DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Scheduling

2 de Mayo de 2016 — Sistemas Operativos

Integrante	LU	Correo electrónico
Costa, Manuel José Joaquin	035/14	manucos94@gmail.com
Coy, Camila Paula	033/14	camicoy94@gmail.com
Ginsberg, Mario Ezequiel	145/14	ezequielginsberg@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 1

Para crear la tarea pedida, creamos un nuevo método en el archivo `task.cpp` llamado `TaskConsola`. El código que implementa la solución es sumamente simple.

La tarea itera sobre la cantidad de llamadas bloqueantes que el usuario pasó por parámetro, generando por cada iteración un número pseudoaleatorio, `nro_random`, y realizando la respectiva llamada de *I/O* (cuya duración es `nro_random`).

Notar que para aumentar el grado de aleatoriedad inicializamos el valor de la semilla a partir del tiempo del sistema, en lugar de un valor fijo, cada vez que se llama a `TaskConsola`.

Es importante que, aunque no se vea en nuestro código, antes de bloquearse la tarea tiene que gastar un ciclo en el CPU, justamente para poder hacer la solicitud de *I/O*.

La figura 1 nos ayuda a entender mejor el funcionamiento de este tipo de tarea: los procesos 2 y 3 entran juntos en tiempo 0, el 0 y 1 en tiempo 5. Cada par corre en paralelo gracias a que hay dos procesadores, lo que facilita la comparación. El objetivo del primer par es ilustrar el caso en que la duración de las llamadas bloqueantes está prefijada: ambos procesos hacen dos llamadas de *I/O* cuya duración es exactamente 4 (para lograr esto hacemos que `bmin=bmax=4`). Por otra parte, los procesos 0 y 1, aunque reciben los mismos parámetros cada uno, muestran comportamientos distintos debido a que el rango de duración de cada bloqueo es de longitud mayor a 0.

Puede verse también que el tiempo que cada proceso pasa en el CPU antes de bloquearse es de exactamente un ciclo, que es lo que tarda en hacer la solicitud de *I/O*. En el último ciclo de cada proceso se hace el `exit`. Ambas cosas no dependen directamente de nuestro código, sino de las implementaciones de `uso_IO` y el simulador.

