DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

22 / 04 / 2015 Sistemas Operativos

Integrante	LU	Correo electrónico
Abdala, Leila	950/12	abdalaleila@gmail.com
Enrique, Natalia	459/12	natu_2714@hotmail.com
Salinas, Pablo	456/10	salinas.pablom@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. Introducción

En este informe presentaremos la implementación de diversos schedulers y una breve experimentación comparativa entre estos. La idea de este TP es conocer las distintas formas de administrar el scheduler, entendiendo asi que ventajas presenta cada modo y en que contexto. El desarrollo del informe se basa en el enunciado, por lo que cada item del mismo tiene en este informe una sección que responde al mismo.

2. Ejercicios

2.1. Ejercicio 1: TaskConsola

En éste ejercicio debemos programar una tarea *TaskConsola* la cual debe realizar n llamadas bloqueantes, cada una con una duración al azar entre bmin y bmax, ambas pasadas por parámetro.

Para ello, decidimos utilizar un contador de 0 hasta n y generar un número pseudo-alaeatorio por medio de la función *rand*. Es decir, cada vez que se aumenta el contador, realizamos una llamada bloqueante (uso_IO) que durará la cantidad de ciclos que se haya generado en la llamada a rand.

2.2. Ejercicio 2: Experimentando FCFS

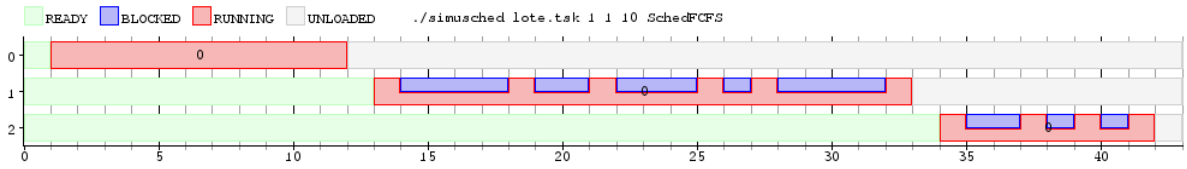
Para probar el Scheduler First Came First Served usaremos el siguiente lote de tareas:

TaskCPU 10
TaskConsola 5 1 5
TaskConsola 3 1 2

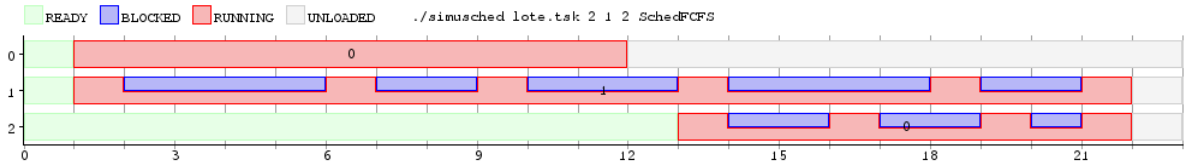
En la primera, utlizaremos TaskCPU y para darle uso intensivo correrá durante 10 ciclos de reloj. Las siguientes, son de tipo TaskConsola implementado anteriormente, pasándole como parámetro el número de llamadas bloqueantes y el rango en el cual debe seleccionar el número aleatorio.

En el algoritmo FCFS, la CPU se asigna a los procesos en el orden en que la solicitan.
Por lo tanto, esperamos observar que, con un solo núcleo, un proceso no pueda correr hasta que no terminaron los anteriores a el.

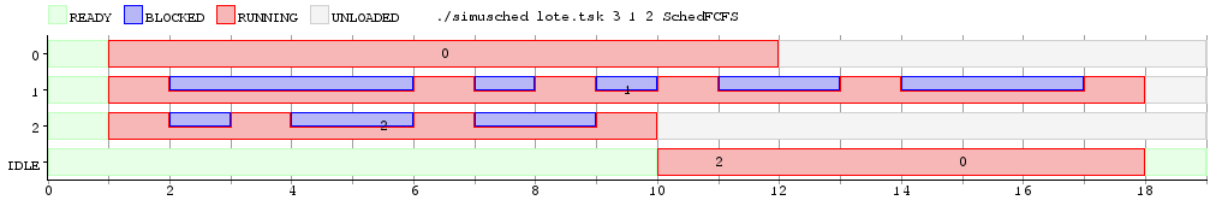
Con dos nucleos, correrán dos procesos simltáneamente y el último empezará cuando alguno de los otros dos finalicen, y por último, si el procesador tiene 3 núcleos, los 3 correrán al mismo tiempo.



Simulacro FCFS con un núcleo



Simulacro FCFS con dos núcleos



Simulacro FCFS con tres núcleos

Los experimentos corroboraron nuestra hipótesis, pues el comportamiento reflejado es exactamente el descripto previamente.

Además, notemos que en los casos en los que utilizamos la tarea TaskConsola, se observan claramente las llamadas bloqueantes de manera random, dado que el tiempo tanto de la ejecución como el de las llamadas bloqueantes varía. En el caso en que el procesador tiene tres núcleos, se observa que los núcleos 2 y 0 ejecutan Idle ya que están desocupadas, esperando la solicitud del próximo proceso.

2.3. Ejercicio 3: Implementando Round-Robin

La idea del scheduler *Round-Robin* es darle un quantum a cada proceso, iterando los mismos para que todos ejecuten una vez antes de volver a comenzar la iteración.

