Sistemas Operativos

Segundo Cuatrimestre de 2015

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo practico 1

Integrante	LU	Correo electrónico
Integrante Ignacio Rodriguez	797/13	igna_nacho286@hotmail.com
Integrante Nicolás Mastropasqua	828/13	me.nicolas7@gmail.com
Integrante Uriel Fadel	104/14	urielfadel@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Ejercicio 1:

1.1 Enunciado:

Programar un tipo de tarea TaskConsola, que simulará una tarea interactiva. La tarea debe realizar n llamadas bloqueantes, cada una de una duración al azar 1 entre bmin y bmax (inclusive). La tarea debe recibir tres parámetros: n, bmin y bmax (en ese orden) que serán interpretados como los tres elementos del vector de enteros que recibe la función. Explique la implementación realizada y grafique un lote que utilice el nuevo tipo de tarea.

1.2 Resolución:

Tarea Consola: Esta tarea recibe tres parámetros: n, bmin y bmax. A partir de esto genera n llamadas bloqueantes con duración al azar(entre bmin y bmax).

Para ello, hacemos un ciclo cuya duración esta indicada por el primer elemento del vector de parametros(n). En el cuerpo del ciclo, creamos un numero pseudo-aleatorio usando la función **rand()** y llamamos a **uso_IO** con el mismo. Por defecto, el numero devuelto se encuentra en el rango $[0, rand\_MAX]$. Para que quede definido en $[bmin, bmax]$ basta con lo siguiente:

Sea $rango = bmax - bmin + 1$

$0 \quad \leq \quad$ **rand()** mod $rango \; < rango = bmax - bmin + 1$

$bmin \leq \quad$ **rand()** mod $rango \; + \; bmin \; < bmax \; - \; bmin \; + 1 \; + \; bmin \; = \; bmax \; + 1 \leq bmax$

Además **rand()** generará siempre la misma sucesión de números pseudo-aleatorios en cada nueva ejecución.

Exactamente esta operatoria es la que se hace en el codigo.

1.3 Gráficos:

A continuación se muestra un gráfico representando la ejecución de un lote de cuatro tareas: Las de ID cero y uno del tipo consola, la de ID 2 del tipo **uso_cpu** y la ultima del tipo **uso_IO**. Todas son manejadas por un scheduler *FCFS(First Come First Served)*. Todas se van incorporando al scheduler con 2 clocks de diferencia. Notese que, por ejemplo, la primer tarea fue corrida con una duración de 5 clocks(+1) y un rango de $[2, 7]$. Se pueden identificar las 5 llamadas, de duración 2,5,3,6,2 en esa secuencia.

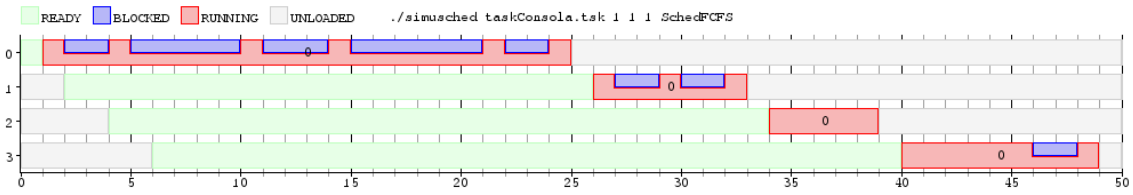

Figure 1: FCFS con dos TaskConsola, una TaskCPU y otra TaskIO

2 Ejercicio 2:

2.1 Enunciado:

Rolando, uno de los investigadores del departamento, utiliza su computadora para correr un algoritmo muy complejo que hace un uso intensivo de la CPU por 100 ciclos y no utiliza ninguna llamada bloqueante. Mientras corre su algoritmo suele poner su canción preferida y luego navegar por internet (estas tareas realizan 20 y 25 llamadas bloqueantes respectivamente con una duración variable entre 2 y 4 ciclos). Escribir el lote de tareas que simule la situación de Rolando. Ejecutar y graficar la simulación usando el algoritmo FCFS para 1 y 2 núcleos con un cambio de contexto de 4 ciclos. Calcular la latencia de cada tarea en los dos gráficos. Explicar qué desventaja tendría Rolando si debe mantener este algoritmo de scheduling y sólo tiene disponible una computadora con un núcleo,