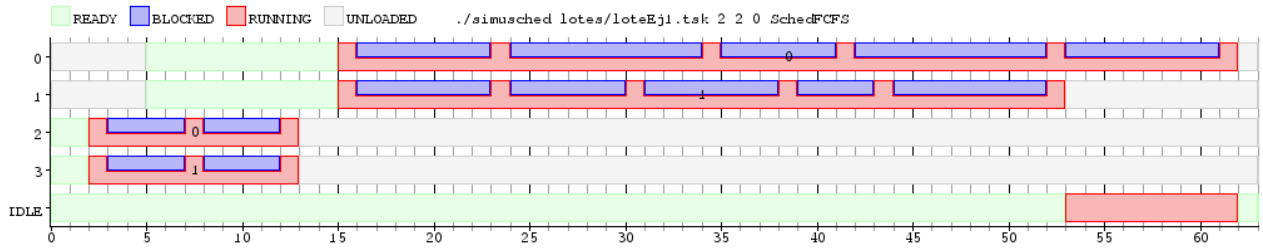

Figura 1

2. Ejercicio 2

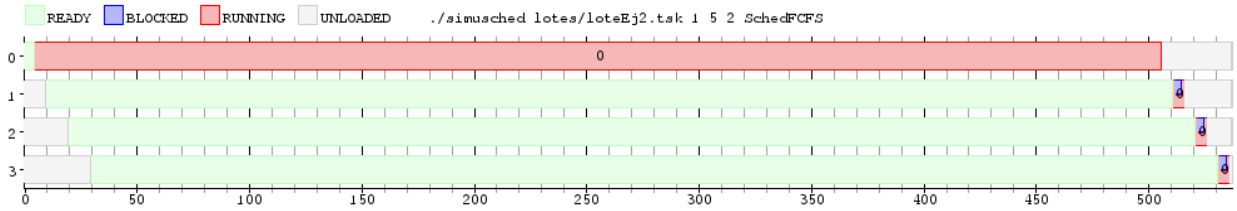

Figura 2

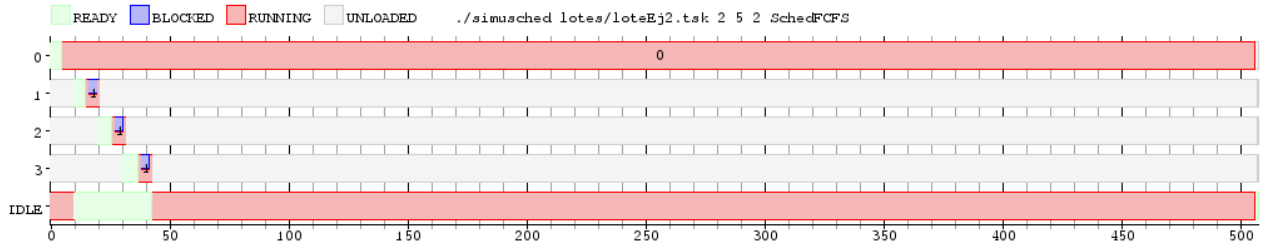

Figura 3

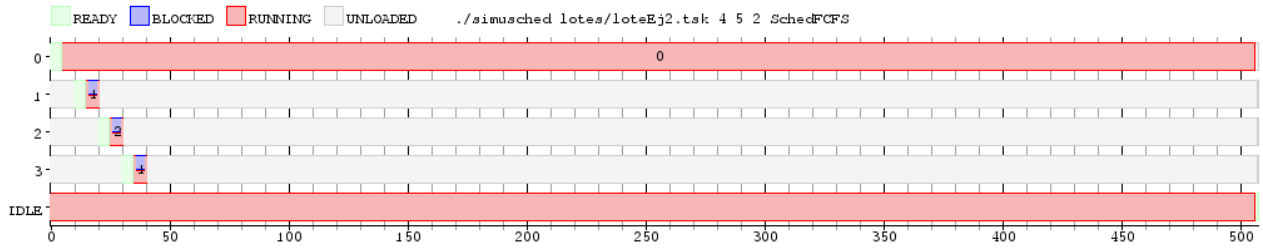

Figura 4

	Latencia (ciclos)		
#Procesadores	1	2	4
Tarea 0	5	5	5
Tarea 1	501	5	5
Tarea 2	502	6	5
Tarea 3	502	7	5
Promedio	377.5	5.75	5

Tabla 1: Latencia de cada tarea y latencia promedio según la cantidad de procesadores utilizados.

Viendo la tabla 1 resulta clara la principal desventaja de mantener esta política de *scheduling* con un solo procesador: la latencia de los procesos interactivos resulta altísima debido a que deben esperar a que termine el proceso intensivo en CPU, a pesar de que ellas mismas apenas necesitan utilizarlo. Una diferencia de tiempo tan grande entre que las tres tareas se carguen

y efectivamente responden (como puede apreciarse en la figura 2) genera para los usuarios la sensación de que "la máquina se colgó". De hecho cabe la posibilidad de que si un usuario le pidió alguna información al sistema como parte de un protocolo, al momento de recibirlo esta deje de ser útil o válida.

Como contrapartida, vemos que el tener 4 procesadores no aporta grandes ventajas sobre tener 2, pues las diferencias de latencia son marginales como puede verse en la tabla, y de hecho uno de los procesadores se desperdicia completamente realizando la tarea *idle* como se aprecia en la figura 4.

3. Ejercicio 3

Creamos la función `TaskBatch` en el archivo `task.cpp`. Nuevamente la implementación es muy sencilla.

