DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Scheduling

Sistemas Operativos
Primer Cuatrimestre 2015

Integrante	LU	Correo electrónico
Aldasoro Agustina	86/13	agusaldasoro@gmail.com
More Ángel	931/12	angel_21_fer@hotmail.com
Zimenspitz Ezequiel	155/13	ezeqzim@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

Resumen

El presente trabajo aborda el desarrollo algorítmico de varios Scheduler, como así tambien de tipos de tareas que deberan ejecutar. Se evaluara el comportamiento de cada algoritmo para una serie de tareas dada y como afecta dicho desarrollo si se aplican otros factores como modficaciones en el quantum o en la cantidad de core. Además, compararemos el rendimiento de distintos Scheduler bajo diversas métricas.

Índice

1. Parte I

1.1. Ejercicio 1: TaskConsola

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 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.

Al momento de ejecutar la tarea *TaskConsola*, lo que realiza nuestro algoritmo es *uso_IO* n veces (se bloquea n veces), eligiendo cada vez un número al azar entre *bmin* y *bmax*. Esta elección se realiza mediante la función *rand()*, fijando la semilla en srand(time(NULL)) de modo que varía entre ejecuciones seguidas.

1.2. Ejercicio 2: Ejecución de tres tareas para TaskConsola

Escribir un lote de 3 tareas distintas: una intensiva en CPU y las otras dos de tipo interactivo (TaskConsola). Ejecutar y graficar la simulación usando el algoritmo FCFS para 1, 2 y 3 núcleos.

El comportamiento de un Scheduler basado en la técnica FCFS (First Came, First Served) consiste en ejecutar cada tarea entrante, acorde a su orden de aparición en *ready* sin realizar ninguna interrupción ni cambio de tarea. Cada tarea ejecuta desde su comienzo hasta que finaliza. El lote de tareas que observamos consistió de tres tareas: dos se corresponden el tipo de tarea *TaskConsola* y la restante *TaskCPU* (sólo realiza un uso intensivo del CPU -sin bloqueo-).

Las tareas utilizadas las siguientes:

```
TaskConsola 4 2 7
TaskCPU 3
@3:
TaskConsola 5 1 10
```

Además, para observar cómo varía el tiempo total (si es que lo hace), se simuló la ejecución para 1, 2 y 3 cores. Se le otorgó un costo de cambio de contexto de un clock a todos los casos. Dado que en este tipo de Scheduler no hay migración de procesos entre los distintos núcleos, los demás parámetros no son de importancia.

Los diagramas de Gantt obtenidos fueron los siguientes:

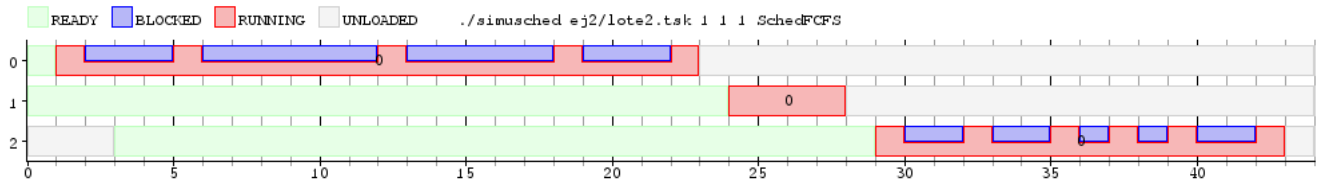


Figura 1: Diagrama de Gantt para la ejecución del lote de tareas bajo 1 núcleo.

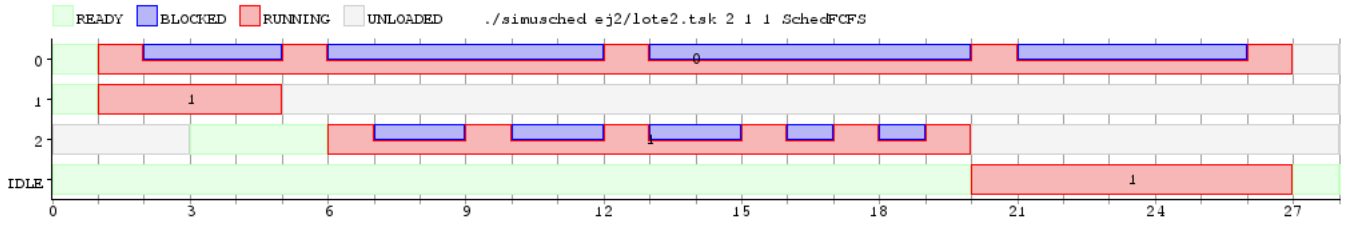


Figura 2: Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos.

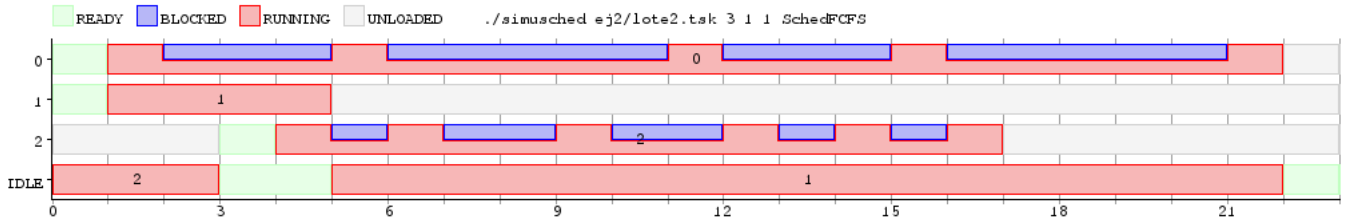


Figura 3: Diagrama de Gantt para la ejecución del lote de tareas bajo 3 núcleos.

Dado a que el algoritmo de Scheduler utilizado es First Come First Served (FCFS), las tareas entrantes se ejecutan secuencialmente por orden de llegada. La ejecución de ninguna tarea es interrumpida, ya que la tarea ejecuta hasta terminar con su ejecución, sin tener en cuenta si está bloqueada o no.

Conociendo el comportamiento explicado, es coherente que el primer caso (donde se cuenta con un sólo núcleo) es donde ejecutar el lote de tres tareas conlleva mayor tiempo. Esto se debe a que el tiempo que demora en ejecutar las tres tareas es la suma de los tiempos de cambio de contexto, el tiempo de ejecución para cada una y el tiempo que permanece bloqueado.

En cambio, al ejecutarlo con dos o tres núcleos se puede apreciar en los diagramas cómo se ejecutan tareas en simultáneo. Este comportamiento reduce notablemente el tiempo de ejecución total, ya que en este caso no sería la suma del tiempo parcial para cada tarea.

A partir de los resultados obtenidos en los gráficos anteriores. Podemos observar que el tiempo total varía de acuerdo al número de cores con los que se este simulando. A medida que se incrementa, el número del tiempo total disminuye. No sólo influyó este factor; sino que también, el hecho de que estamos trabajando con un modelo sin desalojo. Al trabajar con un solo core debemos esperar que una tarea finalice para que la próxima pueda ser ejecutada. Es decir, se van a ir ejecutando secuencialmente según fueron estando $ready$.

Al aumentar en una unidad el número de cores: la primer tarea disponible comenzó a ejecutarse. Dado que aun disponemos de otro core; cuando una segunda tarea pasó a estar disponible, pudo ser ejecutada al instante por el núcleo adicional. Y cuando uno de los núcleos se desocupó, se ejecutó la última tarea. Gracias a que dos tareas pudieron ejecutarse simultáneamente el tiempo total disminuyó.

Aún más cuando se trabajó con 3 cores, cada tarea pudo ser ejecutada en un núcleo distinto y como pudieron ejecutarse todas a la vez, el tiempo total fue igual al máximo entre $release\ time_j + c_j$, donde j indica alguna de las tres tareas utilizadas y c_j su tiempo de ejecución.

2. Parte II

2.1. Ejercicio 3: Scheduler Round-Robin

Completar la implementación del scheduler Round-Robin implementando los métodos de la clase SchedRR en los archivos sched_rr.cpp y sched_rr.h. La implementación recibe, como primer parámetro, la cantidad de núcleos y a continuación los valores de sus respectivos quantums. Debe utilizar una única cola global, permitiendo así la migración de procesos entre núcleos.

Las estructuras de datos con las que vamos a trabajar, en la clase *SchedRR*, son las siguientes:

```
int cores;
vector<int> quantums;
vector<int> quantums_timer;
vector<int> actuales;
int siguiente;
queue< int, deque<int> > cola;
```

- **cores** es la cantidad de núcleos.
- **quantums** es un vector de *cores* posiciones, que guarda en quantums[i] el valor del quantum asignado al núcleo i.
- **quantus_timer** es un vector, tal que quantus_timer[i] representa el valor del quantum restante para la tarea corriendo en el núcleo i.
- **actuales** vector que indica el PID de la tarea ejecutandose en el núcleo i (actuales[i]). Para los núcleos sin tarea devuelve la constante $IDLE\_TASK$.
- **siguiente** indica el core al que se le debe asignar la siguiente tarea.
- **cola** es la cola de tareas que restan ser ejecutadas.

Modificamos a la funcion *next* para que reciba un parámetro más (enum Motivo { TICK, BLOCK, EXIT }).

```
    int next(int cpu, const enum Motivo m);
```

Constructor Scheduler Round-Robin

Al construir un Scheduler Round-Robin, se instancian las estructuras de datos de modo que: se le asigna la cantidad de cores correspondientes (con sus respectivos quantums), todas las tareas actuales se definen como Iddle, la cola está vacía y el siguiente núcleo que le corresponde ejecutar es el primero ingresado como parámetro.

Función Load

Recibe el *pid* de una nueva tarea como parámetro. Luego, las primeras n tareas (siendo n el número de cores) se distribuyen entre cada core (dado que al inicio tengo n cores disponibles). Una vez alcanda las n distribuciones se encolan los nuevos *pid* en *cola*. Estos seran asignados a los distintos cores con la función $next$ explicada luego.

Función Tick

Recibe *cpu* como parámetro y el *Motivo*. Dado que se produjo un tick voy a actualizar el quantum (restarle 1) de la tarea en *cpu*, si no es la Iddle (porque esta corre indefinidamente hasta que la desplace otra tarea). Si se acabó el quantum de la tarea actual (ya sea porque terminó o no) o se bloqueó, voy a querer actualizar la tarea actual. Para esto, invocó a la función *next()*. La misma devuelve el *pid* de la próxima tarea a ejecutar en *cpu*.