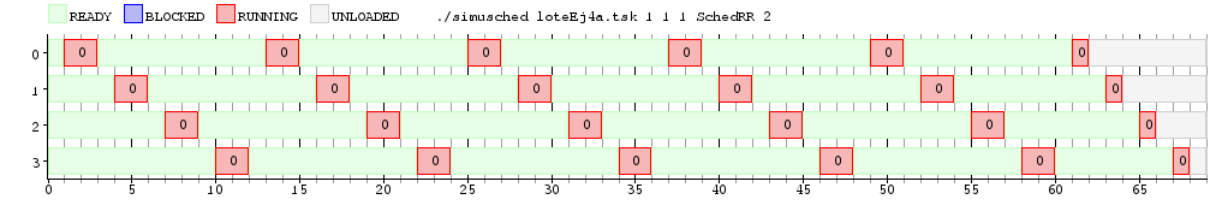
Con esta idea desarrollamos nuestro Scheduler Round-Robin. Utilizamos como estructura una Cola (q), para las llamadas a los procesos, un vector (*quantum*) de tamaño cantidad de cores del procesador que asignará el quantum del iésimo núcleo, y otro vector (*contador*) que va a llevar cuenta del tiempo corrido por el proceso en el iésimo núcleo hasta llegar al quantum del mismo.

Para el correcto funcionamiento en la función *tick* se ven reflejados los casos en el cual el proceso debe dejar de correr ya sea porque terminó su tiempo o el quantum del procesador en el que corría. En éste último caso, la posición correspondiente al núcleo en *contador* volverá a cero y el proceso se encolará para terminar con su tiempo.
Una vez realizada dicha acción, debe dar lugar a la siguiente en la cola. En caso de no haber una, se ejecutara la tarea IDLE hasta el llamado de una nueva tarea.

2.4. Ejercicio 4: Experimentando Round-Robin

En este ejercicio nos proponemos experimentar con el scheduler del punto anterior para verificar que el comportamiento es el esperado. Haremos esto de manera incremental, es decir, empezaremos probando las cosas mas basicas e iremos subiendo la complejidad.

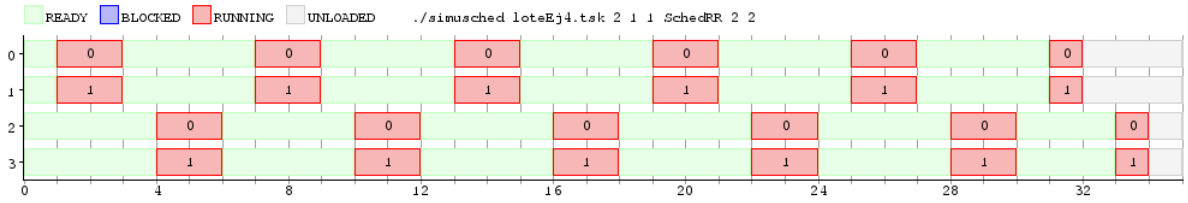
Para empezar, probaremos RR con un solo nucleo. Notese que si le asignamos una sola tarea, su comportamiento no diferiria de algún otro scheduler, por lo que empezamos probando con cuatro tareas simultaneas. Estas tareas solo usan al cpu, por lo tanto no se bloquean. Esperamos verificar que el scheduler RR le asigna el tiempo del quantum a cada tarea antes de comenzar de nuevo a iterar la lista de tareas pendientes.



Simulacro RR con un núcleo

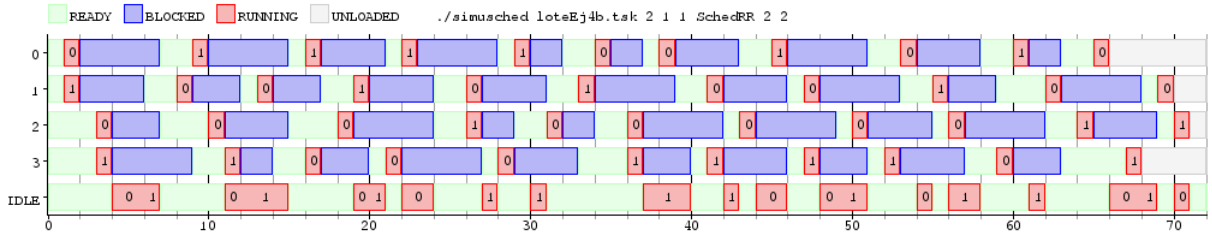
Efectivamente, cuando recibe k tareas simultaneas, el scheduler le asigna tiempo de ejecución a las tareas de modo que todas ejecuten antes de regresar a ejecutar la primera.

A continuación usaremos el mismo lote de tareas, pero agregaremos otro procesador. Asi veremos si el scheduler maneja correctamente los procesadores para asegurarse que todas las tareas ejecuten un tiempo *quantum* antes de volver a empezar.



Simulacro RR con dos núcleo

El comportamiento es identico al anterior, solo que agregando otro nucleo, es decir, podemos deducir las mismas concluciones. Para probar de modo mas realista este scheduler, vamos a simular un lote de cuatro tareas que realicen llamadas bloqueantes, con dos procesadores para su ejecución. Lo que queremos mostrar, es que el scheduler trabajara de manera optima en este caso.



Simulacro RR con dos núcleo

Podemos ver que el scheduler trabaja de la manera esperada. Es decir, si hay un tarea disponible para ejecutar y procesador que no esta ejecutando nada, ese procesador carga la tarea y la ejecuta, o bien hasta que se le acabe el quantum o hasta que se bloquee. Si todas las tareas estan bloqueadas o ejecutando cuando el procesador termina la ejecucion de una tarea porque se bloqueo, este se poner a ejecutar la tarea IDLE hasta que una tarea se desbloque. Y por lo tanto, si todas las tareas estan bloqueadas, ambos procesadores ejecutan la tarea IDLE hasta que alguna tarea se desbloquee.