Tenemos una variable `tiempo_disponible`, que guarda el tiempo disponible de la tarea para ser usado en el CPU sin contar el tiempo que se necesita para lanzar las llamadas bloqueantes (el cual es de un ciclo por llamada).

Si los parámetros son coherentes, es decir que no se pide hacer más llamadas bloqueantes de las que es posible con el tiempo dado, entonces se itera sobre la cantidad de bloqueos, pasado por parámetro. En cada iteración se decide, de forma pseudo-aleatoria, cuánto del tiempo disponible del CPU se va a utilizar antes de hacer el respectivo bloqueo. Una vez usado el CPU por dicha cantidad de tiempo, se procede a actualizar `tiempo_disponible` y se lanza el pedido de *I/O*.

Una vez que se realizaron todos los bloqueos, se gasta en el CPU el tiempo disponible que pudiera haber sobrado (de no haberlo simplemente se prosigue con la finalización del proceso).

La figura 5 ilustra un lote de 4 tareas de este tipo.

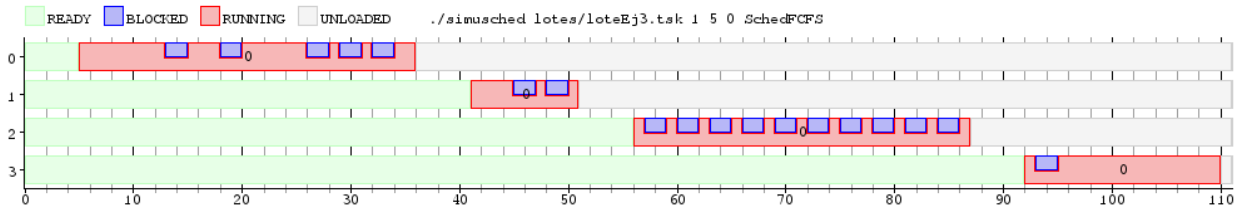

Figura 5

4. Ejercicio 4

A continuación detallamos los atributos privados de la clase `SchedRR`:

- `cant_cores`: almacena la cantidad de procesadores que tiene el sistema.
- `cola_procesos`: la cola *FIFO* en la cual se almacenan todos los procesos que están cargados, listos para ejecutar (o sea, en estado *ready*). Dicha cola es global para todos los procesadores.
- `quantum_original_cpu`: es un vector de longitud `cant_cores`, tal que `quantum_original_cpu[i]` indica la duración de un *quantum* del procesador `i`.
- `quantum_restante_cpu`: otro vector de longitud `cant_cores`, tal que `quantum_restante_cpu[i]` indica cuántos ciclos le quedan al proceso corriendo en el procesador `i` antes de agotar su *quantum*. En el caso en que se esté corriendo la tarea *idle* este valor no representa nada (pues tal tarea debe correr indefinidamente hasta que aparezca una nueva tarea para ser ejecutada).

Además, la clase cuenta con una función auxiliar, `int next(int cpu)`, que se encarga de devolver el *pid* del siguiente proceso a ejecutar, removiéndolo de la cola de procesos y reiniciando el *quantum* disponible para el proceso que llega. Notar que en caso de que no hayan procesos en *ready* los últimos dos pasos no se ejecutan y simplemente se devuelve el *pid* de la tarea *idle*.

La clase posee los siguientes métodos públicos:

- `void load(int pid)`: se encarga de cargar el proceso identificado por `pid`. Notar que esto simplemente consiste en agregarlo a la cola. Luego, eventualmente se ejecutará en un tick de reloj.
- `void unblock(int pid)`: vuelve a cargar una tarea que dejó de estar bloqueada, lo que consiste simplemente en llamar a la función `load`.
- `int tick(int cpu, const enum Motivo m)`: esta función se divide en tres casos según el motivo con el que se la haya llamado. Tanto en el caso en que la tarea que corría en `cpu` se haya bloqueado como en el que terminó hacemos lo mismo: sencillamente ponemos a correr a la siguiente tarea disponible (o a *idle* en caso de no haber ninguna), dejando a la tarea actual fuera del ciclo de ejecución (temporalmente en un caso, permanentemente en el otro). Si no sucedieron ninguna de las dos cosas entonces tenemos nuevamente tres posibles escenarios:

  1. La tarea actual es *idle*, en cuyo caso solo queda llamar a `next` y devolver su resultado.
  2. La tarea actual no es *idle* pero agotó su *quantum*, por lo que la volvemos a encolar (pues aún no ha terminado) y llamamos a `next`. Si no había otra tarea se seguirá ejecutando la misma durante otro *quantum*.
  3. La tarea actual ni es *idle* ni terminó su *quantum*, así que debe seguir ejecutando pero reducimos en 1 la cantidad de ciclos restantes.

5. Ejercicio 5

En las figuras 6, 7 y 8, vemos los gráficos obtenidos al simular el lote pedido en el enunciado para cada uno de los casos. Las tablas resumen las métricas solicitadas.

Figura 6

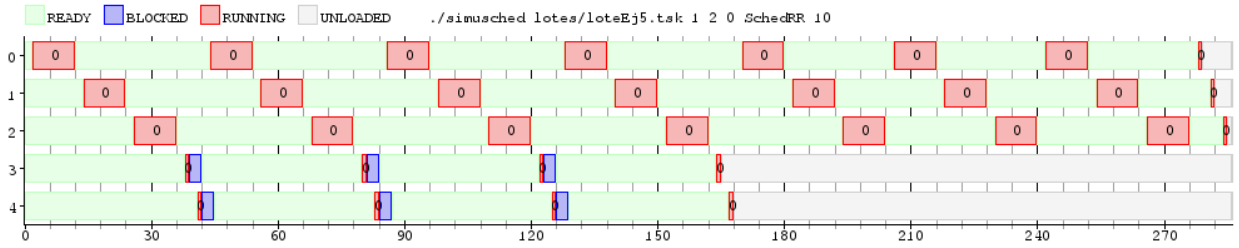

Figura 7

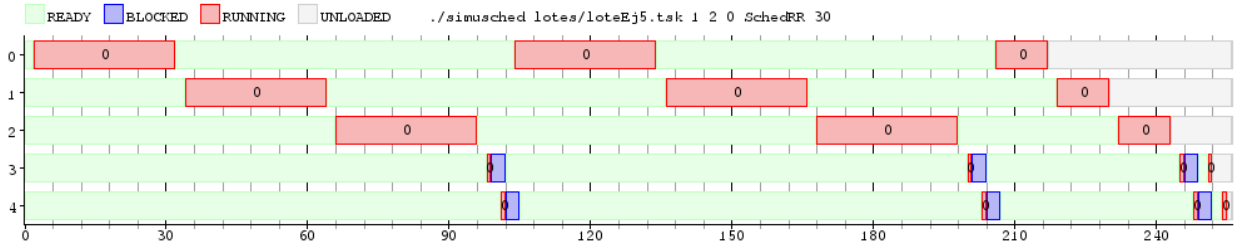

Figura 8

	Latencia (ciclos)		
Quantum	2	10	30
Tarea 0	2	2	2
Tarea 1	6	14	34
Tarea 2	10	26	66
Tarea 3	14	38	98
Tarea 4	17	41	100
Promedio	9.8	24.2	60

Tabla 2: Latencia de cada tarea y latencia promedio según la duración del *quantum* utilizado.

	Waiting time (ciclos)		
Quantum	2	10	30
Tarea 0	377	209	147
Tarea 1	380	212	160
Tarea 2	383	215	173
Tarea 3	60	156	243
Tarea 4	63	159	246
Promedio	252.6	190.2	193.8

Tabla 3: Tiempo total de espera de cada tarea y promedio según la duración del *quantum* utilizado.