Función Next

La función Next también va a ser invocada para un sólo núcleo *cpu* pasado por parámetro.
De este modo, si la tarea que se estaba ejecutando previo a la invocación de *next()* terminó de ejecutarse y la cola se encuentra vacía, la tarea que pondrá a ejecutar va a ser la Tarea Iddle. Si la tarea que se estaba ejecutando no terminó su ejecución o se bloqueo, se deberá encolar en la cola global de pendientes. Si todavía no asigné una tarea actual para el núcleo cpu, le asigno la primera de la cola dándole todo el quantum disponible para el núcleo *cpu*. (Si la cola se encontraba vacía al llamar la función next() y la tarea ejecutándose no había concluído se la encolará para luego volver a asignársela al núcleo).

2.2. Ejercicio 4: Ejecución de lotes de tareas para Round-Robin

Diseñar uno o más lotes de tareas para ejecutar con el algoritmo del ejercicio anterior. Graficar las simulaciones y comentarlas, justificando brevemente por qué el comportamiento observado es efectivamente el esperable de un algoritmo Round-Robin.

Con el fin de observar el comportamiento del Scheduler basado en la metodología Round Robin evaluamos tres casos:

Caso 1

Ejecutamos el mismo lote de tareas utilizado anteriormente (Sección 1.2), manteniendo los mismos parámetros, para así poder evaluar y comparar un Scheduler basado en Round Robin y otro en FCFS.

```
TaskConsola 4 2 7
TaskCPU 3
@3:
TaskConsola 5 1 10
```

Los diagramas de Gantt obtenidos fueron los siguientes:

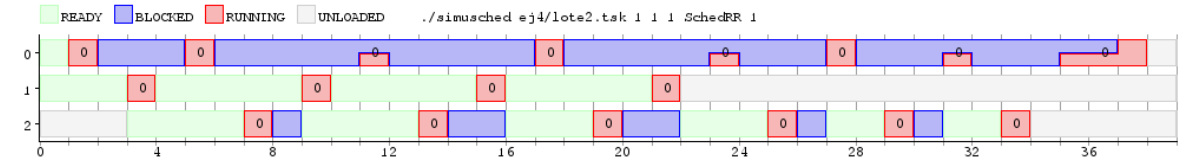


Figura 4: Diagrama de Gantt para la ejecución del lote de tareas bajo 1 núcleo con Scheduler Round Robin.

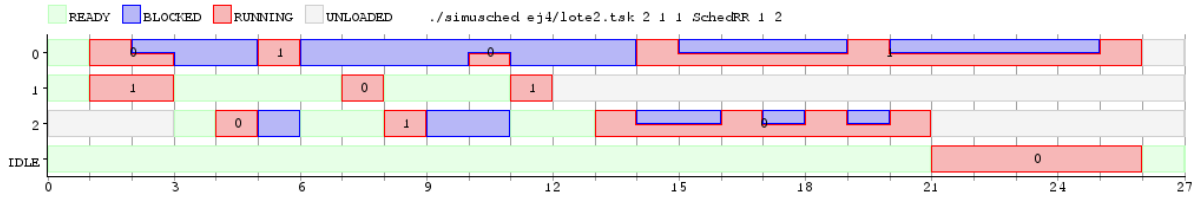


Figura 5: Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos con Scheduler Round Robin.

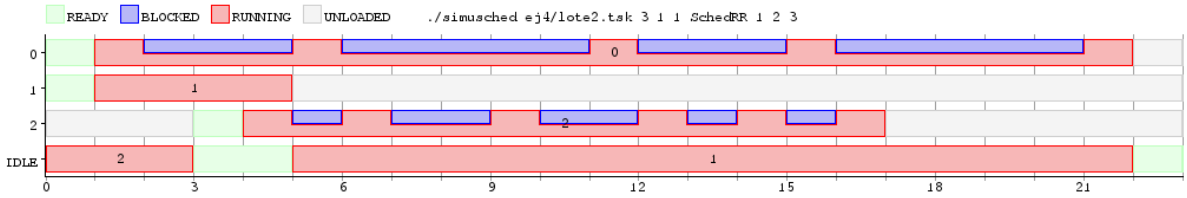


Figura 6: Diagrama de Gantt para la ejecución del lote de tareas bajo 3 núcleos con Scheduler Round Robin.

A partir de los resultados obtenidos; podemos ver que, para las **figuras 5** y **6**, el tiempo total de ejecución fue el mismo que para las figuras 2 y 3 respectivamente.

En estos casos, observamos que el tiempo dependió de la tarea *TaskConsola 4 2 7*. La cual tenía el mayor tiempo de ejecución.

Entonces, sin importar el modelo, aunque las otras dos tareas terminaron, el procesador tuvo que esperar a que finalice dicha tarea.

Un caso interesante para análizar es el de las **figuras 1** y **4**. En ambas, solo se utilizaba un núcleo pero, en el modelo Round Robin, el tiempo de ejecución fue menor. Esto se debe a que el comportamiento de dicho Scheduler permite cambios de contexto y migración de procesos.

Para el *modelo FCFS*, había que esperar que una tarea finalice para ejecutar la próxima; si la primera tenía mucho tiempo de bloqueo entonces, ese tiempo se desperciaba.

En el *modelo Round Robin* se pudieron aprovechar los tiempos que las tareas ocupaban bloqueados para ejecutar otras. Y dado que la tarea con mayor tiempo de ejecución era del tipo TaskConsola, con muchos períodos de bloqueo, el tiempo “perdido” fue suficiente para ejecutar las otras tareas en su totalidad.

Caso 2

Para este caso se creó un nuevo lote de tareas y lo que se busco fue: analizar cómo varía el tiempo total de ejecución para las mismas, a medida que se cambia el quantum para un total de dos cores.

Las tareas que se usaron son las detalladas a continuación:

```
TaskConsola 10 2 6
TaskCPU 10
@3:
TaskCPU 7
@2:
TaskConsola 7 4 10
```

Los resultados obtenidos fueron:

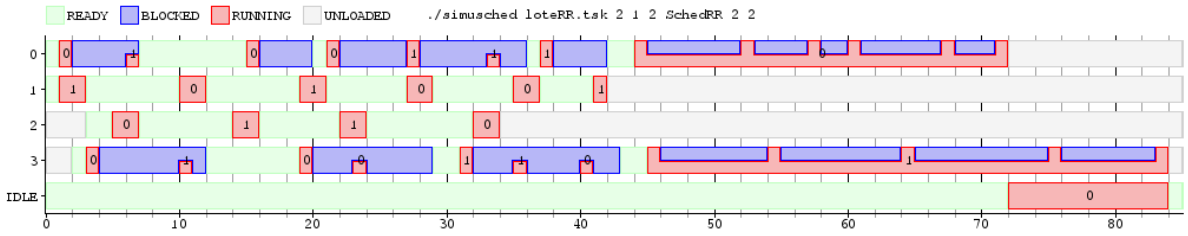


Figura 7: Diagrama para la ejecución del lote de tareas bajo 2 núcleos con quantum de 2 y 2 ciclos respectivamente.

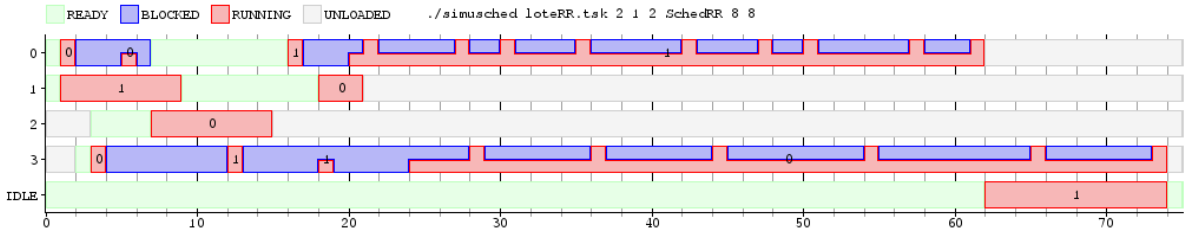


Figura 8: Diagrama para la ejecución del lote de tareas bajo 2 núcleos con quantum de 8 y 8 ciclos respectivamente.

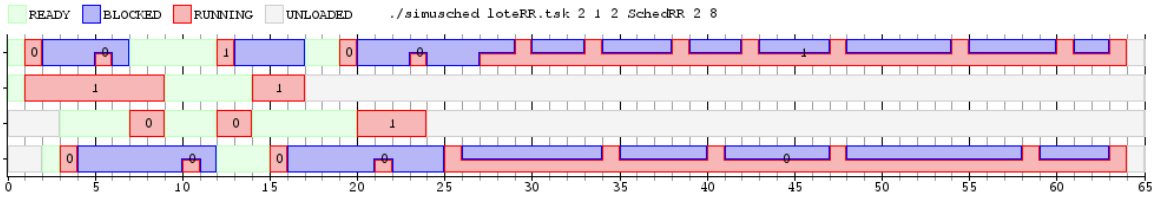


Figura 9: Diagrama para la ejecución del lote de tareas bajo 2 núcleos con quantum de 2 y 8 ciclos respectivamente.

La hipótesis que nos llevo a realizar esta experimentación fue pensar que si los quantums asignados a cada core eran muy bajos, entonces el tiempo total de ejecución para los procesos iba a ser mayor que si se utilizaran quantums con una cantidad de ciclos mayor. Los resultados obtenidos se acercan a lo esperado, para quantums muy chicos el tiempo de ejecución total aumenta frente a otros casos (se puede observar dicho comportamiento en la **figura 7**). Principalmente, cuando se utilizan quantums muy chicos ocurren muchos cambios de contextos. Si su tiempo no es depreciable, aumenta el tiempo total que el procesador necesita para ejecutar todas las tareas.

Luego, al aumentar el quantum de cada core a 8 (**figura 8**), observamos que el tiempo total fue menor que para el caso anterior ya que se produjeron menos cambios de contexto.

Sin embargo, cuando se usó un quantum bajo y otro de mayor ciclo (**figura 9**), el tiempo total bajó con respecto al anterior caso. Podemos atribuir este comportamiento al tipo de tareas con el que trabajamos.

Como se observa, algunas tareas pudieron migrar en este último caso y finalizar su ejecución más rapidamente. Intuyendo, con estos resultados, que quantums de ciclos muy bajos no son beneficiosos.