2.5. Ejercicio 5: *Scheduling algorithms for multiprogramming in a hard-real-time environment*

Este ejercicio está dividido en dos incisos: en el primero, contestaremos una serie de preguntas formuladas por la cátedra, basando nuestras respuestas en el artículo *Scheduling algorithms for multiprogramming in a hard-real-time environment*; en el segundo, explicaremos el diseño e implementación de los algoritmos de scheduling de prioridades fijas y dinámicas presentados en dicho artículo.

Inciso 1

En este inciso, debemos contestar tres preguntas teóricas acerca de los algoritmos de scheduling propuestos en el artículo previamente mencionado. Antes de contestar las preguntas, comenzaremos con una breve descripción de las condiciones de entorno sobre las cuales los algoritmos fueron ideados.

Se cuenta con un sistema, con un conjunto de tareas destinadas a resolver, cada una, una determinada funcionalidad vital para el correcto funcionamiento de dicho sistema. Cada una de estas tareas estará asociada a un evento externo, que solicitará su ejecución. Es importante destacar que las tareas no pueden ser ejecutadas antes de que dicho evento las solicite. Además, se sabe que cada tarea tiene, por un lado, una *deadline* (esto es, una cantidad de tiempo máximo en el cual su ejecución debe terminar), que se mantendrá constante durante toda la ejecución del sistema. Por otro lado, se sabe que el evento que solicitará la ejecución de cada tarea, solicitará periódicamente (esto es, el intervalo de tiempo entre dos solicitudes será siempre el mismo y no dependerá de la terminación de otras tareas solicitadas) la ejecución de dicha tarea y, adicionalmente, se sabe que el tiempo de ejecución de cada tarea (entendiendo tiempo de ejecución como la cantidad de clocks que le llevaría a una tarea comenzar y terminar su ejecución si el procesador sólo tuviera que ejecutar a esa tarea) será constante. Una condición extra establece la existencia de tareas no periódicas: estas tareas desplazaran del procesador a las periódicas y, a diferencia de las periódicas, no tendrán una deadline

estricta para terminar. Una vez establecidas las condiciones del sistema, estamos listos para contestar las preguntas:

a) ¿Qué problema están intentando resolver los autores?

Dado un sistema que se ajuste a las condiciones de entorno previamente explicadas, los autores quieren hallar una forma heuristica de organizar la ejecución de las tareas, a medida que estas son solicitadas, de manera tal de que todas terminen su ejecución antes de su respectiva *deadline*. Para esto, los algoritmos presentados estarán basados en prioridades, es decir, cómo se le asignará la prioridad a cada tarea variará según el algoritmo. Luego, los algoritmos de scheduling deberán desalojar a la tarea que esté ocupando el procesador si llega una solicitud para la ejecución de una tarea más prioritaria. Por lo tanto, la clave para este tipo de algoritmos estará en cómo se le asignarán las prioridades a las tareas.

b) ¿ Por qué introducen el algoritmo de la sección 7? ¿Qué problema buscan resolver con esto?

Los autores introducen el algoritmo de la sección 7, buscando bajar la cota superior de ln 2 sobre el tiempo de utilización del procesador establecida por el algoritmo de scheduling con prioridades fijas, sin tener que asumir ninguna hipótesis extra para los tiempos de ejecución de las tareas, ni tampoco necesitar relajar las *deadlines* de las tareas menos prioritarias según el esquema anterior. Para lograr esto, introducen el algoritmo de scheduling con prioridades asignadas dinámicamente; es decir, a lo largo de la ejecución del sistema, las prioridades de las tareas no estarán necesariamente fijas.

c) Explicar coloquialmente el significado del teorema 7.

El teorema 7 establece una condición necesaria y suficiente, sobre el algoritmo de prioridades dinámicas, para que todas las tareas terminen de ejecutarse antes de su *deadline*. Dicha condición es la siguiente:

$$C_1/T_1 + ... + C_n/T_n \leq 1$$

donde

- $n :=$ número de tareas en el sistema
- $C_i :=$ tiempo de ejecución de la tarea iésima
- $T_i :=$ tiempo entre dos solicitudes consecutivas por la tarea iésima (también llamado período)

Es importante señalar que la condición que establece este teorema nos permitirá, dado un lote de tareas diseñado para nuestros experimentos, definir si el algoritmo de scheduling con propiedades dinámicas logrará que todas las tareas terminen de ejecutar antes de sus respectivas *deadlines*, a lo largo de toda la simulación.

Inciso 2

En este inciso explicaremos brevemente en qué consiste cada algoritmo de scheduling, y luego el diseño y la implementación de cada uno.

Como su nombre indica, el algoritmo de scheduling con prioridades fijas le asignará a las tareas una prioridad que se mantendrá fija a lo largo de toda la ejecución del sistema. Concretamente, una tarea será más prioritaria que otra cuando su período, lease el tiempo constante transcurrido entre dos solicitudes por dicha tarea, sea el menor de los dos. O equivalente, que su *request rate*, definido como el inverso multiplicativo del período, sea mayor.