2.2 Resolución:

Para simular esta situación con el scheduler **FCFS**,se armo un lote de tareas como se muestra en la tabla de abajo. Decidimos dejar un tiempo prudencial entre cada una de ellas para cumplir, de cierta forma, con la serializacion que ,entendemos, describe en el problema(Rolando corre su algoritmo, pone su canción y después navega por internet). El mismo lote se utilizo para ambos casos.

Tarea	TiempoLlegada
TaskCPU	0
TaskConsola	20
TaskConsola	40

Primer Caso:

A continuación se muestra la tabla que se obtiene de simular la situación planteada para un solo núcleo. Los resultados se condicen con el gráfico de la figura 2

Tarea	Latencia(en ciclos)
TaskCPU	4
TaskConsola	89
TaskConsola	150

Los valores exactos fueron extraídos desde la ejecución en consola

Segundo Caso:

A continuación se muestra la tabla que se obtiene de simular la situación planteada para dos núcleos. Los resultados se condicen con el gráfico de la figura 3

Tarea	Latencia(en ciclos)
TaskCPU	4
TaskConsola	4
TaskConsola	65

Los valores exactos fueron extraídos desde la ejecución en consola

Gráficos:

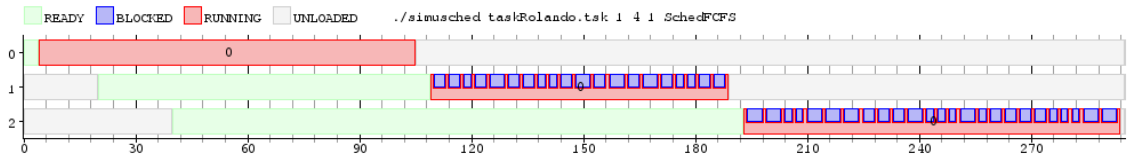

Figure 2: 1 core

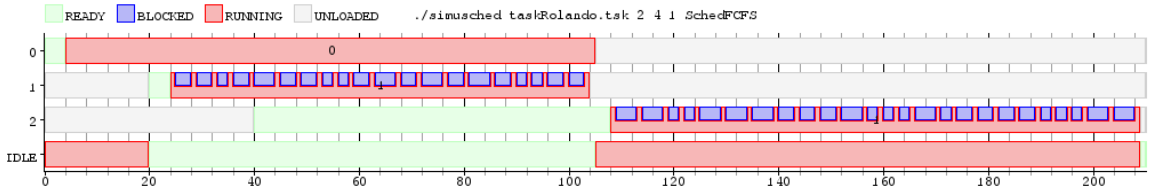

Figure 3: 2 core

En este escenario, se puede observar que la duración del lote de tareas en el caso de un solo núcleo(figura 2) finaliza después de los 290 ciclos, mientras que con dos(figura 3), no supera los 210. Esto se debe a la ganancia en la latencia del segundo caso con respecto al primero. Como se puede ver en el gráfico, y en los cálculos previos, la tarea 1 llega en el ciclo 20 y solo pasa 4 en estado ready pues luego es "recibida" por el segundo núcleo, que hasta el momento se encontraba ejecutando la tarea IDLE(desde 0 a 20 ciclos, como se puede apreciar) Esta situación seria imposible en el caso 1, ya que la tarea 1 debe esperar(por razones de la implementación FCFS y por la ausencia de un segundo núcleo) a que la tareaCPU(ID = 0) llegue a su fin(como se puede ver al cabo de los 105 ciclos).

Por estas razones es que ,también, la ultima tarea(ID = 2) reduce el tiempo de latencia, aproximadamente acla mitad, en el caso 2, ya que ahora a los 105 ciclos, el primer núcleo se libera de la tareaCPU y luego del context switch de 4 ciclos, se comienza a ejecutar. Es importante marcar que las tareas que simulan la reproducción de una canción y la navegación, realizan una cantidad de llamadas bloqueantes de duración entre 2 y 4 ciclos, durante las cuales el flujo de ejecución queda estático, es decir es tiempo que el cpu "desperdicia" , a diferencia de Round Robin, donde se cambiaría inmediatamente a otro proceso.