Lo mejor es encontrar un promedio de la cantidad de ciclos para un quantum dado que para valores mas grandes también se observó que puede tardar más con respecto a otros casos.

Caso 3

En el último caso se observó el comportamiento para un lote de tareas, a medida que modificabamos el número de núcleos en los que se ejecutaban las mismas.

Las tareas usadas fueron:

```
TaskCPU 7
@2:
TaskCPU 6
@1:
TaskCPU 10
@3:
TaskCPU 4
```

Obteniendo los siguientes resultados:

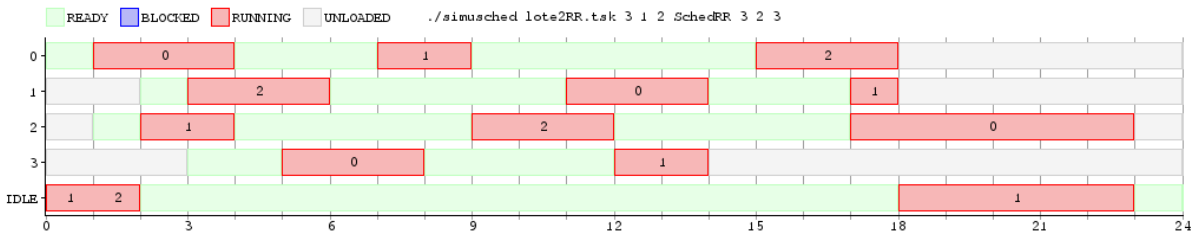


Figura 10: Diagrama de Gantt para la ejecución del lote de tareas bajo 3 núcleos con Scheduler Round Robin.

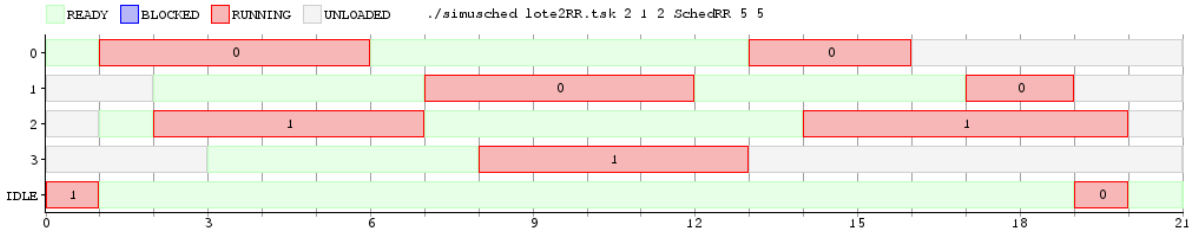


Figura 11: Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos con Scheduler Round Robin.

En este último caso, quisimos comparar el comportamiento de un mismo lote de tareas cuando modificabamos el número de cores bajo un Scheduler Round Robin.

Lo que observamos fue que cuando se contaba con un número mayor de cores, las migraciones de procesos fueron más frecuentes. Debido a que estas no son despreciables -de hecho tienen un costo de 2 ciclos-; al producirse en gran cantidad, el tiempo total para la finalización de tareas aumentó considerablemente.

Como se observa en la **figura 10**, hay un gran lapso de tiempo entre la ejecución de una misma tarea, teniendo en cuenta que en cada quantum se ejecutaron en un core distinto.

En cambio, en la **figura 11** no se prudujeron migraciones, y pese a que se contaba con un core menos, el tiempo fue menor. Pudiendo concluir que aunque las migraciones de procesos pueden ser provechosas como se observó en el *Caso 2* (Sección 2.2), si se producen en exceso pueden perjudicar la perfomance del procesador.

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

A partir del artículo Liu, Chung Laung, and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM) 20.1 (1973): 46-61.

2.3.1. ¿Qué problema están intentando resolver los autores?

Frente al avance que se venía produciendo esos años en el uso de las computadoras para distintos procesos industriales, los autores establecieron que este comportamiento se iría masificiando aún más y que una correcta aplicación solo es posible cuando se tiene un Scheduler eficiente y factible. Lo consideraron primordial en aquellos usos para los cuales se requerían que las tareas llevadas a cabo se ejecuten dentro de un lapso de tiempo determinado (esto lo llaman *"hard-real-time"*). En caso contrario, una respuesta tardía podría provocar un efecto no deseado en una aplicación o no sería de interés.

Otro motivo por el cual intentaron desarollar este tema fue, como describen a lo largo del paper, que hasta entonces estaban orientados a contextos en los que no existía una cota de tiempo para llevar a cabo todas las tareas. Y los que poseían hacían uso de varios supuestos, llevando a situaciones irreales ciertos casos.

Por estos motivos, plantearon dos modelos de Scheduler para tareas en un entorno de *hard-real-time*. Ambos se basan en fijar prioridades para cada una de las tareas: uno de manera estática y el otro dinámicamente. Además para mejorar el rendimiento del mismo, buscan establecer cotas (cuando es posible) para determinar si es aplicable alguno de los modelos de manera que no se produzca *overflow* (alguna de las tareas finalice su ejecución después del tiempo límite).

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

En la sección 7 se introduce un nuevo algoritmo, el cual es para un scheduler basado en asignaciones de prioridades dinámicamente (llamado *The deadline Driven Scheduling*). Lo introducen como una variante al algoritmo de prioridades fijas (presentado previamente), intentando evitar o disminuir algunos de los problemas presentados para dicho modelo.

Una de los problemas a resolver es mejorar las asignaciones de las prioridades, de manera de aprovechar el procesador a un 100 %. A diferencia del modelo de prioridades fijas, donde la utilización del procesador podía variar de un 70 % a un 100 %. De esta forma, este nuevo algoritmo establece que el nuevo modelo nunca va a ejecutar la Tarea Iddle, sino que siempre se va a estar ejecutando alguna tarea.

Para conseguir esto, se va a aumentar o disminuir la prioridad de una tarea conforme se acerca a su deadline, es decir las prioridades de una misma tarea van a ser modificadas a lo largo de su ejecución. Además, buscaron determinar si un conjunto de tareas evitaría producir overflow, con este algoritmo.

En el caso del scheduler con prioridades fijas, se estableció una condición para tal motivo (*teorema 4* [1]), pero esta es sólo necesaria. Para el scheduler presentado en esta sección se desarrolló una nueva condición, *teorema 7*, la cual garantiza que si se cumple entonces es aplicable el algoritmo y en caso contrario, no lo es (el mismo sera explicado en el siguiente punto).

[1] Scheduling Algorithms for Multiprogramming in a HardReal-Time Environment. C. L. LIU Project MAC, Massachusetts Institute of Technology AND JAMES W. LAYLAND Jet Propulsion Laboratory, California Institute of Technology. pág 6

2.3.3. Explicación coloquial del teorema 7

Teorema 7:[2]
For a given set of m tasks, the deadline driven scheduling algorithm is feasible if and only if

$(C_1/T_1) + (C_2/T_2) + \ldots + (C_m/T_m) \leq 1$

Siendo C_i el tiempo de ejecución para una tarea i, $1 \leq i \leq m$, y T_i su respectivo período.

En las secciones anteriores al presente teorema, se determina que la fracción de CPU que utiliza una tarea en el sistema esta dado por C_i/T_i (*the utilization factor*). Lo que se plantea en este teorema es que un Scheduler, para una cantidad m de tareas, es factible y por lo tanto realizable sin que se produzca overflow cuando la suma del costo de CPU de cada tarea es menor que uno o igual a 1. Es decir que el ejecutar todas las tareas este dentro de la capacidades del CPU (no supere el 100 % del rendimiento del mismo), ya que en caso contrario se estaria exigiendo que el CPU trabaje a un límite mayor, lo cual es imposible.

2.3.4. Scheduler

Diseñar e implementar un scheduler basado en prioridades fijas y otro en prioridades dinámicas. Para eso, complete las clases SchedFixed y SchedDynamic que se encuentran en los archivos sched fixed.[h—cpp] y sched dynamic.[h—cpp] respectivamente.

Implementaciones:

Dado el contexto temporal en el que se plantearon ambos scheduler la cantidad de cores con los que los implementaremos será igual a 1.

Scheduler con prioridades fijas:

Las estructuras de datos con las que vamos a trabajar, en la clase *SchedFixed*, son las siguientes:

```
struct datos{
    int pid;
    int periodo;
    bool operator< (const datos& otro) const{
        return otro.periodo <= periodo;
    }
};

vector<int> periodos;
priority_queue<datos> pq;
```

- **struct datos** Diseñado para representar a una tarea, en él se podra encontrar el PID y el período de la misma.
- **periodos** vector donde cada posición representa un PID (dado que estos se enumeran desde el cero y se van incrementando en uno) en las mismas se encuentra el período que le fue asignado a cada tarea.

[2]Scheduling Algorithms for Multiprogramming in a HardReal-Time Environment. C. L. LIU Project MAC, Massachusetts Institute of Technology AND JAMES W. LAYLAND Jet Propulsion Laboratory, California Institute of Technology. pág 10

- **pq** cola de prioridad de datos donde la prioridad más alta la tiene aquel dato cuyo periodo es el menor. Dado que se esta implementando el scheduler con prioridades fijas, en él se atribuye que la prioridad esta dada por:
  $prioridad = 1/periodo$
  Es decir a menor periodo mayor prioridad, es por esto que la cola siempre va a devolver como primer elemento el más priorítario respecto del período mínimo.

Inicializando Scheduler Fixed, función initizalize

Al construir el Scheduler basado en prioridades fijas, como primer medida se inizializó *periodos*, el mismo contiene el período asignado a cada tarea. Para esto, se utilizó la función *period* de esta manera en cada posición guardamos el correspondiente período asignado.

Función Load

Recibe el *pid* de una nueva tarea como parámetro. Procede a crear una tarea del tipo *datos* con el *pid* recibido y el período del mismo que se puede obtener de *periodos* (*periodos[pid]*). Finalizado esto, se encola la tarea en *pq*.

Función Tick

Recibe *cpu* como parámetro y el *Motivo*. Dado que se produjo un tick, se procede a actualizar (de ser necesario) la asignación del *cpu* para las distintas tareas. Es por esto, que si la tarea que se estaba ejecutando hasta entonces terminó, o si se estaba ejecutando la IDDLE, y existe otra tarea en *pq*; la próxima a ejecutar será la primera de ella (la de menor período), en caso contrario se ejecutara la IDDLE.

