 0.03125
[ time 32 ] Run JOB 0 at PRIORITY 1 [ TICKS 7 ALLOT 1 TIME 7 (of 20) ]
[ time 33 ] Run JOB 0 at PRIORITY 1 [ TICKS 6 ALLOT 1 TIME 6 (of 20) ]
[AGREGADO TP] THROUGHPUT: 0.029411764705882353
[ time 34 ] Run JOB 0 at PRIORITY 1 [ TICKS 5 ALLOT 1 TIME 5 (of 20) ]
[ time 35 ] Run JOB 0 at PRIORITY 1 [ TICKS 4 ALLOT 1 TIME 4 (of 20) ]
[AGREGADO TP] THROUGHPUT: 0.027777777777777776
[ time 36 ] Run JOB 0 at PRIORITY 1 [ TICKS 3 ALLOT 1 TIME 3 (of 20) ]
[ time 37 ] Run JOB 0 at PRIORITY 1 [ TICKS 2 ALLOT 1 TIME 2 (of 20) ]
[AGREGADO TP] THROUGHPUT: 0.02631578947368421
[ time 38 ] Run JOB 0 at PRIORITY 1 [ TICKS 1 ALLOT 1 TIME 1 (of 20) ]
[ time 39 ] Run JOB 0 at PRIORITY 1 [ TICKS 0 ALLOT 1 TIME 0 (of 20) ]
[ time 40 ] FINISHED JOB 0
[AGREGADO TP] THROUGHPUT: 0.05
```

Figure 3: Throughput simulado

3 Ejercicio 3

Tomemos el siguiente comando y veamos en detalle su significado:
./mlfq.py -n 1 -q 10 -A 1 -j 2 -i 0 -S -I -l 0,30,10:0,5,3 -c

- **-j 2:** Declaramos 2 procesos
- **-l 0,30,10:0,5,3:** Declaramos los procesos con comienzo en tick 0, el primero durará 30 ticks y tendrá opearción E/S cada 10 ticks, el 2do durará 5 ticks y realizará opearciónes E/S cada 3.
- **-n 1:** Tendremos una única cola de procesos
- **-q 10:** Para esta única lista, el quantum es de 10 unidades tiempo
- **-a 1:** Cada proceso debería ser enviado a un nivel inferior luego de gastar su quantum (Veremos que este parámetro es irrelevante)
- **-i 0:** La duración de los E/S será de 0 unidades de tiempo
- **-S:** Luego de la interrupción por E/S el scheduler restartea el quantum y allot del proceso que utiliza E/S
- **-I:** Al finalizar la E/S el proceso pasa al frente de la cola actual
- **-c:** Lista detallada de la ejecución (obviable)

¿Qué consecuencias tiene?
Podemos ver que nuestro proceso **P0** va a ser el primero en ejecutarse, cuando consume sus 10 ticks de quantum realiza una opearción E/S que, gracias a los flags -S -I, le permiten recuperar su quantum y allot, mantenerse en la cola y pasar a ser el primero en el orden.
Todo esto hace que a diferencia de lo normal (pasar a una cola de menor prioridad/final de la cola actual) es como si a **P0** no se le hubiera sido otorgado el ciclo de schedule anterior, entonces puede acaparar todo el CPU hasta que termine con su ejecución.
¿Que pasaría si modificamos el -i 1 por -i x para algún N arbitrario?
Tener un tiempo de ejecución para las operaciones E/S distinto a 0 nos cambiaria un poco el resultado, si por ejemplo nuestro proceso **P0** llama a la operación E/S que dura N tiempo ¿ quantum restante vamos a ver que el scheduler reemplazará a **P0** por un nuevo proceso. Esto tendrá diferentes consecuencias según la configuración del sistema:

- Si tenemos mas de una cola de prioridad y **P0** se queda sin allot va a ser degradado un nivel, en ese caso si hay otro proceso con mayor prioridad entonces muy malicioso no va a ser ya que **P0** pasará al frente de la cola a la que fue degradado y el scheduler le va a dar prioridad a todo aquel que esté en una cola de mayor nivel que **P0**

- Si tenemos mas de una cola de prioridad pero **P0** todavia tiene allotment entonces una vez que termine la operación E/S (y si no hay otro proceso en ejecución de mayor prioridad) **P0** va a recibir el control de la CPU, restarteando su quantum y allotment (por -S). Esto es medianamente malicioso, ya que si aparece otro proceso de mayor nivel cuando salte una nueva seleccion de proceso el scheduler le va a dar prioridad a este proceso nuevo.

- Si tenemos una única cola ahí si puede ser malvado (un verdadero Micky Vainilla incluso) ya que va a pasar al frente de la misma cola cada vez que finalice su E/S y como la politica del scheduler es seleccionar un proceso nuevo para ejecutar cada vez que ocurre una operación del estilo, va a interrumpir a cualquier otro que este siendo ejecutado.

3.1 Resultados

Acá podemos observar los resultados de las operaciones corriendo en el simulador, donde a raíz de las operaciones I/O, **P0** puede forzar al scheduler a darle todo el quantum que quiera hasta finalizar, haciendo que **P1** se vea obligado a permanecer en estado *READY* hasta que **P0** termine (resultando en un *Waiting Time* = duración de **P0**)

