Trabajo Práctico 1: Scheduling

Lisandro Cordoba Lazzaro

1. Ejercicio 1

1.1. Generando la simulacion

Con los siguientes comandos generamos simulaciónes aleatorias sin E/S asociadas a las semillas 1 y 3 respectivamente.
Limitamos la duración máxima de los procesos para facilitar el posterior analisis de los datos obtenidos.
Agregamos '-c' para ver el responseTime y turnaroung de cada proceso.

```
mlfq.py -m 20 -M 0 -c -s 1
mlfq.py -m 20 -M 0 -c -s 3
```

1.2. Analizando los datos

Aclaracion: para referir al iesimo proceso notaremos Pi.

En ambos casos se generan 3 colas y 3 procesos

Proceso	startTime	runTime
0	0	3
1	0	15
2	0	10

Cuadro 1: Procesos creados semilla 1

Cola	quantum	allotment
2	10	1
1	10	1
0	10	1

Cuadro 2: Colas creadas semilla 1

Proceso	startTime	runTime
0	0	5
1	0	8
2	0	12

Cuadro 3: Procesos creados semilla 3

Cola	quantum	allotment
2	10	1
1	10	1
0	10	1

Cuadro 4: Colas creadas semilla 3

En las dos simulaciones se observa que los 3 procesos comienzan en la cola de maximo nivel (2) y comienza ejecutando P0, quien llega a terminar pues su runTime es menor al quantum. Luego se le entrega la CPU a P2, donde el comportamiento en cada caso se bifurca.

En la semilla 1 P2 tiene runTime mayor al quantum, por lo que ejecuta los 10 ms y baja a la siguiente cola de prioridad (1).
Se le entrega la CPU a P3, el cual agota su quantum y termina la ejecucion.
Finalmente, al no haber mas procesos en la cola de prioridad, el scheduler baja a buscar en la siguiente cola. Alli ejecuta P1 los 5 ms que le restaban y termina.

En la semilla 3 P2 tiene runTime menor al quantum, por lo que ejecuta sus 8 ms y termina.
Se le entrega la CPU al unico restante en la cola de prioridad, P3. Este ejecuta todo el quantum pero le restan 2 ms, por lo que es desalojado y baja a la siguiente cola de prioridad (1).
Como no hay mas procesos en la cola de maxima prioridad, el scheduler busca en la siguiente y ejecuta P3, terminando su runTime.

Tiempo	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Proceso	P0	P0	P0	P0	P0	P1	P1	P1	P1	P1	P1	P1	P1	P2
Tiempo	14	15	16	17	18	19	20	21	22	23	24			
Proceso	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2	P2			

Cuadro 5: Diagrama de Gantt

1.3. Diagrama de Gantt de la semilla 3

2. Ejercicio 2

2.1. Throughput

Para esta metrica aprovechamos que el programa ya lleva registro de la cantidad de tareas terminadas. Para imprimir el throughput cada 2 unidades de tiempo basta con calcular en cada iteración par del while la siguiente ecuacion.

```
currentThroughput = finishedJobs / currTime
```

2.2. Waiting Time

Para calcular esta métrica es necesario saber en cada iteracion cuales son los procesos ready. Es decir, aquellos que estan listos para ejecutar pero el scheduler no les otorgó la CPU.
En este caso un proceso está ready si cumple las siguientes condiciones:

1. Ya llegó al scheduler (startTime ¡= currTime).
2. Aún no terminó (timeLeft ¿0).
3. No está running (no es currJob).
4. No está bloqueado por E/S (doingIO == False).

Para llevar cuenta del waitingTime total de cada proceso, se agregó el campo "waitingTime.[a] la estructura de cada proceso.
En cada iteración, se aumenta en 1 el waitingTime de todos los procesos que cumplan las 4 condiciones.

El calculo del waitingTime promedio sigue la misma lógica que las metricas del codigo original:

$$\frac{1}{countJobs} \sum_{job}^{jobs} job.waitingTime$$

3. Ejercicio 3

3.1. Diseñando el escenario

Simularemos una situacion donde hay 2 procesos:

- P0 es un programa malicioso que utilizando E/S buscará maximizar el uso de la CPU con el objetivo de que los demas procesos no puedan acceder a ella.
- P1 es un programa intensivo en CPU que no hace E/S.

La idea es que P0 se mantenga siempre en la cola de prioridad maxima y P1 en la inferior. Es importante que P0 tampoco esté constantemente haciendo E/S pues el scheduler iria a la proxima cola y ejecutaria P1.

Para lograr esta situación pensé lo siguiente:
Tendremos dos colas de prioridad, la prioritaria tendrá quantum 10 y la no prioritaria quantum 1. Las politicas del scheduler diran que los procesos con E/S no bajan de prioridad y los que terminan E/S pasan al frente de la cola.
El programa malicioso hará E/S cada 9 ms, otorgandole la CPU al P1 pero manteniendose en la

prioritaria.
Aca es donde importa haberle dado quantum=1 a la no prioritaria: P1 podrá ejecutar exactamente el tiempo que P0 tarde en hacer E/s, ni 1 tick mas. Esto se debe a que quantum=1 obliga al scheduler a chequear si hay algun proceso en la cola prioritaria luego de cada tick, P0 retomará la CPU ni bien termine de hacer E/S .
Tambien es importante que el proceso malicioso dure lo suficiente como para terminar despues que P1. En caso contrario, una vez que el malicioso muera, P1 podria ejecutar libremente todo lo que le reste.

Proceso	startTime	runTime	ioFreq
0	0	100	9
1	0	50	0

Cuadro 6: Procesos

Cola	quantum	allotment
1	10	1
0	1	1

Cuadro 7: Colas

3.2. Comprobando con el simulador

Para ver como dan las métricas sin agregar que los procesos con E/S se mantengan en la misma prioridad y pasen al frente de la lista, corremos:

```
./mlfq.py --jlist 0,100,9:0,50,0 -n 2 -Q '10, 1' -a 1 -c
```

Luego queremos ver las metricas con las modificaciones del scheduler:

```
./mlfq.py --jlist 0,100,9:0,50,0 -n 2 -Q '10, 1' -a 1 -S -I -c
```

Proceso	responseTime	turnAround	waiting
0	0	186	31
1	9	86	36

Cuadro 8: Sin priorizar E/S

Proceso	responseTime	turnAround	waiting
0	0	155	0
1	9	140	90

Cuadro 9: Priorizando E/S

Como se puede ver, las hipotesis eran correctas. Al agregar las modificaciones al scheduler (Cuadro 9), el programa malicioso hace que P1 aumente significativamente su turnAround y aumente mas del doble su waitingTime.

4. Ejercicio 4

4.1. Beneficiar J1 (interactivo)

Se puede interpretar como el mismo caso que el Ejercicio 2, donde J2 sería el programa malicioso que quiere acaparar la CPU.

Configuramos el scheduler con 3 colas. La primera tiene quantum=3 y las demas quantum=1. Tambien agregamos que los procesos con E/S se mantengan en la misma prioridad y pasen al frente de la lista luego de terminar E/S.
De esta forma, al igual que en el Ejercicio anterior, logramos que P1 sea el único que se mantiene en la cola de prioridad y siempre que no esté haciendo E/S tendrá CPU.

```
./mlfq.py --jlist 0,30,0:0,8,2:10,12,0 -n 3 -Q '3,1,1' -a 1 -S -I -c
```

Lo podemos comparar corriendolo con la siguiente configuracion: que los procesos con E/S se mantengan en la misma prioridad y pasen al frente de la lista

4.2. Perjudicar J1 beneficiando a los demas

Para lograr que J0 tenga buenas métricas, podemos setear quantum=30 para que una vez que se le otorgue la CPU pueda terminar toda su ejecucion. Lo mismo ocurre con J3 pues corre en menos de 30ms.

Basta con definir 1 cola con quantum=30 y allotment=1.

```
./mlfq.py --jlist 0,30,0:0,8,2:10,12,0 -n 1 -Q '30' -a 1 -c
```

4.3. Comparación

Proceso	responseTime	turnAround	waiting
0	0	30	0
1	30	60	37
2	22	34	22

Cuadro 10: Version 4.2

Proceso	responseTime	turnAround	waiting
0	0	50	20
1	3	26	3
2	2	26	14

Cuadro 11: Version 4.1