	Turn-around (ciclos)		
Quantum	2	10	30
Tarea 0	447	279	217
Tarea 1	450	282	230
Tarea 2	453	285	243
Tarea 3	69	165	252
Tarea 4	72	168	255
Promedio	298.2	235.8	239.4

Tabla 4: Tiempo total de ejecución de cada tarea (*turn-around*) y promedio según la duración del *quantum* utilizado.

Se puede notar que, a la hora de optimizar la latencia, el *quantum* que mejor lo hace es el de 2 ciclos, y a medida que aumenta empeora la métrica. Esto es totalmente razonable puesto que los procesos que están en la cola de espera tienen que esperar (valga la redundancia) a que los procesos que ejecutan antes completen sus *quantums* (o bien se bloqueen o terminen). Al tener un *quantum* pequeño, la espera de cada proceso para empezar a ejecutar también lo es.

En contraposición, el *quantum* de menor tamaño tiene un pobre desempeño a la hora de considerar el *waiting time* o el *turn-around*. [1] En efecto, si notamos que la duración del *quantum* es igual a la duración del cambio de contexto, no resulta sorprendente ver estos resultados: cerca de la mitad del tiempo total que se requiere para que el lote completo termine de procesar se gasta haciendo *contex-switch*. Para ser más exactos, en total se pasan 221 ciclos de reloj procesando tareas[2] y 232 haciendo cambios de contexto. Esto repercute fuertemente en el tiempo de espera de las tareas.

Para *quantums* de tamaño 10 y 30, el *waiting-time* resulta sigificativamente menor. Aunque, como puede verse en la tabla 3, el tiempo de espera promedio no varía demasiado entre ambos casos, un análisis más detallado nos permite ver que la composición de dichos promedios si que es distinta. Si consideramos las tareas 0, 1 y 2 (que son las que solo consumen CPU en gran cantidad) vemos que su tiempo de espera se reduce notablemente al incrementar el *quantum* de 10 a 30. Sin embargo, en contraste las tareas 3 y 4 (que apenas usan el CPU para realizar llamadas de I/O) ven incrementados sus tiempos de espera en forma aún más significativa. Esto se explica porque mientras que por un lado el aumento del *quantum* no beneficia a las

[1] Notar que el segundo es consecuencia directa del primero, pues el tiempo total es la suma del tiempo efectivamente utilizado en la CPU más el tiempo de espera. Entonces nos centraremos principalmente en el *waiting-time*.

[2] Tres tareas usan el CPU 70 ciclos, y las otras dos usan 3 ciclos para hacer tres llamadas de I/O. Además, hay que considerar que cada tarea gasta un ciclo extra para terminar. Eso nos da $70 \times 3 + 3 + 3 \times 2 + 2 = 221$

tareas bloqueantes (pues las mismas solo requieren un ciclo), sí lo hace para las tareas de alto consumo de CPU. Pero este beneficio perjudica indirectamente al *waiting-time* del resto de las tareas, incluso al de otros procesos de CPU, aunque en este caso queda absorbido por el propio beneficio.

De hecho, los procesos bloqueantes muestran su menor tiempo de espera cuando el *quantum* es de 2 ciclos. Sin embargo, como ya vimos el dimensionamiento del *quantum* respecto del cambio de contexto es tan malo que en términos generales no tiene sentido considerar esta opción, y aún en caso de estar en una situación donde atender *I/O* rápido sea prioritario, es preferible explorar otras alternativas de *scheduling* que aprovechen mejor el procesador (como *multilevel feedback-queue scheduling*). Entre los *quantums* de 10 y 30 ciclos puede preferirse uno u otro según la importancia relativa que se le asigne a atender rápido procesos bloqueantes y terminar pronto con procesos largos.

6. Ejercicio 6

En la figura 9 se ilustra el mismo lote que en el punto anterior, pero ahora usando el *scheduler FCFS*.