Para el diseño de este algoritmo de scheduling, optamos por utilizar una cola de prioridad, que contendra duplas de la forma $< periodo(pid), pid >$, donde el más prioritario será el que tenga menor período. Este diseño nos permitirá obtener de manera sencilla cuál es la próxima tarea a ser ejecutada, aprovechando las funcionalidades ya implementadas en la clase *priority queue* para encolar elementos y obtener el más prioritario. De esta manera, obtener en cada tick de reloj cuál es la tarea a ejecutarse se resumirá a verificar, en primer lugar, si la cola de prioridad está vacía. En caso de que esté vacía, se ejecutará la tarea IDLE; en caso contrario, se obtendrá el *pid* de la próxima tarea a ser ejecutada mediante la función *top()*, devolviendo el segundo componente de la dupla devuelta por dicha función.

A diferencia del algoritmo anterior, el algoritmo de scheduling con prioridades dinámicas, le asignará las prioridades a cada tarea en cada *tick* de reloj. Esto se hara de la siguiente manera: a cada momento de la ejecución del sistema, la tarea más prioritaria será la que tenga su *deadline* más próxima; coloquialmente, esto quiere decir que lo más urgente será lo más prioritario. Para el diseño de este scheduler, como debíamos actualizar las prioridades en todos los *ticks* de reloj, optamos por utilizar arreglos en vez de una cola de prioridad. Esto es porque, de utilizar una cola de prioridad, sería más complicado iterar los procesos contenidos en la tabla para actualizar sus prioridades. Por lo tanto, contaremos con los siguientes arreglos:

- *int deadline[totaltasks]* indicara, para la tarea iésima, cuánto tiempo le queda antes de su *deadline* en $deadline[i]$. Para las tareas no periódicas, adoptamos la convención de almacenar un -1 en esa posición del arreglo.
- *bool ready[total tasks]* indicará, para la tarea iésima, si está lista para correr o no.

Adicionalmente, implementaos la funcion *int tareasready()*, que devolverá la cantidad de tareas en estado *ready*. Para obtener, en cada *tick* de reloj, la próxima tarea a ejecutarse, implementamos una función de acuerdo a la siguiente lógica: Si está corriendo una tarea no periódica, seguir ejecutando esa. En caso contrario, verificar, en primer lugar, si hay alguna tarea no periódica en estado *ready*. En caso de haberla, pasar a ejecutar esa; en caso contrario, buscar la tarea periódica en estado *ready* cuya *deadline* esté más próxima y devolver esa. Vale la pena aclarar que además, en cada *tick* de reloj, se decrementará el valor contenido en el arreglo *deadline* para cada tarea periódica que esté lista para ejecutarse. En este punto, hacemos la aclaración de que hemos dejado fuera de la descripción algunos de los casos borde para los cuales no haya tareas listas para ejecutarse, en que deba devolverse el *pid* de la tarea Idle, ya que no suma a la comprensión del caso en que el algoritmo deba buscar, entre las tareas existentes, la más prioritaria.

2.6. Ejercicio 6: TaskBatch

En este ejercicio implementamos la tarea TaskBatch, que recibe como parametros totalcpu y cantbloqueos. Esta tarea dura totalcpu tiempo de cpu, y realiza cantbloqueos llamadas bloqueantes en momentos psudoaleatorios. Implementamos esta tarea de dos maneras. La primera implementación itera totalcpu veces pidiendo un numero aleatorio rand. Si rand >0.5 entonces llama a una tarea bloqueante, sino, a una tarea que use el cpu. Si la cantidad de iteraciones se acaba sin haber realizado todas las llamadas bloqueantes, entonces se realizaban las restantes seguidas al final. Sino, cuando se realizaba la ultima llamada bloqueante, se utilizaba el tiempo restante del cpu todo junto. Lo que observamos en esta implementacion fue que las llamadas bloqueantes se hacian para tiempos grandes siempre al principio del intervalo de uso del cpu, y para tiempos de medianos a cortos, siempre al final.

Por eso realizamos otra implementacion. En esta se seleccionan previamente en que momentos se van a realizar las llamadas bloqueantes, marcando en un arreglo de totalcpu posiciones los momentos en los que se va a llamar la tarea bloqueante. Se decide los momentos de bloqueo eligiendo un numero aleatorio entre 0 y totalcpu, asegurandonos de que no halla repetido. Finalmente iteramos este arreglo y llamamos a UsoIO si debemos hacer un bloqueo y a Uso CPU sino.

2.7. Ejercicio 7

En el siguiente ejercicio, se nos pide escoger distintas métricas para poder analizar el rendimiento de el Shcheduler Round Robin para tareas de tipo TaskBatch.
A continuación, daremos algunos detalles de las seleccionadas.

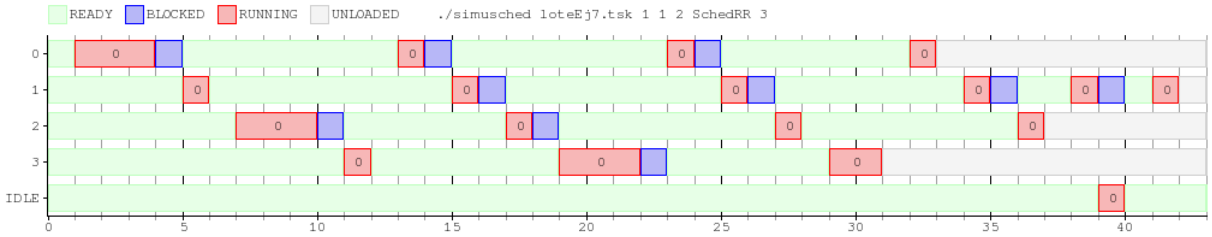
Fariness
Medimos que cada proceso reciba una dosis "justa"de CPU. Es decir, todos los procesos deben correr la misma cantidad de tiempo, en el caso de Round Robin, podemos decir el quantum que se le asigna a los procesos sea el mismo en todos los casos.