En el caso de no haber terminado la tarea, se procede a encolarla en *pq*. Si existe una tarea con mayor priodidad, será devuelta como próxima a ejecutar y sino será la misma que se acabó de encolar.

Scheduler con prioridades dinámicas:

Las estructuras utilizadas, en la clase *SchedDynamic*, son las siguientes:

```
struct datos{
    int pid;
    int deadline;
    bool operator< (const datos& otro) const{
        return otro.deadline <= deadline;
    }
};

priority_queue<datos> pq;
datos actual;
bool primera;
```

- **struct datos** Al igual que para el scheduler anterior, está diseñado para representar a una tarea. En él, se encontrará el *PID* y el *período* de la misma.
- **pq** cola de prioridad de *datos* donde la prioridad más alta la tiene aquel cuyo deadline sea el menor. Como se está implementando un scheduler con prioridades dinámicas, en el mismo se atribuye que la prioridad mas álta la tiene aquella tarea cuyo deadline esta mas próximo a finalizar. Por lo que *pq* va a devolver como primer elemento (el de mayor prioridad) a la tarea con deadline mínimo.
- **actual** va a representar a la tarea ejecutandose actualmente.
- **primera** Se utiliza para saber si hay una tarea asignada a *actual*, la misma se inicializa con True en el constructor.

Función Load

Recibe el *pid* de una nueva tarea como parámetro y procede a crear la tarea asociada a ese *pid*, el deadline de la misma será (inicialmente) su período. Finalizado esto, se encola la tarea en *pq*. En caso de que *primera* sea true, se le asigna a *actual* esta última tarea creada.

Función Tick

Recibe *cpu* como parámetro y el *Motivo*. Al producirse un tick se procede a actualizar la asignación del *cpu* para las distintas tareas. Si la que se estaba ejecutando hasta entonces terminó, o si se estaba ejecutando la IDDLE, y existe otra tarea en *pq*: la primer tarea de la cola será la próxima a ejecutar (la de menor deadline), en caso contrario se ejecutara la IDDLE. En el caso de no haber terminado la tarea, se procede a encolar a *actual* en *pq* pero, disminuyendo su deadline en uno ya que la tarea completo un TICK. Si existe una tarea con mayor prioridad, será devuelta como próxima a ejecutar y sino, será la misma que se acaba de encolar.

3. Parte III

3.1. Ejercicio 6: TaskBatch

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 pseudoaleatoriamente. 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).

Nuestra función arma un vector de *cant_bloqueos* posiciones y les asigna para cada una: un valor pseudoaleatorio, sin repetidos, dentro de su rango de tiempo de ejecución *total_cpu*. A estos valores se los ordena dentro del arreglo para poder recorrerlo secuencialmente.

Luego, para cada ciclo de clock se pregunta si ese momento está indicado para realizar una llamada bloqueante. En caso afirmativo, se ejecuta *uso_IO(pid 1)*; en caso contrario, *uso_CPU(pid 1)*.

Debemos aclarar que nuestro algoritmo está definido sólo para casos donde alcanzan los clocks para ejecutar las llamadas bloqueantes requeridas (el caso contrario no nos es de interés).

3.2. Ejercicio 7: Ejecución lote de tareas TaskBatch

Elegir al menos dos métricas diferentes, definirlas y explicar la semántica de su definición. Diseñar un lote de tareas TaskBatch, todas ellas con igual uso de CPU, pero con diversas cantidades de bloqueos. Simular este lote utilizando el algoritmo SchedRR y una variedad apropiada de valores de quantum. Mantener fijo en un (1) ciclo de reloj el costo de cambio de contexto y dos (2) ciclos el de migración. Deben variar la cantidad de núcleos de procesamiento. Para cada una de las métricas elegidas, concluir cuál es el valor óptimo de quantum a los efectos de dicha métrica.

Las métricas elegidas fueron: Tiempo de Respuesta y TournAround.

Tiempo de Respuesta: Tiene en cuenta el tiempo que tarda cada proceso en empezar a ejecutarse desde su primera aparición en *ready*. Cuanto menor tiempo de Respuesta tienen los procesos, el usuario adquiere una mayor interactividad.

TournAround: Tiene en cuenta el tiempo que utiliza un proceso en ejecutarse completamente, es decir desde su primera aparición en *ready* hasta que termina su ejecución.

El lote de Tareas TaskBatch diseñado es el siguiente:

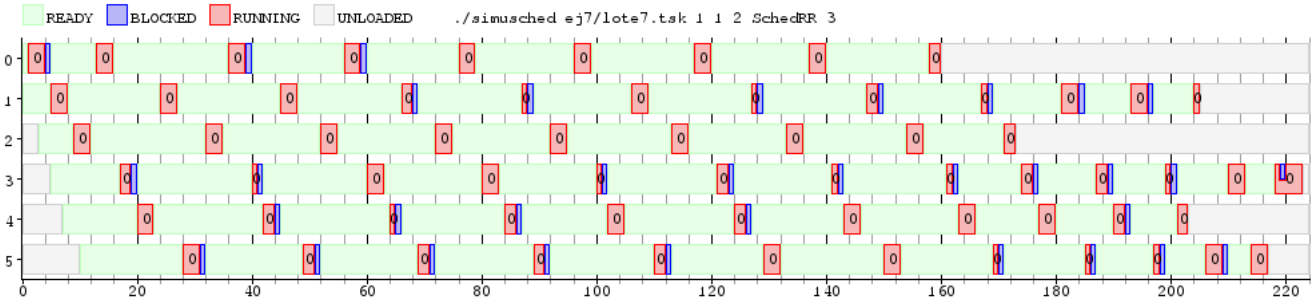
```
TaskBatch 25 3
TaskBatch 25 7
@3:
TaskBatch 25 0
@5:
TaskBatch 25 10
@7:
TaskBatch 25 5
@10:
TaskBatch 25 9
```

Ejecutamos este lote de tareas, considerando diferentes cantidades de núcleos variando dentro de ellas el quantum de cada uno (En los casos A se eligieron los quantums más chicos, en los C los más grandes y en los B los intermedios). En cada caso, tomamos el promedio de los valores de *Tiempo de Respuesta* y *TournAround* para luego comparar cualitativamente.

Para cada caso a tratar, ejecutamos 100 veces el mismo comando para luego sacar un promedio de las mediciones hechas ya que estamos trabajando con números (pseudo)random. Exponemos un ejemplo tomado de cada caso a continuación:

1 Núcleo

Caso A: El quantum de este núcleo es de 3 clocks.

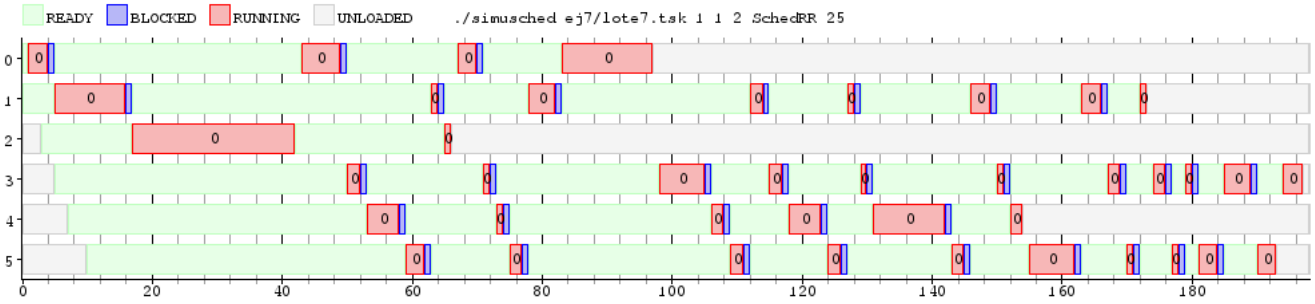


Tiempo de Respuesta: 9.166666667
Tiempo de TournAround: 192.6666667

Caso B: El quantum de este núcleo es de 10 clocks.

Tiempo de Respuesta: 18.5
Tiempo de TournAround: 152.8333333

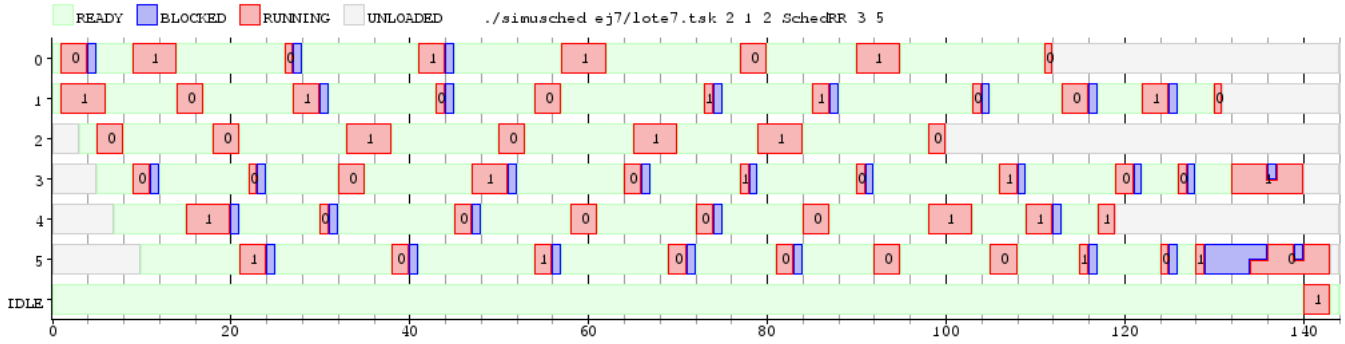
Caso C: El quantum de este núcleo es de 25 clocks.