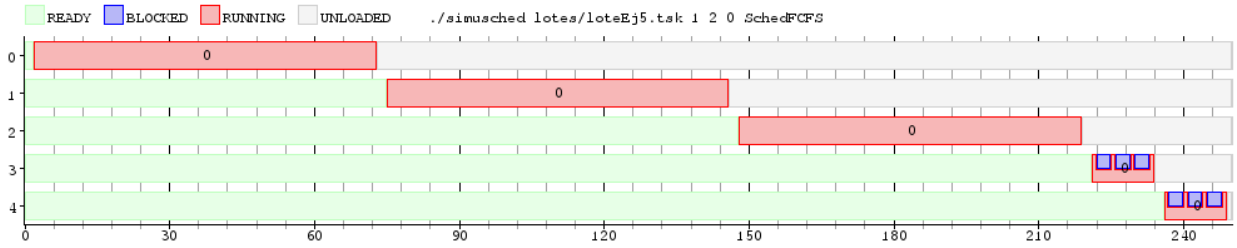

Figura 9

Una de las diferencias entre ambos schedulers es que el tiempo total que se requiere para ejecutar todo el lote es siempre menor con una estrategia FCFS. Esto se debe a que este algoritmo de *scheduling* realiza la mínima cantidad de cambios de contexto posibles (uno por cada proceso involucrado).

Al mismo tiempo empeora mucho la latencia de las tareas respecto a *Round Robin*. Además, con FCFS, esta métrica depende fuertemente del orden de llegada de los procesos, pues hay una gran diferencia entre que lleguen las tareas en orden creciente de tiempo de ejecución (mejor caso), y lo opuesto (peor caso, figura 9). En ese sentido, *Round Robin* es más robusto pues la latencia de los procesos depende mucho menos del orden de llegada y más de la duración del *quantum* (lo que resulta mucho más manejable).

En la figura 10 se grafica el mismo lote de tareas para FCFS pero cambiando el orden de llegada por el de mejor caso. Una buena forma de notar la diferencia en las latencias respecto del gráfico 9 es comparar la latencia del proceso con máxima latencia de cada gráfico, es decir el último en ejecutar en cada caso.

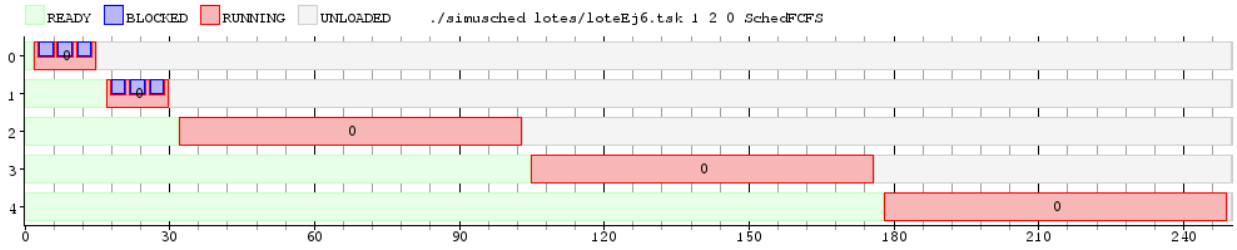

Figura 10

7. Ejercicio 7

Después de varias pruebas descubrimos que el comportamiento de `SchedMistery` tiene el compartamiento de un *multilevel feedback-queue*. Los parametros pasados son los quantums de las colas en orden de mayor a menos prioridad y tiene por defecto una cola de quantum 1 que es la de mayor prioridad.
Detallamos los atributos privados y publicos de la clase `SchedNoMistery` donde replicamos el comportamiento de `SchedMistery` Atributos privados:

- `vd`: Es un vector que tiene las colas de prioridad en orden, la de mayor prioridad en el 0 y la de menor al final.
- `def_quantum`: Tiene el quantum de cada una de las colas en vq. La cola en el subíndice i de vq tiene su respectivo quantum en el subíndice i de def_quantum.
- `unblock_to`: Un vector hay un subíndice para cada proceso que tenga el procesador. Cuando un proceso se bloquea guarda en el subíndice pid la prioridad que le toca al desbloquearse. Esto funciona ya que los id de procesos empiezan en 0 y aumentan de a uno a medida que llegan.
- `quantum`: El quantum que le queda a el proceso que esta corriendo.
- `n`: La cantidad de colas que tiene el scheduler.
- `cur_pri`: La prioridad de la proceso que se esta corriendo

La clase tiene una función privada, `int next()`, que se encarga de devolver el *pid* del siguiente proceso a ejecutar, buscado, desde la cola de mayor prioridad hasta la de menor, la primera cola vacia donde remueve el primer procesos, reinicia el *quantum* y actualiza *cur_pri*. En caso de que todas las colas esten vacias devuelve el *pid* de la tarea *idle*.
Además posee los siguientes métodos públicos:

- `SchedNoMistery(vector<int>argn)`: El constructor lee los quantum de las colas pasados como parámetros y los coloca en orden en def_quantum y genera una cola vacia para cada cola en vq, para así poder acceder más tarde. Además inicializa cur_pri con 0, quantum como 1 y n como la cantidad de colas.
- `void load(int pid)`: Carga el proceso identificado por `pid` en la cola de mayor prioridad.
- `void unblock(int pid)`: Agrega el proceso con id `pid` en la cola de prioridad indicada por el contenido del subíndice pid de `unblock_to`.
- `int tick(int cpu, const enum Motivo m)`: Tiene tres casos según el motivo con el que se la haya llamado:
  1. `Caso EXIT`: Simplemente llamamos a `next()` para que corra el siguiente proceso.
  2. `Caso BLOCK`: Guardo en `unblock_to` en la posición del id del proceso que se corre, la prioridad actual menos uno. Después se llama a `next()` para que pueda correr el siguiete proceso.
  3. `TICK`: Pueden pasar varias cosas:
     a) La tarea actual es *idle*, en cuyo caso solo queda llamar a `next` y devolver su resultado.

b) La tarea actual no es *idle* pero se acabó su *quantum*, por lo que hay que encolarla en la siguiente cola con menor prioridad y llamamos a `next`.

c) La tarea actual ni es *idle* ni terminó su *quantum*, así que debe seguir ejecutando pero reducimos en 1 la cantidad de ciclos restantes.

8. Ejercicio 8

8.1. Introducción

En este ejercicio se nos propone implementar un scheduler *Round-Robin* que no permita migración de procesos (al contrario del *Round-Robin* del ejercicio 4), y cumpliendo una serie de otras características, como que la asignación de CPU se realiza al momento que se carga el proceso, y que se elige a la CPU que tiene menor cantidad de procesos activos totales (RUNNING + BLOCKED + READY).

8.2. Estructura Interna

Para resolver el problema, elegimos la siguiente estructura interna:

- `cant_cores`: Un número entero que indica la cantidad de núcleos del procesador.
- `cola_procesos_cpu`: Un vector de colas de enteros que guarda, para cada cpu (numeradas del 0 a cant_cores-1), su correspondiente cola de procesos en el estado de ready.
- `quantum_original_cpu`: Un vector de enteros que guarda, para cada cpu, su correspondiente quantum. Notar que esta variable no se modifica nunca, ya que cada cola tiene su quantum fijado desde el inicio de los tiempos.
- `quantum_restante_cpu`: Un vector de números enteros que contiene, para cada cpu, el quantum restante que le queda al proceso actual que actualmente ejecuta dicha cpu. En cada tick irá decreciendo hasta llegar a 0.
- `cant_procesos_cpu`: Un vector de enteros que guarda, para cada cpu, la cantidad de procesos activos totales (RUNNING + BLOCKED + READY).
- `procesos_por_nucleo`: Un diccionario de enteros a enteros que establece una relación entre el pid y la cpu que la "acogió" como su hija, y por lo tanto, a ella le pertenece. El pid es la clave, mientras que el número de cpu es el significado.