Tiempo de Respuesta
En este caso, seria el tiempo que tarda una tarea en empezar a ejecutarse. Cuanto tiempo permanece en estado ready hasta la primera ejecucion. En el caso de Round Robin, esto depende de cuantas tareas hay esperando antes de la misma, ya que en este Scheduler se utiliza una cola como estructura, y ademas de cuantos nucleos tiene el procesador.

Throughput
Son la cantidad de procesos que terminan por unidad de tiempo.
Esto dependera de la cantidad de bloqueos que tenga cada proceso y como se organiza la CPU en cuanto a quantum y cantidad de nucleos.

Turnaround
Es el tiempo total que le toma a un proceso su ejecucion completa, contando bloqueos y cantidad de corridas en quantums de algun nucleo del CPU.

Para empezar, tomaremos una CPU con un solo nucleo y con un quantum de 3 clocks.

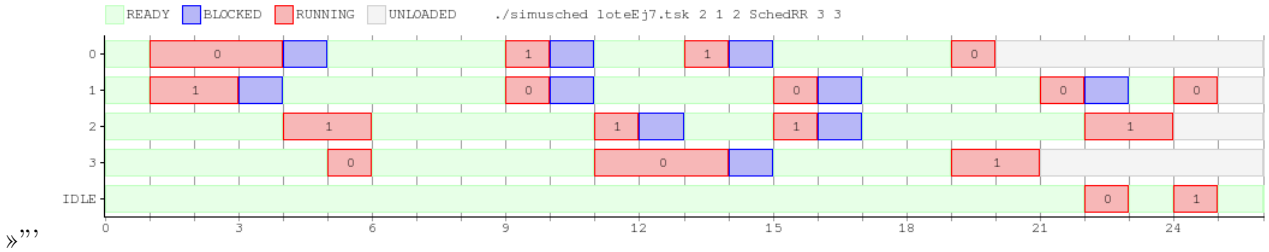


Simulacro RR con un núcleo de quantum 3

«"'<HEAD ======= Como podemos observar y ya evaluamos en ejercicios anteriores el Scheduler Round Robin funciona como lo esperado. Ahora,nos detallaremos en su rendimiento segun las metricas detalladas anteriormente.

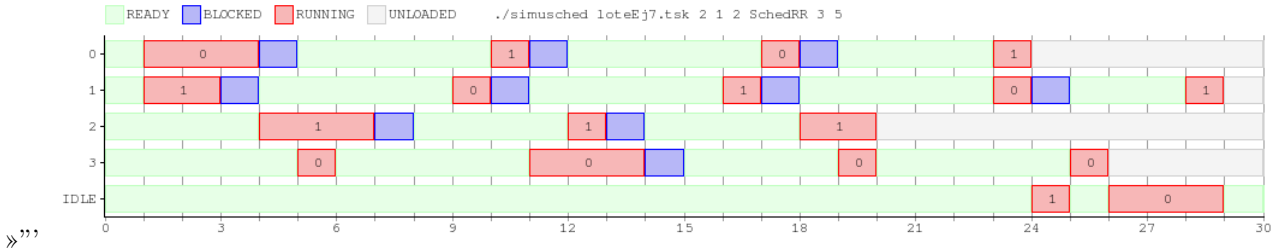
»"'Si utilizamos la metrica *Fairness*, este tipo de Scheduler con un solo nucleo es apropiado ya que todos los procesos recibiran la misma cantidad de tiempo de ejecucion. Sin embargo, si nos basamos en *Tiempo de respuesta* o *Turnaround*, no es optimo ya que, al tener un solo nucleo, van a demorar mas tiempo en comenzar a ejecutarse y en finalizar por completo.

»"'Ahora nos preguntamos que pasaria si aumentamos lacantidad de nucleos del procesador. De esta manera pueden suceder dos cosas,la primera que tengan la misma cantidad de quantum ambos nucleos, y la segunda que sean distintos. A continuacion, estudiaremos ambos casos.



»"'

»"'*Simulacro RR con dos núcleos de quantum iguales a 3*



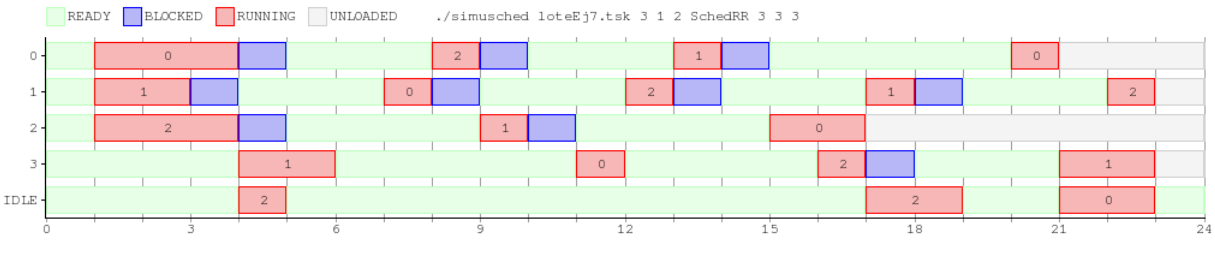
»"'