Tiempo de Respuesta: 26.66666667
Tiempo de TournAround: 142.5

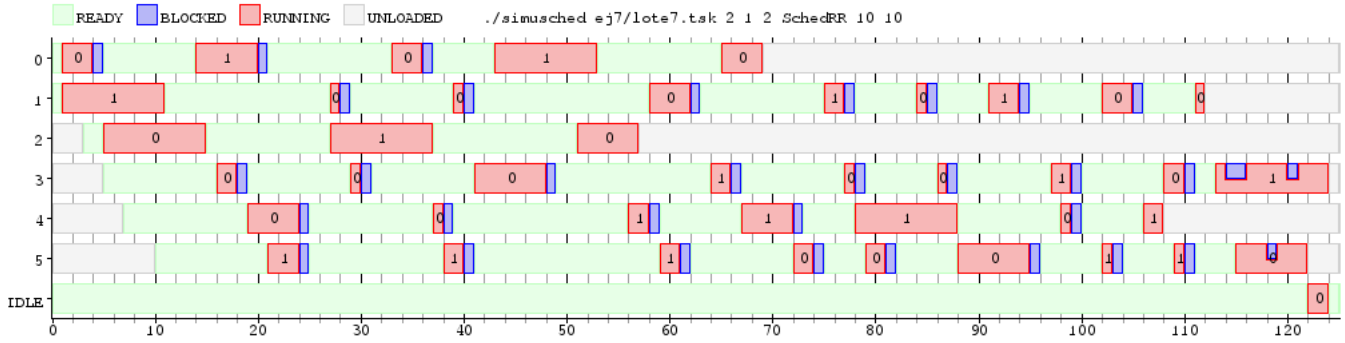
2 Núcleos

Caso A: Un quantum es de 3 y el otro de 5 clocks.



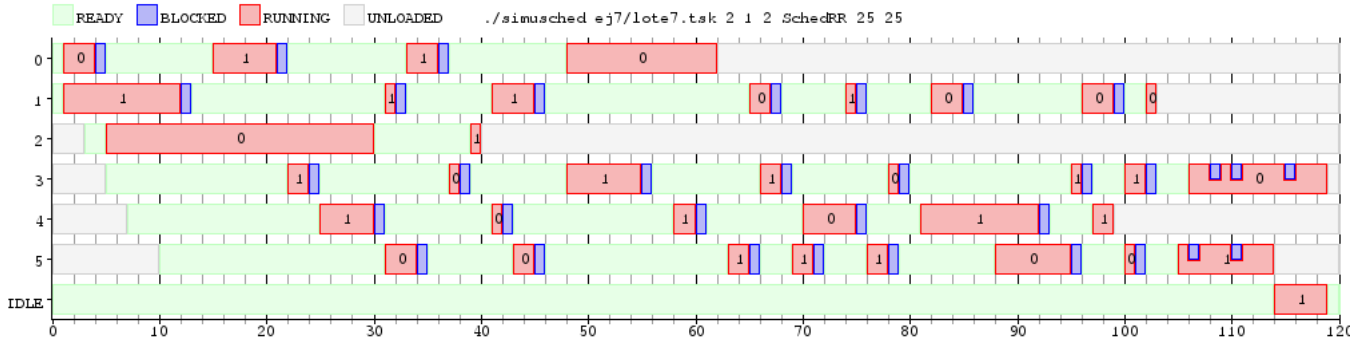
Tiempo de Respuesta: 4.5
Tiempo de TournAround: 120

Caso B: Los dos quantums son de 10 clocks.



Tiempo de Respuesta: 6.333333333
Tiempo de TournAround: 94.5

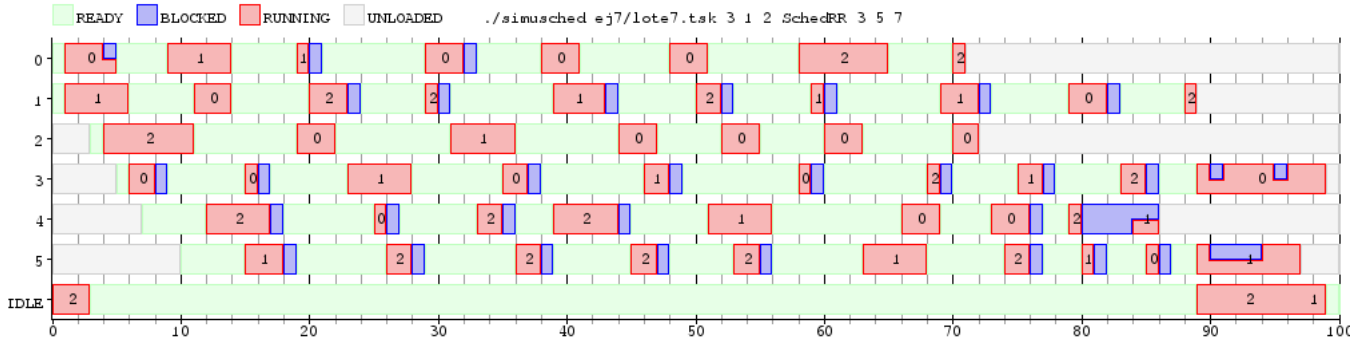
Caso C: Los dos quantums son de 25 clocks.



Tiempo de Respuesta: 10
Tiempo de TournAround: 85.33333333

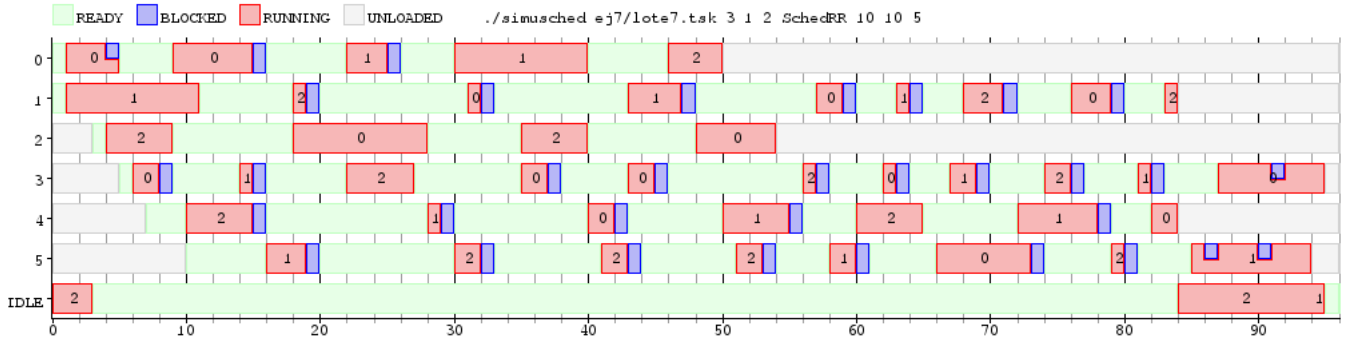
3 Núcleos

Caso A: Los quantums son de: 3, 5 y 7 clocks.



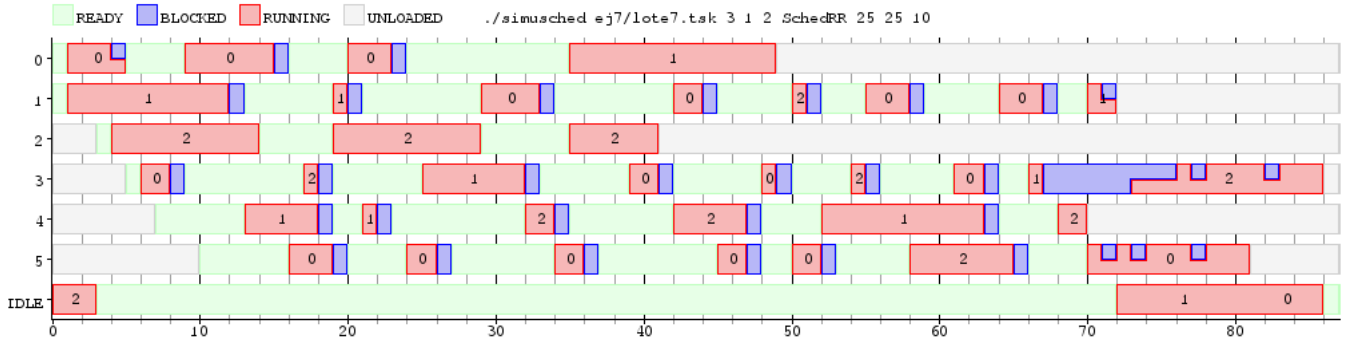
Tiempo de Respuesta: 2.333333333
Tiempo de TournAround: 81.5

Caso B: Los quantums son de: 10, 10 y 5 clocks.



Tiempo de Respuesta: 2.166666667
Tiempo de TournAround: 72.66666667

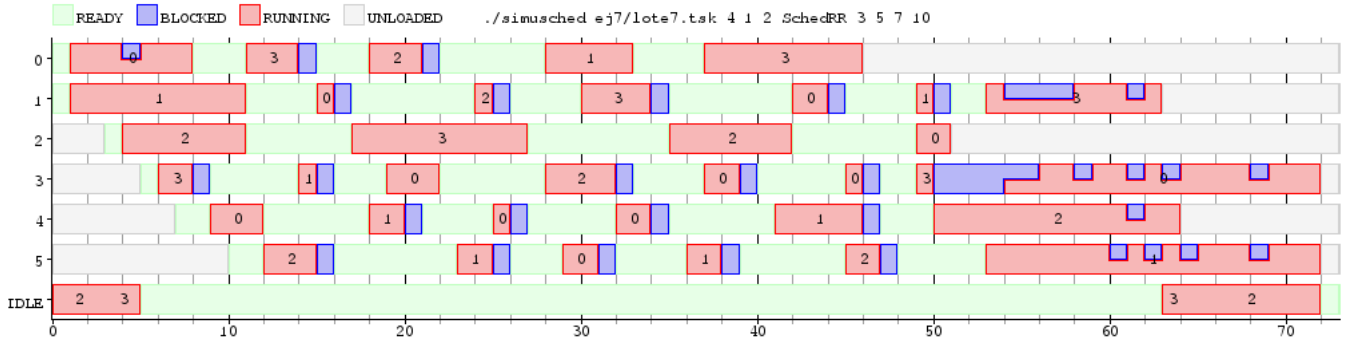
Caso C: Los quantums son de: 25, 25 y 10 clocks.