Además, la clase cuenta con una función auxiliar, `int next(int cpu)`, que se encarga de devolver el *pid* del siguiente proceso a ejecutar, removiéndolo de la cola de procesos y reiniciando el *quantum* disponible para el proceso que llega. Notar que en caso de que no hayan procesos en *ready* los últimos dos pasos no se ejecutan y simplemente se devuelve el *pid* de la tarea *idle*.

8.3. Funcionamiento

Como todo scheduler, la clase posee los siguientes métodos públicos:

- `SchedRR2(vector<int>argn)`: El constructor de la clase. Se encarga básicamente de inicializar la estructura interna para que tenga sentido (o sea, que se cumpla el invariante de representación). No encontramos nada importante a destacar en este método más que aclarar que el vector *cant_procesos_cpu* se llena con ceros ya que al inicio todas las cpu tienen 0 procesos activos totales, y que el vector *cola_procesos_cpu* se llena con colas de enteros vacías.
- `void load(int pid)`: Se encarga de cargar el proceso identificado por el pid recibido por parámetro. Pueden haber dos escenarios en la carga de un proceso:

1. Que sea un nuevo proceso que se esté cargando, en cuyo caso se debe elegir una de las cpu's que tienen menor cantidad de procesos activos totales para "apadrinar.ª este nuevo proceso entrante, sumarle 1 a la cantidad de procesos de ésa cpu, agregar ese proceso a la cola de procesos de ésa cpu, y por último agregar la relación pid-cpu en el diccionario *procesos_por_nucleo*.
2. Que sea un proceso existente que se acaba de desbloquear, en cuyo caso se debe obtener la cpu a la cual pertenece este proceso y agregarlo a la cola de procesos de la misma.

- `void unblock(int pid)`: Se vuelve a cargar la tarea que dejó de estar bloqueada, lo que consiste simplemente en llamar a la función `load`.
- `int tick(int cpu, const enum Motivo m)`: Esta función se divide en tres casos dependiendo de qué ocurrió en el último tick ejecutado por la cpu:

  1. TICK: En el caso de que haya ejecutado un tick entero, pueden ocurrir dos cosas: 1) Usó todo su quantum; 2) No usó todo su quantum. En el caso 1 hay que encolar el proceso actual y traer al siguiente proceso a la cpu invocando a la función `next(cpu)`. En el caso 2 sólo hay que restarle 1 al quantum restante de la cpu y devolver el pid de la tarea actual, ya que es a la que le toca seguir ejecutando.
  2. BLOCK: En el caso de que la tarea se haya bloqueado sólo hay que llamar a la función `next(cpu)` explicada en la sección "Estructura Interna".
  3. EXIT: En el caso de que la tarea haya finalizado, hay que restarle 1 a la variable que guarda la cantidad de procesos activos totales de cada cpu (ya que ahora este proceso no forma parte de los procesos activos totales de la cpu), borrar del diccionario el pid del proceso que acaba de finalizar e invocar a la función `next(cpu)` para cargar un nuevo proceso a la cpu.

  Si por algún error misterioso el motivo no es ninguno de los descriptos anteriormente, se procederá a enviar un mensaje de error por el standard error.

8.4. Análisis de las métricas

waiting time empeora con migracion de procesos xq hay un tiempo de cambio de procesador que es tiempo muerto.

para las tareas muy interactivas que requieran bloquearse

HAY QUE HACER TODA LA PARTE IMPORTANTE, O SEA EXPLICAR EN QUE CASOS ESTA BUENO MIGRACION DE PROCESOS Y EN QUE CASOS NO, Y DAR LOTES Y GRAFICAR Y MOSTRAR LAS METRICAS QUE MEJORAN O EMPEORAN EN CADA CASO.