»"'*Simulacro RR con dos núcleos de quantum distintos*

»"'Como podemos observar los graficos son similares, esto radica en que si bien cuando aumentamos el quantum de uno de los nucleos tenemos la posibilidad de que un procesos se ejecute mas rapido, el mismo es afectado por los bloqueos realizados en la tarea corriendo.
Si nos basamos en la teoria, podemos decir que cuando el Scheduler corre con dos nucleos con la misma cantidad de quantum *Fairness* es optimo ya que todos tendran las mismas posibilidades en tiempo de ejececucion, sin embargo, el mismo no es apropiado en el segundo caso, porque los procesos recibieran distintos quantums lo cual aletera esta metrica.
En el caso de *Tiempo de Respuesta* va a mejorar en comparacion a el procesador con un nucleo ya que hay uno mas que puede ser utlizado, esto sucede en ambos casos.
Con respecto a *Throughput* teniendo dos nucleos al menos 2 procesos pueden terminar por unidad de tiempo, lo cual mejora el primer experimento que solo perimitia uno.
En *Turnaround* podemos decir que la mejora con respecto al primer experimento es que ademas de que algun proceso

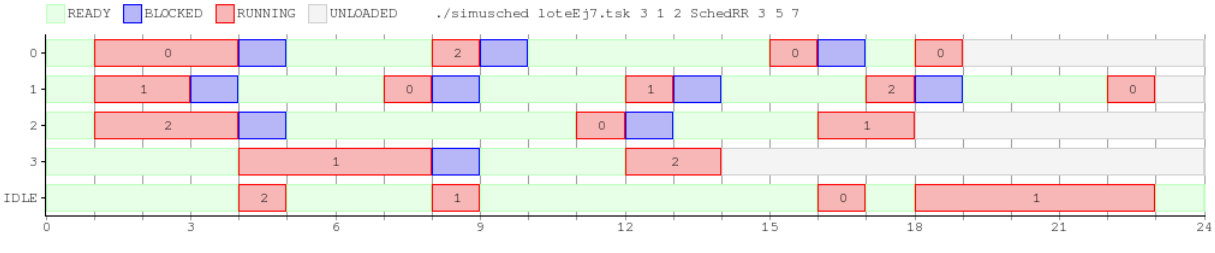
comenzara a ejecutarse antes ya que ahora hay un nucleo mas, en el caso de quantums distintos, si el proceso arbitrariamente es beneficiado obteniendo el nucleo con mayor quantum terminara antes, mientras que los bloqueos no alteren su ejecucion.

»"'Por ultimo, observamos que pasaria si nuestro procesador tiene tres nucleos, al igual que el ultimo experimento realizado tenemos dos posibilidades, aqui las mismas:



»"'

»"'*Simulacro RR con tres núcleos de quantum iguales a 3*



»"'

»"'*Simulacro RR con tres núcleos de quantum distintos*

»"'Para comenzar, las conclusiones que obtenemos, al igual que anteriormente son afectadas por los bloqueos que tiene los procesos.
Analicemos teoricamente nuestos experimentos.
En el caso de utlizar como metrica *Fairness*, los resultados que obtenemos son iguales a los obtenidos en el segundo experimento. Es decir, en caso de poseer quantums iguales esta metrica es satifactoria, sin embargo, en el segundo caso no lo sera, ya que la CPU no esta brindandole una dosis justa a cada proceso.
Para *Tiempo de Respuesta* obtenemos una mejora para este experimento con respecto al anterior, un proceso tiene posibilidades de comenzar a ejecutar mas tempranamente debido al aumento en la cantidad de nucleos. Aun siendo visible esta mejora claramente en quantums iguales, en el caso contrario tambien podemos notarlo, ya que al tener distintos quantums el procesador puede estar libre en distintos momentos.
Otra mejora que notamos es en el caso de utilizar como metrica *Troughput* ya que con respecto a los experimentos anteriores, ahora podran finalizar al menos tres procesos en ambos casos.
Por Ãºltimo, si utilizamos *Turnaround* podemos obetener una mejor medicion respecto a los ultimos dos experimentos. Esto es debido a que el procesaodr tiene tres nucleos, esto es mas notorio si ademas los quantums son distintos, ya que un proceso puede ser beneficiado obteniendo el nucleo con mayor tiempo y asi finalizar antes.

»"'Podemos concluir con estos experimentos que deducir que si un Scheduler es optimo varia segun algunos factores y que metrica estemos utilizando.
Pudimos ver que *Fairness* depende de los quantums que se le asigna a cada nucleo del procesador, y segun eso puede resultar satifactorio o no.
En *Tiempo de Respuesta* notamos que varia segun la cantidad de nucleos y, de tener distintos quantums, esto puede alterar los resultados aun mejor.
Para *Troughput* depende de la cantidad de nucleos que tenga el procesador, ya que para que termine mas de una tarea, deben trabajar nucleos paralelamente.
Por ultimo, *Turnaround* depende de varias cosas, en caso de que los quantums sean iguales no podemos concluir en

demasiado, pero si tenemos disintos podemos obtener resultados satifactorios segun de cada proceso.
Si bien los resultados son teoricos, pudimos notar que nuestras medidas resultan alteradas por los bloqueos realzados por las tareas, el costo de la migracion de un nucleo a otro y el cambio de contexto.