Tiempo de Respuesta: 2.666666667
Tiempo de TournAround: 62.33333333

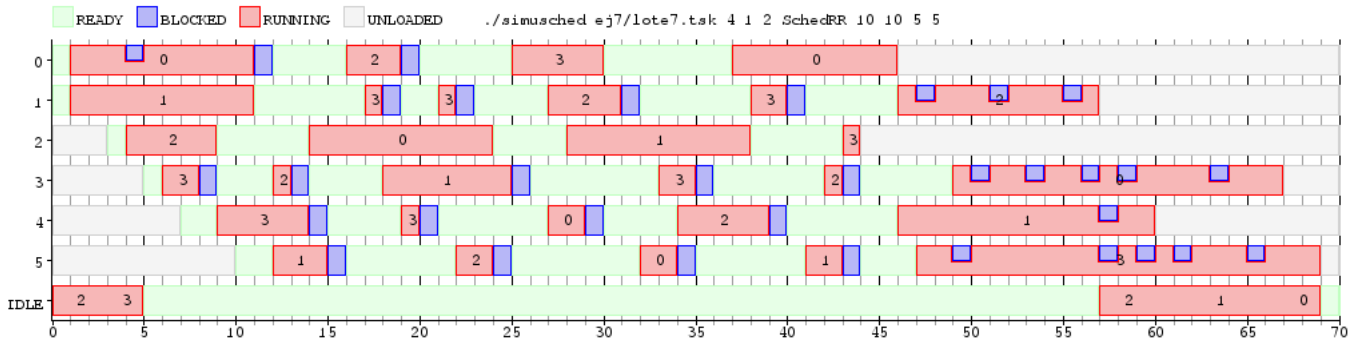
4 Núcleos

Caso A: Los quantums son de: 3, 5, 7 y 10 clocks.



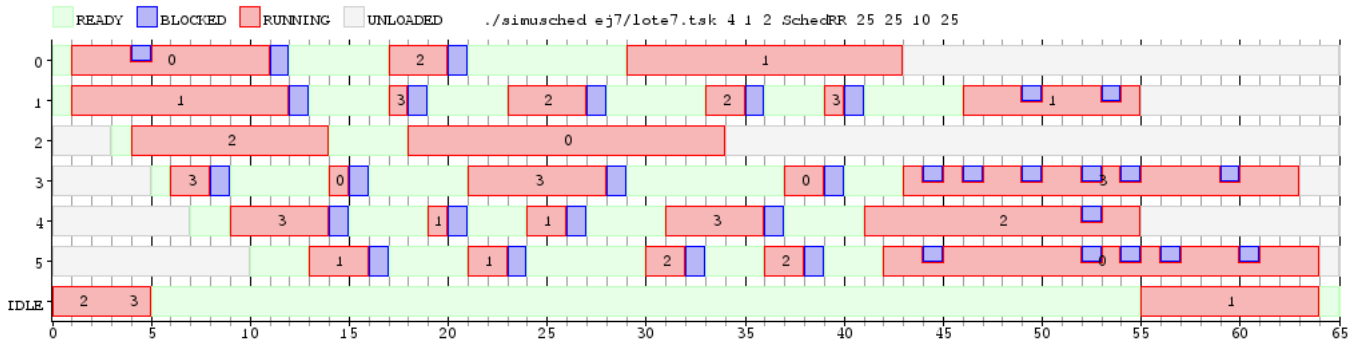
Tiempo de Respuesta: 1.333333333
Tiempo de TournAround: 57.16666667

Caso B: Los quantums son de: 10, 10, 5 y 5 clocks.



Tiempo de Respuesta: 1.333333333
Tiempo de TournAround: 53

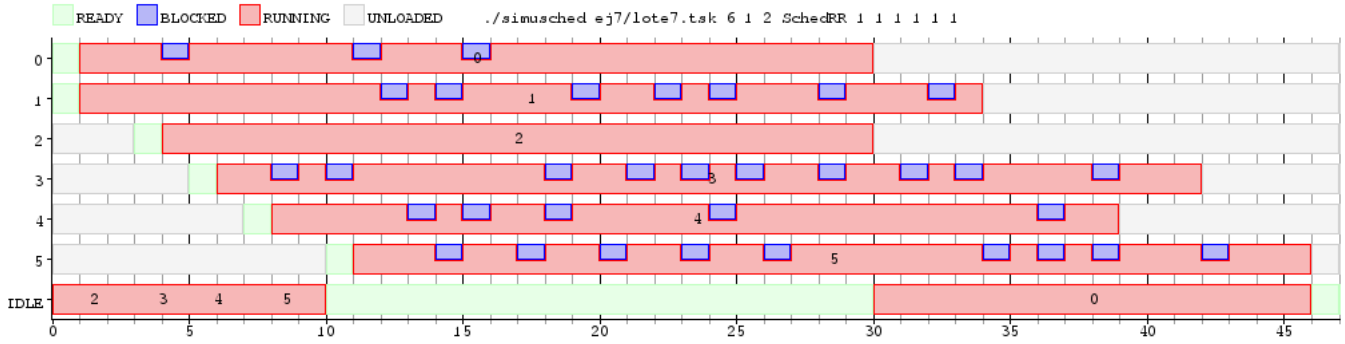
Caso C: Los quantums son de: 25, 25, 10 y 25 clocks.



Tiempo de Respuesta: 1.5
Tiempo de TournAround: 48.16666667

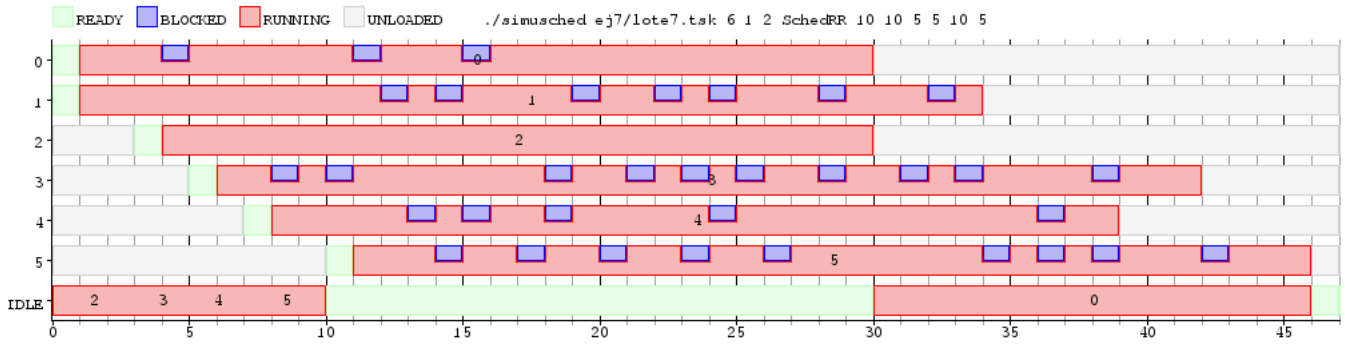
6 Núcleos

Caso A: Los quantums son todos de 1 clock.



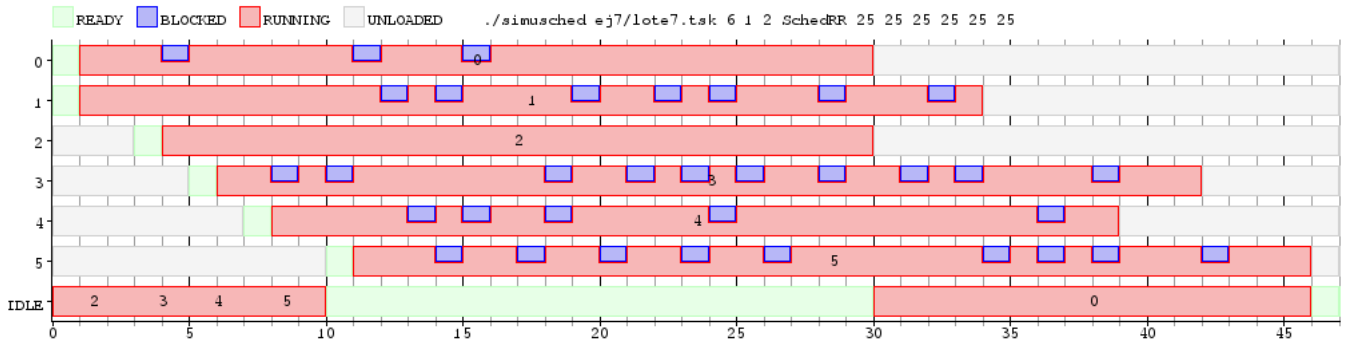
Tiempo de Respuesta Promedio: 1
Tiempo de TournAround Promedio: 32.66666667

Caso B: Los quantums son de: 10, 10, 5, 5, 10, 5 clocks.



Tiempo de Respuesta Promedio: 1
Tiempo de TournAround Promedio: 32.66666667

Caso C: Los quantums son todos de 25 clocks.

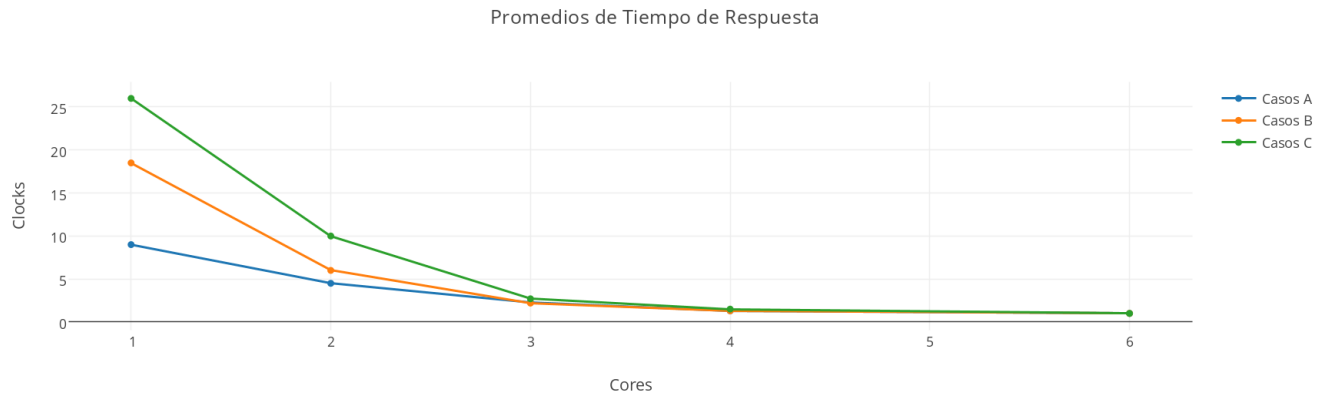


Tiempo de Respuesta Promedio: 1
Tiempo de TournAround Promedio: 32.66666667

Luego graficamos los tiempos obtenidos, con el fin de tener una herramienta más declarativa para la comparación de casos bajo las dos métricas usadas.

Métrica de Tiempo de Respuesta

Graficamos los promedios de tiempo de respuesta.