```
Final statistics:
  Job  0: startTime   0 - response   0 - turnaround  30 - waitingTime   0
  Job  1: startTime   0 - response  30 - turnaround  35 - waitingTime  30
```

Figure 4: Resultados simulador

4 Ejercicio 4

4.1 Priorizando Interactivo

Tomemos el comando
./mlfq.py -j 3 -l 0,30,0:0,8,2:10,12,0 -Q 3,1,2 -A 1,1,800 -S -I
Con esto definimos 3 distintos niveles de queues, cada una con un quantum y allotment diferente:

Nivel 1: quantum 3, allotment 1

Nivel 2: quantum 1, allotment 1

Nivel 3: quantum 2, allotment 800

La motivacion de esta configuracion es forzar a los procesos 1 y 3 a ser degradados por no hacer I/O, como el allotment de las dos primeras queues es de 1 van a ser degradadas apenas gasten su primer quantum. Por otro lado el proceso 2 realiza operaciones I/O cada 2ms, entonces nunca llega a gastar el quantum otorgado y -gracias a los flags Stay e IOBump- se va a mantener en la queue 1. Por esto es que el scheduler le va a otorgar siempre prioridad al proceso 2 (al estar en la queue de mayor prioridad).

```
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
```

Figure 5: Priorizando Interactivo

4.2 Priorizando No Interactivo

En este caso deberiamos sacar los flags -S -I y modificar quantum largo, allotment corto para la cola de prioridad mas alta, un quantum bajo, alltoment bajo para la prioridad media y un quantum (indiferente), allotment alto para la ultima:
./mlfq.py -j 3 -l 0,30,0:0,8,2:10,12,0 -Q 800,1,800 -A 1,1,800

Nivel 1: quantum 800, allotment 1

Nivel 2: quantum 1, allotment 1

Nivel 3: quantum 800, allotment 800

Podemos ver una clara diferencia respecto al waiting time, esto se debe a que al tener un allotment alto en la cola de prioridad alta y no tener los flags -S -I el proceso 2 se degrada a colas de prioridades mas bajas, mientras que los otros dos procesos pueden aprovechar al maximo el quantum otorgado por el scheduler (que al ser un numero tan alto alcanza para finalizar su ejecución en un solo ciclo, tecnicamente se podria haber puesto un quantum = duracion del proceso no-interactivo mas largo). Por lo tanto el proceso interactivo esta obligado a esperar a que los otros procesos terminen de ejecutarse para poder recibir capacidad de procesamiento (luego del primer I/O).

```
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
```

Figure 6: Priorizando No-Interactivo