»”””»>63ea077d0c52c262c2910339492dcc2387679a4d

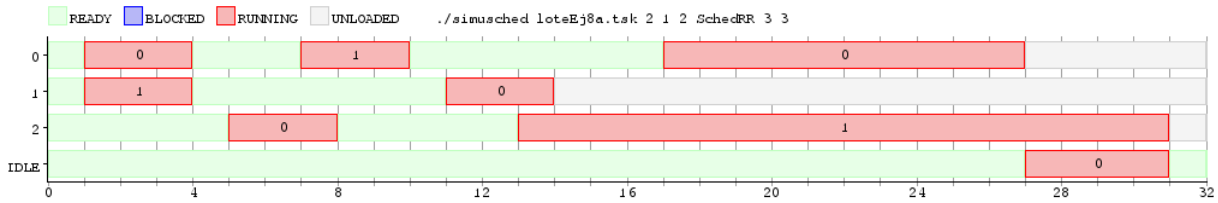
2.8. Ejercicio 8: Round Robin sin migracion

En este ejercicio implementamos un scheduler Round Robin que no permite migraciones entre procesos, llamado RoundRobin2. Para lograr esto, mantenemos una cola por procesador y un contador que controla cuantas tareas hay activas por cpu. Para asignar una tarea a un cpu, nos basta con recorrer los contadores y quedarnos con alguno de los de valor minimo. Ademas, cada vez que una tarea se bloquea, almacenamos en una variable cual es el procesador al que estaba asignada. De este modo, para todos los procesadores, podemos ejecutar como si fuera Round Robin comun. La diferencia radica en que cuando una tarea se bloquea, se almacena el valor de cpu de la misma. Cuando la tarea se desbloquea, basta con buscar su cpu y pushearla en la cola del mismo.

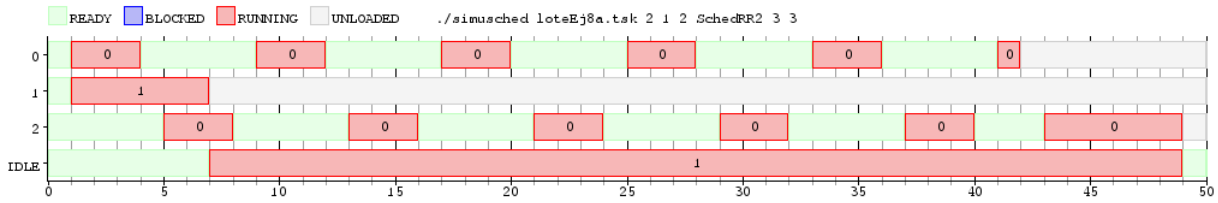
Vamos a comparar esta version del scheduler Round Robin con la presentada en el ejercicio 3. Los schedulers tiene diferente comportamiento solo en determinados casos, que son lo que vamos a analizar. Dejaremos los puntos en comun de lado, como por ejemplo, la rotacion total de las tareas. Por lo tanto, compararemos la optimalidad en el uso de los procesadores. Es decir, compararemos la cantidad de tiempo de cpu despercidiado. Para esto correremos con dos procesadores el siguiente lote de tareas:

TaskCPU 10
TaskCPU 5
TaskCPU 20

La idea de es comparar cuanto tarda cada scheduler en completar el procesamiento total. Notese que pusimos la tarea de mejor ejucucion en el medio aproposio. Asi forzamos al RR2 a usar un procesador unicamente para una tarea corta, mientras que debe usar el otro para dos tareas grandes. Es decir, estamos forzando un caso concreto para representar casos mas generales. Ademas, se debe tener el cuenta que el costo de migracion es de 2 tick, lo cual es bajo teniendo en cuenta todos los datos que se podrian tener que duplicar.

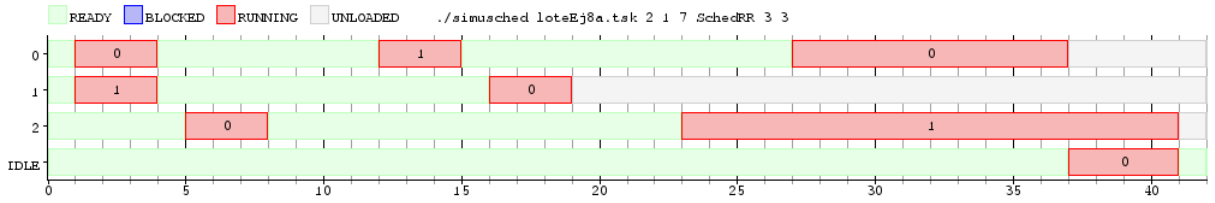


Simulacro RR con 2 núcleos de quantum 3.
Con 2 tick de costo de migracion.

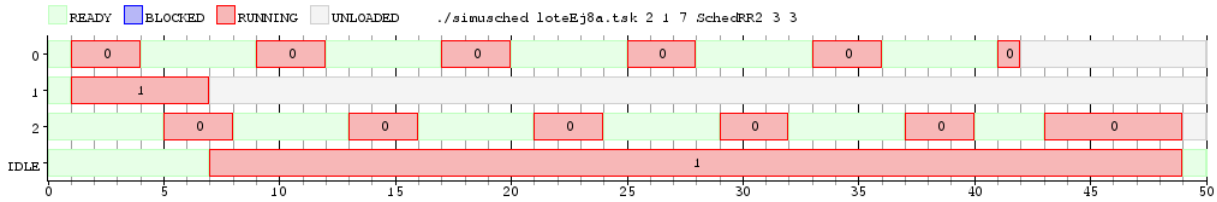


Simulacro RR2 con 2 núcleos de quantum 3
Con 2 tick de costo de migracion.