Acorde a lo que resulta intuitivo, estos valores siempre descienden acorde aumenta la cantidad de cores. Los casos con una cantidad de núcleos mayor a seis fueron descartados, ya que resultan triviales dado a que la cantidad de tareas es seis (son análogos al caso de 6 núcleos dejando el resto ejecutando la Tarea Iddle).

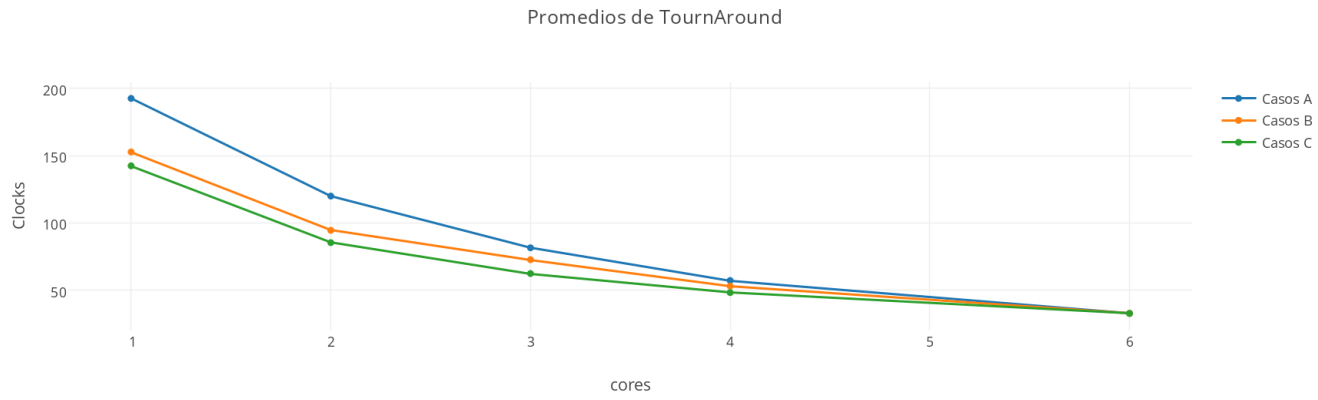
En el gráfico se puede apreciar la relación que preservan los casos A, B y C indiferentemente de la cantidad de núcleos existentes: los casos con quantum de mayor tamaño arrojaron valores más altos. Este comportamiento se debe a que, al momento de ingresar una tarea debe esperar a que algún core quede libre (es decir, la tarea que estaba ejecutando haya terminado su quantum o quede bloqueada).

En los casos donde la cantidad de tareas es menor o igual a la cantidad de núcleos, el promedio de los tiempos de respuesta siempre va a ser el tiempo que tarde en cargarse una tarea. En nuestro caso este tiempo es 1 clock.

De este modo, podemos definir que para tiempos de respuesta óptimos no sólo debemos contar con la mayor cantidad de núcleos posibles (siempre menor igual o a la cantidad de tareas, ya que una cantidad mayor carece de mejoras) sino que también con (al menos un) core que tenga un quantum de tamaño pequeño. Esto nos asegura que al momento de ingresar una nueva tarea, el tiempo que debe esperar hasta que algún core quede libre es menor.

Métrica de TournAround

Graficamos los promedios de tiempos de TournAround.



En primera instancia, debemos marcar que el hecho de que la cantidad de cores sea igual a la cantidad de tareas no sólo otorga un tiempo pequeño sino que es el óptimo ya que al poseer más núcleos estos quedarían libres ejecutando continuamente la Tarea Iddle. Resulta intuitivo marcar que acorde aumenta la cantidad de cores, descienden los promedios de tiempos ya que cada tarea tiene su núcleo para ejecutar sin importar el quantum debido a que nadie la va a desalojar.

Bajo esta métrica, la relación que se preserva entre los casos A, B y C es inversamente proporcional a la métrica anterior. Contando con quantums más pequeños, el tiempo que lleva ejecutar una tarea en total es menor ya que se ejecuta con una frecuencia más alta. Hay que tener en cuenta que contar con quantums menor es beneficioso en todos los casos mostrados, ya que se usó un tiempo de cambio de contexto de 2 clocks lo que resulta un tiempo pequeño.

En el caso de ambas métricas tratadas, se utilizaron la ejecución de cada caso 100 veces. Esto independiza los gráficos obtenidos de los bloqueos que pudieron ser causados, ya que fueron elegidos aleatoriamente.

3.3. Ejercicio 8: Scheduler Round-Robin modificado

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). Diseñe y realice un conjunto de experimentos que permita evaluar comparativamente las dos implementaciones de Round-Robin.

Las estructuras de datos con las que vamos a trabajar, en la clase *SchedRR2*, son las siguientes:

```
int cores;
vector<int> quantums;
vector<int> quantums_timer;
vector<int> actuales;
int siguiente;
vector<pair<int, queue<int, deque<int>>*>> colas;
```

- **cores** cantidad de núcleos.
- **quantums** vector de *cores* posiciones, donde quantums[i] es el valor del quantum asignado al core i.
- **quantus_timer** vector tal que en quantus_timer[i] se encuentra el valor del quantum restante para la tarea corriendo en el i-esimo núcleo.
- **actuales** vector que indica el PID de la tarea ejecutandose en el núcleo i (actuales[i]). Para los núcleos sin tarea devuelve la constante *IDLE_TASK*.
- **siguiente** indica el core al que se le debe asignar la próxima tarea.
- **colas** vector de tuplas. Donde cada posición reresenta a un core y la primer componente de la tupla hacer referencia al número de tareas running + blocked + ready en el mismo. Y la segunda componente es una cola FIFO con los respectivos PID's.

Modificamos a la funcion *next* para que reciba un parametro más (enum Motivo { TICK, BLOCK, EXIT }).

```
    int next(int cpu, const enum Motivo m);
```

Constructor Scheduler Round-Robin Modificado

Al construir un Scheduler Round-Robin sin migración de procesos, empezamos por instanciar las estructuras de datos de modo que: a *cores* se le asigna la cantidad de cores correspondientes. A *quantums_timer* los respectivos quantums; Todas las tareas actuales se definen como Iddle en *actuales*; Se inicia cada tupla de *colas* en cero ya que aún no hay tareas y las colas vacias; Por último *siguiente* es el núcleo al que le corresponde ejecutar la primer tarea, es decir al 0.

Función Load

Recibe el *pid* de una nueva tarea como parámetro. Al igual que para el Scheduler Round Robin, adoptamos distribuir las primeras n tareas entre cada core (dado que al inicio hay n cores disponibles). Además, se incrementa la primer componente de cada tupla en uno (ahora en cada core tengo una tarea).

Una vez alcanzadas las n distribuciones de las futuras tareas, serán asignadas a los CPU's con menor cantidad de tareas *running + blocked + ready*. Es decir, se encolarán sus pid's en las colas de cada CPU de *colas* con la primer componente más baja, incrementado ahora dicha componente.

Función Tick

Recibe *cpu* como parámetro y el *Motivo*. Dado que se produjo un tick, voy a actualizar el quantum (restarle 1) de la tarea en *cpu*, siempre que no sea la IDLE. Si se acabó el quantum de la tarea actual (ya sea porque terminó o no) o se bloqueó, voy a querer actualizar la tarea actual. Invocando a la función *next()*. La misma devuelve el *pid* de la próxima tarea a ejecutar en *cpu*.

Función Next

Si la tarea que se estaba ejecutando previo a la invocación de *next()* terminó de ejecutarse y la cola se encuentra vacía, la tarea que pondrá a ejecutar va a ser la Tarea Iddle y se actualiza el número que indica cuantas tareas hay en ese cpu.

Si la tarea que se estaba ejecutando no terminó su ejecución o se bloqueo, se deberá encolar en la cola del *cpu* pasado por parametro.

Si todavía no asigné una tarea actual para el mismo, le asigno la primera de la cola dándole todo el quantum disponible para el núcleo *cpu*. (Si la cola se encontraba vacía al llamar la función next() y la tarea ejecutándose no había concluído se la encolará para luego volver a asignársela al núcleo).

Para evaluar el comportamiento del Scheduler Round Robin sin migraciones frente al diseñado en la Sección 2.1, se construyo un lote de 5 tareas. Cada lote esta compuesto por 3 tareas del tipo TaskConsola, donde la cantidad de bloqueos era un valor random entre 1 y 20 al igual que el tiempo mínimo y máximo de bloqueo; y dos del tipo TaskBatch con cantidad de llamadas bloqueantes y tiempo de CPU random entre 1 y 20. Este lote fue ejecutado un total de 10 veces. Para comparar los resultados se útilizó tanto la métrica tiempo de respuesta es decir, la cantidad de tiempo que los procesos tardan en empezar a ejecutarse desde que se encuentran *ready*, como Turnaround (tiempo que les toma ejecutarse completamente). Para ambos casos se utilizó el prómedio de los valores de los distintos lotes mencionados.
Lor resultados obtenidos fueron los siguientes (los diagramas de Gantt, los lotes de tareas, así como los tiempos obtenidos para cada uno se pueden encontrar en TP1/Informe/imagenes/ej8/Anexo):

3.4. Tiempo de respuesta:

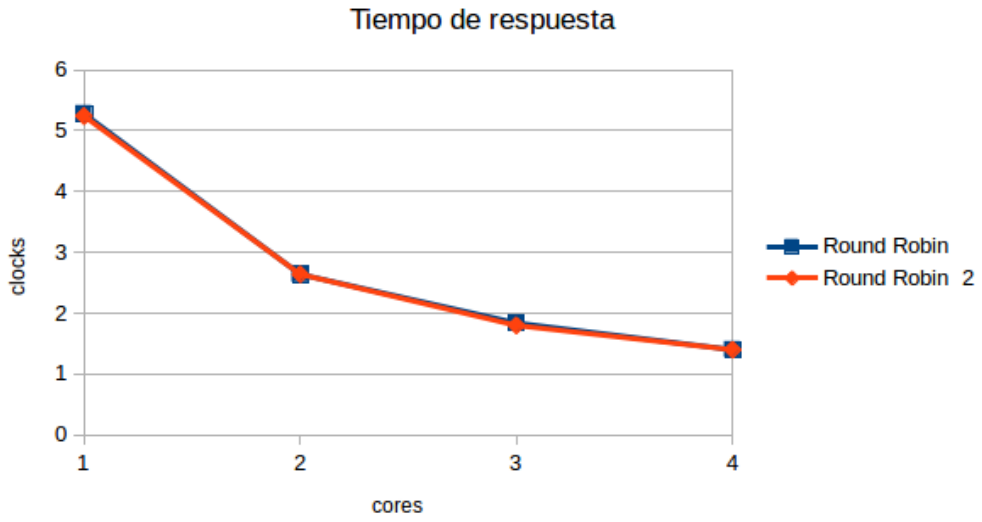


Figura 12: Gráfico de lineas para evaluar el tiempo prómedio de respuesta de las tareas bajo un scheduler Round Robin y Round Robin sin migración de procesos (Round Robin2) cada core con quantum igual a 3