Notese que hay un ciclo adicional por tarea, siendo este la llamada a la syscall exit().

3 Ejercicio 3:

3.1 Enunciado:

Programar un tipo de tarea TaskBatch que reciba dos parámetros: total cpu y cant bloqueos. Una tarea de este tipo deberá realizar cant bloqueos llamadas bloqueantes, en momentos elegidos pseudo aleatoriamente. En cada tal ocasión, la tarea deberá permanecer bloqueada durante exactamente un (1) ciclo de reloj. El tiempo de CPU total que utilice una tarea TaskBatch deberá ser de total cpu ciclos de reloj (incluyendo el tiempo utilizado para lanzar las llamadas bloqueantes; no así el tiempo en que la tarea permanezca bloqueada). Explique la implementación realizada y grafique un lote que utilice 3 tareas TaskBatch con parámetros diferentes y que corra con el scheduler FCFS.

3.2 Gráficos:

El siguiente lote de tareas es el utilizado para graficar como funciona la tarea taskBatch.

Tarea	TotalCpu	CantBloqueos
TaskBatch	20	2
TaskBatch	60	9
TaskBatch	22	5

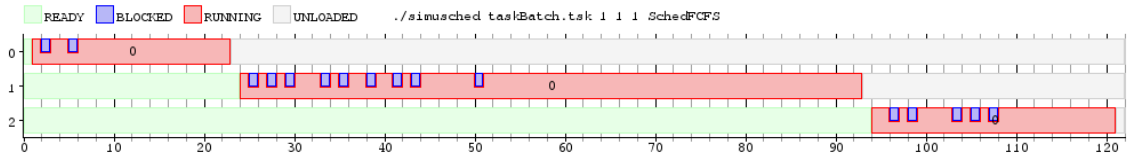

Figure 4:

3.3 Resolución:

La forma de solucionar el ejercicio fue planteada de manera que el tiempo total de cpu, que se le pasa por parámetro, es la cantidad de tiempo que esta en Running el procesador, mas el ciclo de return que se corre al terminar cada proceso. La forma en que lo pensamos fue tener un array, del tamaño total de cantidad de ciclos que necesita el procesador para terminar, y luego fijarse cuantos bloqueos necesita.

Entonces se va a ciclar sobre todos los posibles valores donde puede llegar a bloquearse, y se van a elegir la cantidad de bloqueos necesarios, distribuidos aleatoriamente en ese array, y luego ese va a ser el que decide ,para cada momento, si el proceso va a estar corriendo o si va a estar bloqueado. Como lo corremos con el scheduler **First Come First Served**, sabemos que no se le va a quitar nunca el privilegio del procesador, entonces el tiempo total que va a correr un lote de tareas ,de este tipo, va a ser el tiempo total de Cpu pasado por parámetro (+1) por cada tarea, mas la cantidad de veces que se va a bloquear. La forma de decirle cuando corre Cpu, o cuando es I/O, es simplemente recorrer el array, y verificar si antes se eligió poner un bloqueo o un ciclo de CPU. Como no hay desalojamiento, entonces no hay cambio de contexto, salvo cuando se termina cada proceso.

4 Ejercicio 5:

4.1 Enunciado:

Diseñe un lote con 3 tareas de tipo TaskCPU de 50 ciclos y 2 de tipo TaskConsola con 5 llamadas bloqueantes de 3 ciclos de duración cada una. Ejecutar y graficar la simulación utilizando el scheduler Round-Robin con quantum 2, 10 y 50. Con un cambio de contexto de 2 ciclos y un sólo núcleo calcular la latencia, el waiting time y el tiempo total de ejecución de las cinco tareas para cada quantum. ¿En cuál es mejor cada uno? ¿Por qué ocurre esto?

4.2 Resolución:

Primero, hacemos algunas aclaraciones pertinentes.En el lote implementado, todas las tareas llegan al sistema en el ciclo inicial. En las tareas del tipo TaskConsola, el waiting time (por definición los ciclos que pasa la tarea en estado *ready*), no incluye los ciclos que se encuentra en estado bloqueada. Para el running time , consideramos la cantidad de ciclos que le toma a la tarea ejecutarse, es decir desde que recibe por primera vez el cpu, hasta que finaliza por completo.

Primer Caso: Quantum 2

A continuación se muestra la tabla que se obtiene de simular la situación planteada. Los resultados se condicen con el gráfico de la figura 5

IDtarea	Latencia(en ciclos)	Waiting time	Running time
Tarea0	2	288	338
Tarea1	6	292	337
Tarea2	10	296	346
Tarea3	14	79	92
Tarea4	17	82	92

Los valores exactos fueron extraídos desde la ejecución en consola

Segundo Caso: Quantum 10

A continuación se muestra la tabla que se obtiene de simular la situación planteada. Los resultados se condicen con el gráfico de la figura 6

IDtarea	Latencia(en ciclos)	Waiting time	Running time
Tarea0	2	167	212
Tarea1	14	170	203
Tarea2	26	173	194
Tarea3	38	206	185
Tarea4	41	209	185

Los valores exactos fueron extraídos desde la ejecución en consola

Segundo Caso: Quantum 50

A continuación se muestra la tabla que se obtiene de simular la situación planteada. Los resultados se condicen con el gráfico de la figura 7

IDtarea	Latencia(en ciclos)	Waiting time	Running time
Tarea0	2	114	164
Tarea1	54	117	115
Tarea2	106	121	66
Tarea3	158	178	40
Tarea4	161	181	41

Los valores exactos fueron extraídos desde la ejecución en consola

4.3 Gráficos:

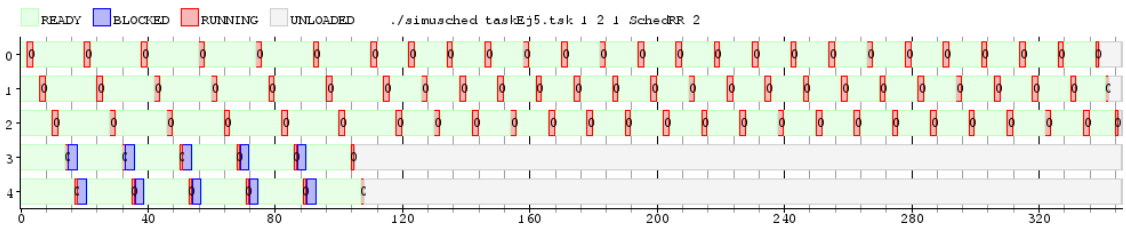

Figure 5: Quantum = 2

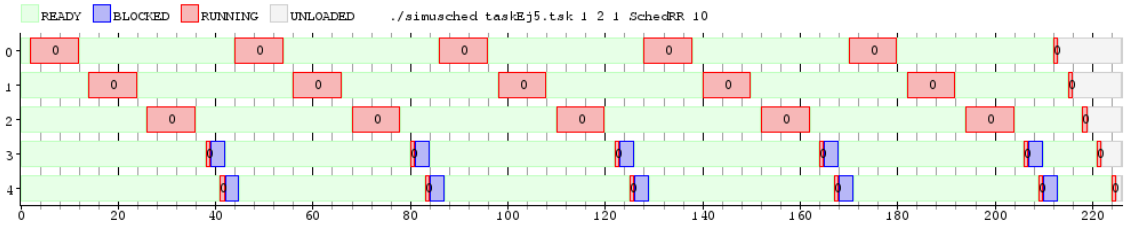

Figure 6: Quantum = 10

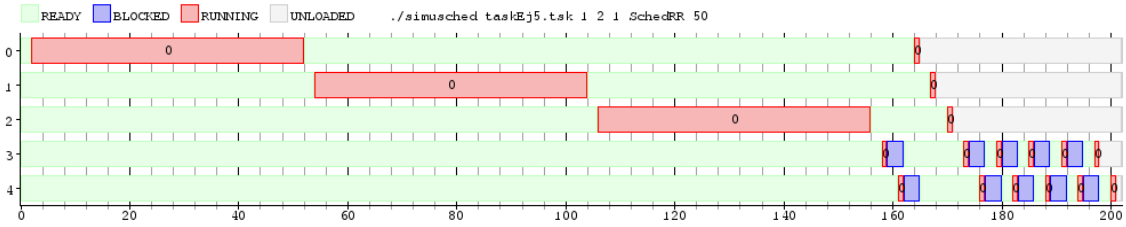

Figure 7: Quantum = 50

¿En cual caso es mejor cada uno de los parámetros medidos? ¿Por que ocurre esto?

Los cálculos, basados en los datos que arroja el simulador y respaldados por lo que se observa en el gráfico, nos permitirán sacar conclusiones sobre los aspectos medidos: Latencia, Waiting Time y Running Time. En el caso de la latencia, se ve que aumenta conforme lo hace el quantum. Esto se debe a que , asumiendo que todas las tareas llegan en el instante inicial como lo hacen en los graficos, las mismas deben esperar rondas de mayor tiempo para poder comenzar a ejecutar. Tambien se podría observar, por la misma razon, que si el costo del cambio de tareas aumentase, la latencia tambien lo hará.

Sin embargo, el waiting time decrece ,para todos los procesos ,del tipo TaskCPU, si el Quantum lo hace. Si se define el overhead producido por los context switch(son tiempos de procesamiento "desperdiciado") como:

$$\textbf{context switch overhead} = \frac{C}{(Q+C)}$$

Donde Q = quantum y C = costo del context switch

Podemos ver que si aumentamos Q, con el C, estaremos disminuyendo el tiempo que se "desperdicia" en hacer los cambios entre tareas. Además, como otra consecuencia, se aumentará el tiempo que los procesos tienen a su disposición el cpu antes de ser desalojados,pero ,a su vez, incrementando el waiting time entre ronda. En este caso, la combinación resulta favorable para las tareas TaskCpu, no así para las tareas Consola, que para poder ejecutar las cinco llamadas, deberán esperar al menos 5 rondas, cuya duración aumento con respecto a las de Quantum 2.

Cuando subimos el quantum a 50, observamos una situación particular. Cada tarea TaskCpu puede terminar en una ronda, por lo tanto se reducen drásticamente la cantidad de ciclos desperdiciados en los cambios de contexto, al mismo tiempo que se reduce el Waiting time, en detrimento de la latencia, como se explico anteriormente. Este escenario genera, entonces, un tiempo de respuesta ,para los procesos que no son atendidos de inmediato(como las tarea 3 y 4), mucho mayor. Esta penalidad podria no ser deseable en situaciones donde se corren tareas interactivas. Se ve también que la situación para las tareas Consola mejora levemente, ya que el waiting time disminuye dado que luego de realizar las dos primeras llamadas, no hay mas taras Consola en la cola del scheduler(como se

observa en el gráfico, promediando el ciclo 180), por lo que pueden alternarse entre ellas y finalizar mas rápidamente las llamadas restantes. En conclusión, las tareas Consola obtuvieron su mejor waiting time con Quantum 2, donde el era lo suficientemente chico como para completar rápidamente sus llamadas, debido a que el tiempo esperado por ronda era menor. Esta situación, pudo verse también con Quantum = 50 sobre el final de la ejecución(por la particularidad que se explico anteriormente). Sin embargo, no fue óptimo porque también sufrieron un notable aumento en la latencia, con respecto al primer escenario.

Como consecuencia de lo anteriormente explicado es que se obtiene un running time óptimo, para estos casos, con Quantum = 50. Es decir, en este ultimo escenario ,el scheduler tuvo un desempeño similar al de un FCFS(ver figura 8) que, debido a la naturaleza de las tareas elegidas de este caso, minimiza el running time.

5 Ejercicio 6:

5.1 Enunciado:

Grafique el mismo lote de tareas del ejercicio anterior para el scheduler FCFS. Haciendo referencia a lo que se observa en los gráficos de este ejercicio y el anterior, explique las diferencias entre un scheduler Round-Robin y un FCFS.

5.2 Gráficos:

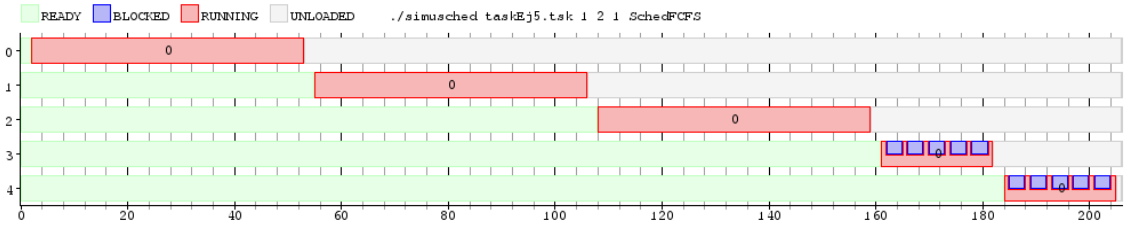

Figure 8: Ejercicio 5 con FCFS

5.3 Resolución:

La diferencia importante entre Round Robin, y First Come First Served, es el tiempo que se pierde entre cada cambio de contexto. Tiene un beneficio que es el hecho de que el tiempo en el que un proceso es bloqueado no pierde tiempo de procesamiento, pero sin embargo con lo que cuesta un cambio de contexto, y mas cuando los quantums son tan chicos y el cambio de contexto tan grande, se pierde mucho tiempo cada vez que necesita cambiar de tarea. Esto se puede ver en el gráfico de quantum 2, ya que tarda tanto tiempo en cambiar el contexto de lo que se tiene de procesador, sin embargo cuando vamos agrandando los quantums a 10 o 50, y se mantiene el tiempo de cambio de contexto de 2, entonces se aprovecha un poco mas el procesador y ahí si le sirve sacárselo en tiempo de bloqueo.

La mayor diferencia entre **FCFS**, y **Round Robin**, es el tiempo de latencia, ya que no se empezaran los siguientes procesos hasta no haber completado los anteriores, entonces se tarda demasiado en empezar a atenderlos. Sin embargo el waiting time de los primeros procesos es mas corta, pero la de los últimos dos es bastante mas grande, ya que tienen que esperar a que terminen los primeros procesos, que tardan bastantes ciclos. Al no perder tanto tiempo en cambio de contexto y al haber pocos bloques, o de poca duración, hace que el **FCFS** funcione mas rápido porque el tiempo de cambio de contexto es el mismo, y solo se pierde cuando termina cada uno de los procesos ya que no se les quita el procesador. El **FCFS** tendría mas problemas si los bloqueos serian de mayor tiempo.

Además, como contrapartida, esta política admite una mayor perdida de paralelismo en, al menos en este caso, en comparación con **Round Robin**

6 Ejercicio 8:

6.1 Enunciado:

Implemente un scheduler Round-Robin que no permita la migración de procesos entre núcleos (SchedRR2). La asignación de CPU se debe realizar en el momento en que se produce la carga de un proceso (load). El núcleo correspondiente a un nuevo proceso será aquel con menor cantidad de procesos activos totales (RUNNING + BLOCKED + READY). Explique un escenario real donde la migración de núcleos sea beneficiosa y uno donde no (mencione específicamente qué métricas de comparación vistas en la materia mejorarían en cada caso). Diseñe un lote de tareas en nuestro simulador que represente a cada uno de esos escenarios y grafique su resultado para cada implementación. Calcule y compare en cada gráfico las métrica que mencionó.

6.2 Gráficos:

A continuación se mostraran escenarios en los cuales la migración de núcleos resulta favorable y otros en los que no ocurre.

Caso 1: Ambos schedulers corren el siguiente lote de tareas, con el mismo quantum y cantidad de núcleos.

IDtarea	Tipo	Instante de llegada	Tiempo De Ejecución
Tarea0	TaskCpu	0	80
Tarea1	TaskCpu	5	5
Tarea2	TaskCpu	6	30

A continuación se muestra los cálculos,que se desprenden de los gráficos de la figura 9 y 10, para las tareas que nos interesa comparar.

Figura 9

IDtarea	Running Time	Waiting time
Tarea0	87	6
Tarea1	39	7

Throughput = $\frac{3}{88} = 0,034$

Figura 10

IDtarea	Running Time	Waiting time	Throughput
Tarea0	136	53	
Tarea1	84	52	

Throughput = $\frac{3}{135} = 0,022$

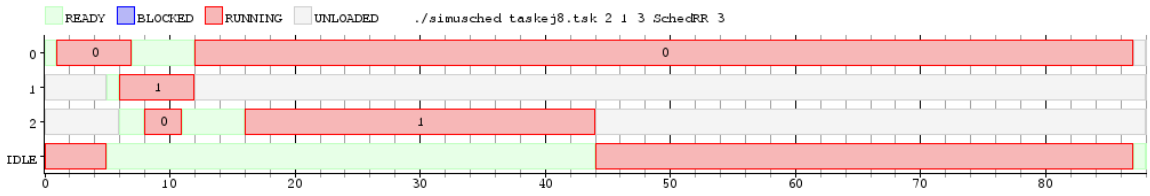

Figure 9: Situación favorable con migración. Q = 3, N = 2, Tmigracion = 3.

Figure 10: Situación desfavorable sin migración. Q = 3, N=2.

Este pequeño ejemplo sirve para ilustrar un caso en el que la migración de procesos resulta en un **running time** mejor para todos los procesos. No solo eso, sino que también reduce el **waiting time**. La particularidad que produce este comportamiento, es la llegada de una tarea(**Id1**), de una duración corta, justo antes de que comience a ejecutar la de **Id2**. Esto le otorgará el núcleo 1 a la misma, forzando a que la primer tarea(**Id0**) ceda el suyo(al finalizar el quantum) a la de **Id2**(ya que la de **Id1** aun no habrá terminado su quantum y no habrá libera el núcleo). Al finalizar el quantum, la tarea de **Id2** se desaloja en el core cero, e ingresa la de **Id0**. Sin embargo, como la tarea de **Id1** termina pronto, el núcleo 1 quedara disponible inmediatamente para que pueda ser usado por la de **Id2**. Como hay migración, la próxima ronda que le sea concedida a la tarea de **Id2** sera ejecutada en este núcleo(pues el cero esta ocupado con la de **Id0**). Esta situación, no hubiese posible de no existir esta cualidad en el scheduler, provocando así que cada tarea quede "fija" en su núcleo.Este hecho queda evidenciado en la figura 10. Como las mas largas fueron adjudicadas al núcleo cero inicialmente, como se ve en el gráfico,deberán continuar ejecutando en este, aunque el núcleo 1 se libere rápidamente.

Caso 2: Ambos schedulers corren el siguiente lote de tareas, con el mismo quantum y cantidad de núcleos.

IDtarea	Tipo	Instante de llegada
Tarea0	TaskAlterno	0
Tarea1	TaskAlterno	0
Tarea2	TaskCpu	8
Tarea3	TaskCpu	32
Tarea4	TaskCpu	48

A continuación se muestra los cálculos,que se desprenden de los gráficos de la figura 10 y 11, para las tareas que nos interesa comparar.

Figura 11

IDtarea	Running Time	Waiting time	Throughput
Tarea0	64	31	
Tarea1	60	25	

Throughput = $\frac{5}{65} = 0,022$

Figura 11

IDtarea	Running Time(Figura 12)	Waiting time(Figura 12	Throughput)
Tarea0	38	5	
Tarea1	40	5	

Throughput = $\frac{5}{65} = 0,022$

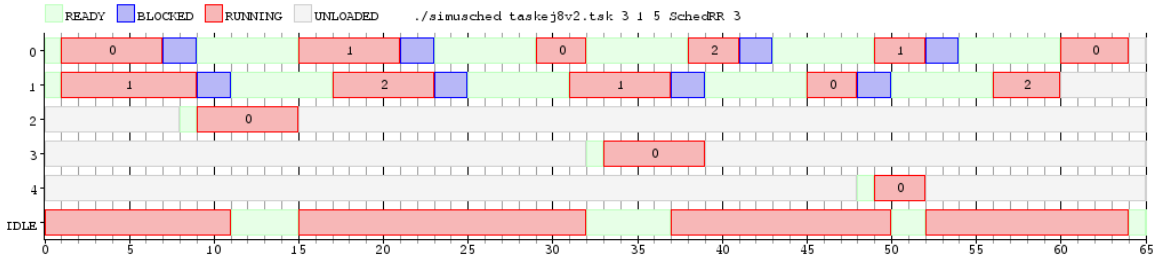

Figure 11: Situación desfavorable con migración. Q = 3, N = 3, Tmigracion = 5.

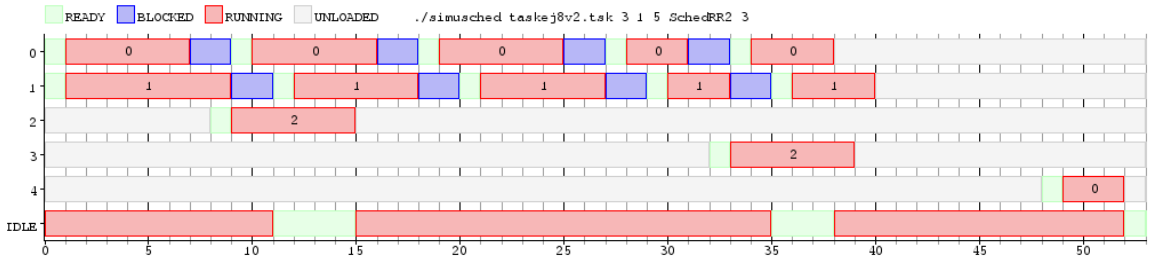

Figure 12: Situación favorable sin migración. Q = 3, N=3.

En este ejemplo,se ilustra un caso que desfavorece a la migración de núcleos. Notemos que existe una penalidad de tiempo(costo de migración)que se suma cada vez que un proceso cambia de núcleo. Al igual que el tiempo de cambio de tarea(context switch), es un intervalo en el que el cpu no hace ningún trabajo "útil" para algún proceso. La situación que se plantea, en este caso , muestra que el intercambio constante de núcleos, para un proceso, puede terminar siendo desventajoso. Básicamente, hay dos tareas, que se bloquean alternadamente con cierto desfase, con una duración mayor a las ultimas tres . Se da que cada que cuando la tarea de **id0** se encuentra bloqueada, llega una nueva tarea **TaskCpu**. En ese instante, se le asignará siempre el primer núcleo disponible(que es el cero, porque la tarea que estaba ejecutando se bloqueó). Esto provocará que entonces, cuando la primera retome la ejecución lo haga en el núcleo 1(pues el cero seguirá ocupado) y por esta misma razón, cuando la segunda vuelva a su estado de "ready" solo podrá tomar el núcleo 2(porque aun las otras tienen tomado los demás cores). Esta particularidad forzó que las dos primeras tareas migraran sus núcleos, pero no trajo ningún beneficio ,si no lo contrario. De esta manera, iran llegando otras dos tareas que surtan el mismo efecto sobre las dos primeras, las de **id0 y id1**.

Si se actúa sin migración, se da lo que uno esperaría en un principio. Cuando llega una tarea corta, como hay un núcleo disponible y las taras de **id1 y id2** están fijas en la cola de tareas del núcleo cero y uno respectivamente, se le otorga el core restante(el dos). Esto "paralelizara" las tareas de una forma mas eficiente, decrementando el **waiting time y running time**.

PUEDEN PONER LOS OTROS QUE HABIAN PENSADO, YO AGREGUE ESTOS DOS NADA MAS!!!

6.3 Resolución:

Para implementar el scheduler referido en el enunciado blablabla (Aca viene la explicacion del codigo. Basicamente mencionar como difiere de la otra implementacion)