Como podemos observar, el tiempo para obtener la ejecucion total es mucho mayor en RR2, ya que no puede usar uno de los procesadores. ¿Pero que pasaria si el costo de migrar entre proesadores fuera mayor? Digamos, mas del doble del quantum...



Simulacro RR con 2 núcleos de quantum 3.
Con 7 tick de costo de migracion.



Simulacro RR2 con 2 núcleos de quantum 3.
Con 7 tick de costo de migracion.

Podemos notar que aunque la diferencia se redujo, el tiempo total de RR2 sigue siendo mayor que el de RR. Por lo tanto, solo nos resta concluir que no permitir migracion entre procesadores es un error. Esto sujeto a condiciones normales. Si tuviesemos un costo de migracion ridiculamente alto, si trabajasemos fuera de la cache para la migracion, entonces deberia considerarse realizar una nueva experimentacion comparando los nuevos porcentajes.

2.9. Ejercicio 9

En este ejercicio, debimos idear un lote de tareas que cumpliera en simultáneo, las siguientes condiciones:

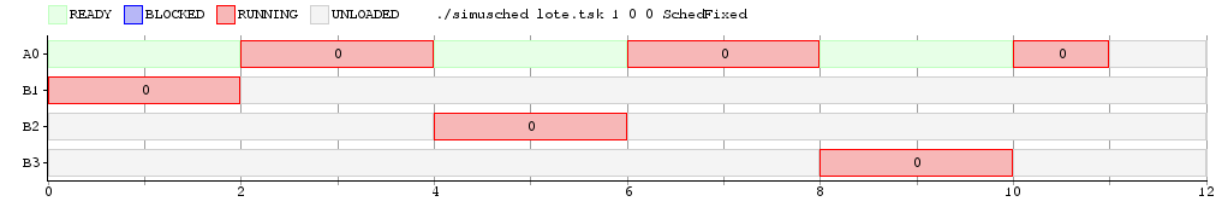
- Tener un scheduling no factible para el algoritmo de prioridades fijas
- Tener un scheduling factible para el algoritmo de prioridades dinámicas

El lote que propusimos para este experimento es el siguiente:

```
lote.tsk:
  &A1,10,4
  &B3,4,1
```

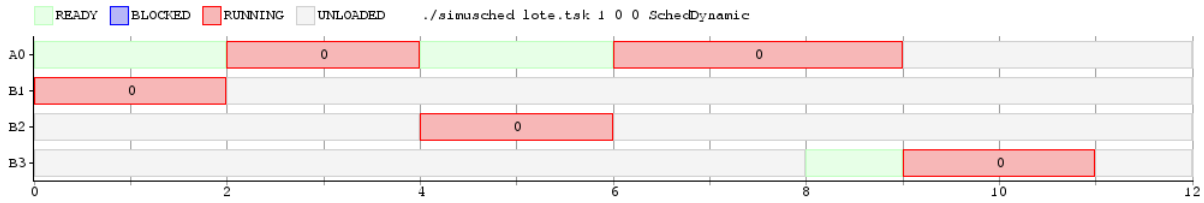
Es decir, una repetición de tarea de tipo A, con 4 ciclos de clock de tiempo de ejecución y 10 ciclos de clock como período, y 3 repeticiones de tareas de tipo B, con 1 ciclo de clock de tiempo de ejecución y período igual a 4. A continuación, mostraremos que con esta combinación de períodos y tiempos de ejecución, la tarea A no terminará de ejecutarse antes de su *deadline* (ciclo de clock número 10) para el scheduler de prioridades fijas. Es importante notar que a los tiempos de ejecución de las dos familias de tareas hay que sumarle el ciclo de clock extra correspondiente a la llamada a *exit()*, con lo cual, la tarea A requerirá de 5 ciclos para completar su ejecución, y las tareas B requerirán de 2 ciclos cada una.

Veamos el diagrama de Gantt para este lote, con scheduling con prioridades fijas:



Simulación SchedFixed

En los instantes múltiplos de 4 (0, 4 y 8) llega al sistema una *request* por una tarea de familia B, mientras que en el instante 0 llega la única *request* por una tarea de tipo A. Como las tareas de tipo B, por tener menor período, son más prioritarias que la de tipo A, en los instantes 0, 4 y 8, el scheduler decide poner a correr las tareas de tipo B, durante los dos ciclos que necesitan para terminar. Esto le deja a la tarea A 4 ciclos de clock disponibles en los primeros 10 ciclos de clock, con lo cual no puede terminar la ejecución antes de que llegue su deadline, haciendo inviable el uso del scheduler de prioridades fijas para este lote de tareas.
Ejecutando el mismo lote de tareas, pero con scheduling de prioridades dinámicas, obtenemos el siguiente diagrama de Gantt:



Simulación SchedDynamic

En esta simulación, el comportamiento del scheduler es idéntico al de prioridades fijas hasta el instante 8, correspondiente a la tercer *request* por una tarea de tipo B. En este instante, la tarea de tipo A tiene su deadline dentro de 2 ciclos de clock, mientras que la tarea de tipo B tiene su deadline a 4 ciclos de clock de distancia, por lo cual el scheduler decidirá poner a correr a la tarea de tipo A en vez de la tarea de tipo B. Esto le permitirá a la tarea de tipo A consumir el último ciclo de CPU que necesitaba, y luego el scheduler pondrá a correr a la última tarea de tipo B, que terminará sin problemas su ejecución. Así, todas las tareas del lote terminaron su ejecución antes de su *deadline*.