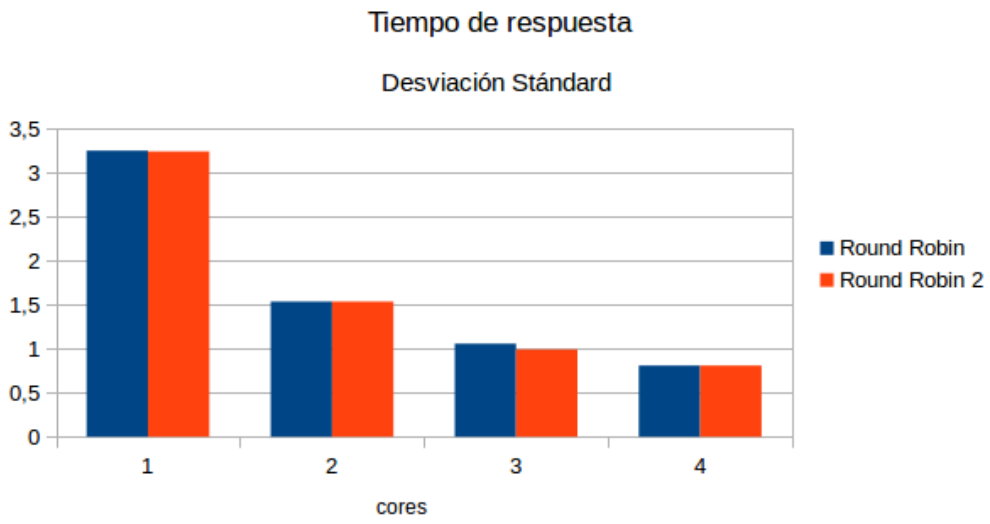


Figura 13: Desviación Standard de los datos de la figura 12.

Para esta primera métrica observamos que el tiempo promedio para ambos casos, así como su Desvio Standard son similares. Esto se debe primordialmente al diseño adoptado para ambos Scheduler, como ya se mencionó comenzamos ditribuyendo las primeras n tareas entre los n núcleos disponibles. De esta mánera, el comienzo de la ejecución, así como donde lo realiza, es igual en ambos casos.

3.5. Turnaround:

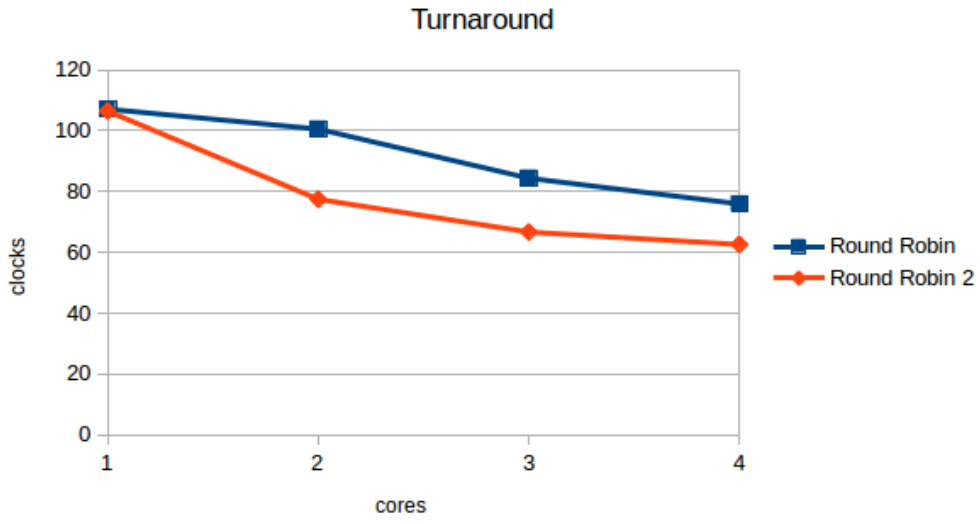


Figura 14: Gráfico de lineas para evaluar el Turnaround prómedio de las tareas bajo un scheduler Round Robin y Round Robin sin migración de procesos (Round Robin2) cada core con quantum igual a 3

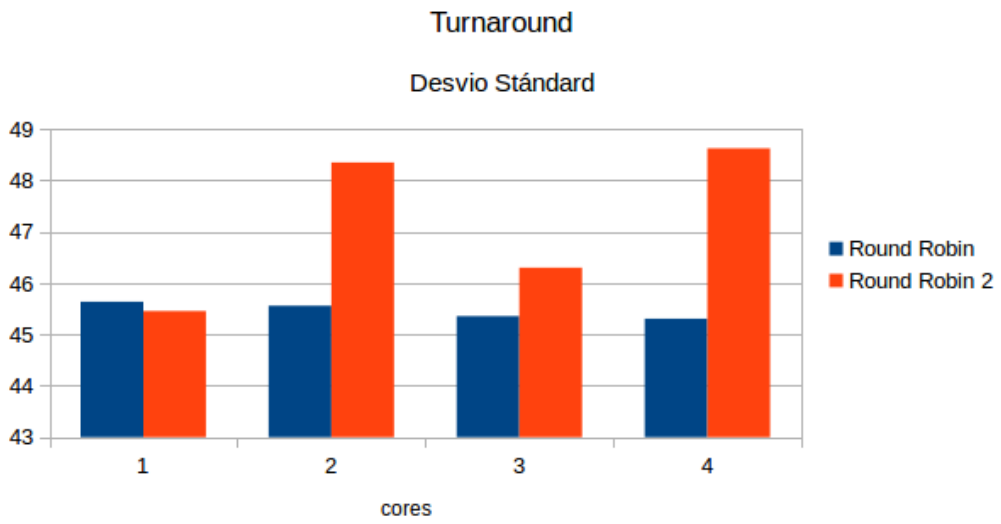


Figura 15: Desviación Standard de los datos de la figura 14.

Al momento de evaluar los resultados con la métrica Turnaround nuestra hipótesis fue que para el caso de Round Robin2 los tiempos iban a ser menores, como se puede ver en la figura 14 para el lote de tareas utilizado nuestra hipótesis es correcta. Lo que nos llevó a formular esta hipótesis fue que si un lote de tareas realiza migraciones de procesos en exceso, o en una cantidad no despreciable, esto se iba a ver bruscamente reflejado en el tiempo total que le tomaba ejecutarse. En nuestro caso utilizamos que el costo de migración era igual 2 ya que el cambio de contexto era igual a 1 y atribuimos que migrar un proceso entre cores es mas costoso que un cambio de contexto. Para ver el comportamiento especifico de un lote de tarea usamos las de la pág 9

```
 TaskCPU 7
@2:
TaskCPU 6
@1:
TaskCPU 10
@3:
TaskCPU 4
```

Al ejecutarlas en el Scheduler Round Robin modificado los resultados fueron los siguientes:

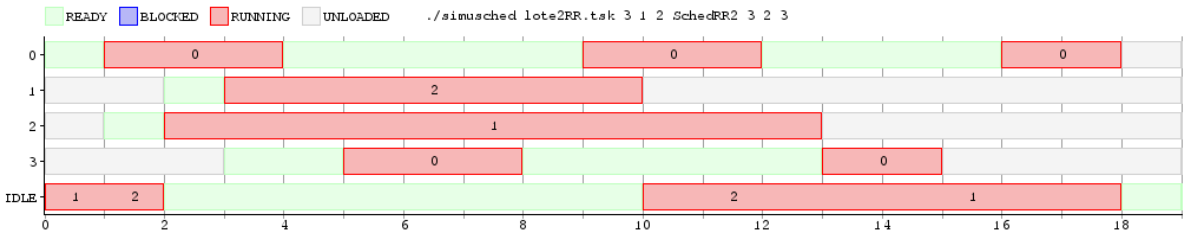


Figura 16: Diagrama de Gantt para la ejecución del lote de tareas bajo 3 núcleos con quantums de 3, 2 y 3 ciclos respectivamente. En un Scheduler Round Robin sin migración de procesos.

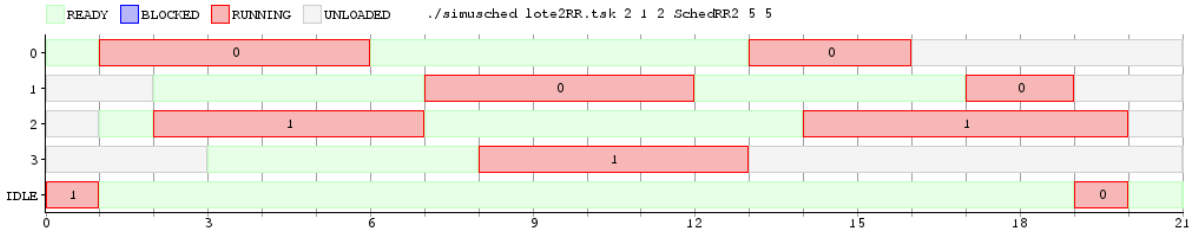


Figura 17: Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos con quantums de 5 y 5 ciclos respectivamente. En un Scheduler Round Robin sin migración de procesos.

Si comparamos la figura 10 con la 16 y la figura 11 con la número 17 vemos que para los primeros casos el tiempo total de ejecución fue mayor. Se puede observar, en las figuras 10 y 11, que los procesos realizan muchos cambios de migración aumentando el intervalo de tiempo en el que no se estan ejecutando en comparación a las figuras 16 y 17. Si bien no podemos concluir que permitir migraciones sea desfavorable en un Scheduler, lo que podemos decir es que deben realizarse de manera cuidadosa y sin abusar de la cantidad permitida.

3.6. Ejercicio 9: Ejecución lote de tareas

Diseñar un lote de tareas cuyo scheduling no sea factible para el algoritmo de prioridades fijas pero sí para el algoritmo de prioridades dinámicas.

4. Conclusión: