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**

En el presente trabajo se describe la problemática de ...

# Índice

**1. Parte I** **4**
1.1. Ejercicio 1: TaskConsola . . . . . 4
1.2. Ejercicio 2: Ejecución de tres tareas . . . . . 4

**2. Parte II** **6**
2.1. Ejercicio 3: Scheduler Round-Robin . . . . . 6
2.2. Ejercicio 4: Ejecución de lotes de tareas . . . . . 7
2.3. 1er caso: . . . . . 7
2.4. 2do caso: . . . . . 8
2.5. 3er caso: . . . . . 9
2.6. Ejercicio 5: Scheduling algorithms for multiprogramming in a hard-real-time environment 10

**3. Parte III** **11**
3.1. Ejercicio 6: TaskBatch . . . . . 11
3.2. Ejercicio 7: Ejecución lote de tareas . . . . . 11
3.3. Ejercicio 8: Scheduler Round-Robin modificado . . . . . 12
3.4. Ejercicio 9: Ejecución lote de tareas . . . . . 15
3.5. Ejercicio 10: Ejecución lote de tareas . . . . . 16

# 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* (sistema operativo???) n veces, eligiendo cada vez un número al azar -mediante la función rand()- entre bmin y bmax.

## 1.2. Ejercicio 2: Ejecución de tres tareas

*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.*

Para observar el comportamiento de un lote de tareas en un scheduler basado en la técnica FCFS (*First Came, First Served*). Donde cada una se ejecuta en el orden en el que se encuentran $ready$. Se ejecutaron tres tareas distintas. Dos se corresponden el tipo de tarea $TaskConsola$ y la otra solo haciendo 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 como varía el tiempo total (si es que lo hace), se simuló la ejecución para 1, 2 y 3 cores. Para los tres casos se consideró un costo de una unidad para cambiar de contexto y para cambiar un proceso de núcleo de procesamiento.

Los diagramas de Gantt obtenidos fueron los siguientes:

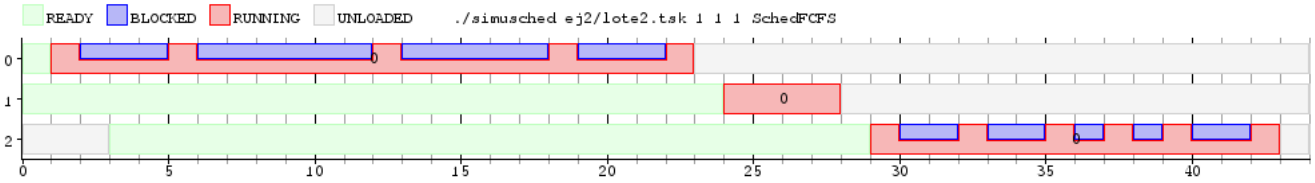


Figura 1: Diagrama de Gantt para la ejecución del lote de tareas bajo 1 núcleo.

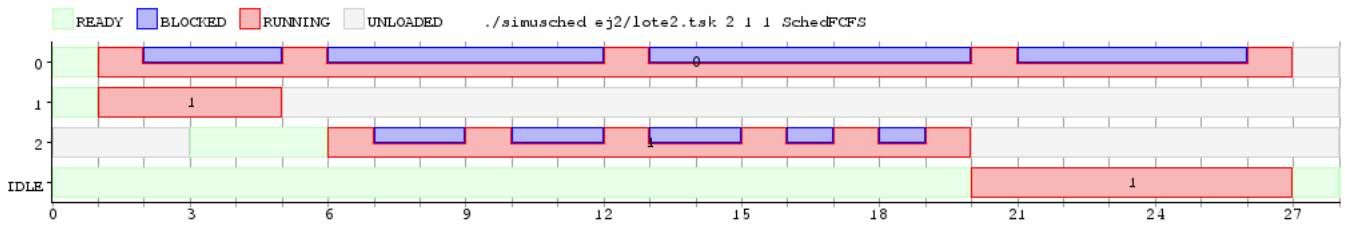


Figura 2: Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos.

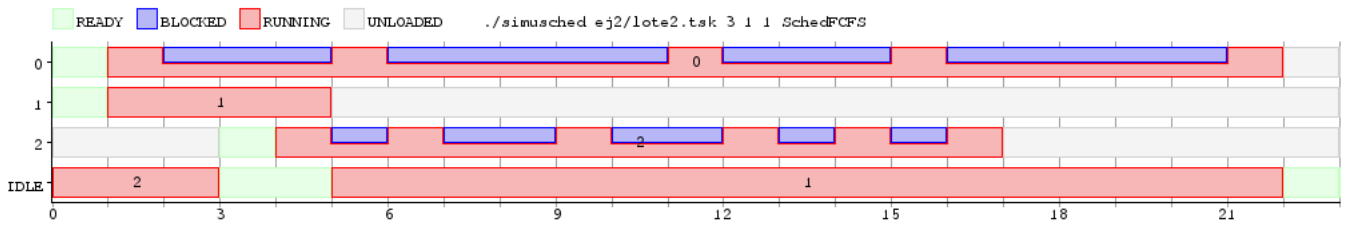


Figura 3: Diagrama de Gantt para la ejecución del lote de tareas bajo 3 núcleos.

Aca habria que hablar un poco y revisar esto: :D

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 el tiempo total disminuye. No solo 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 core. La primer tarea disponible comenzó a ejecutarse. Dado que aun disponemos de otro core, cuando una segunda tarea paso a estar disponible pudo ser ejecutada al instante por el núcleo adicional. Y cuando uno de estos se desocupó se ejecutó la última tarea. Dado que dos tareas pudieron ejecutarse simultaneamente el tiempo total disminuyó. Aún más cuando se trabajó con 3 cores cada tarea pudo ser ejecutada en un núcleo distinto y dado que ambas podian ser ejecutadas mientras se ejecutaban las otras el tiempo total fue igual al máximo entre $release\ time_j + c_j$, donde j indica alguna de las tres tares 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 parametro 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

*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.*

Para poder observar el comportamiento del Scheduler basado en la metodologia Round Robin evaluamos tres casos.

## 2.3. 1er caso:

Ejecutamos el mismo lote de tareas utilizado anteriormente (pág 4), manteniendo los mismos parametros, para asi 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:

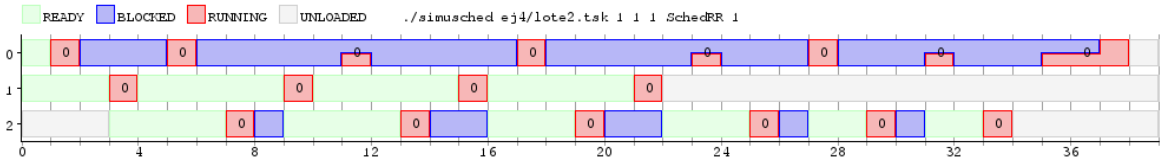


**Figura 4:** Diagrama de Gantt para la ejecución del lote de tareas bajo 1 núcleo con Scheduler Round Robin.

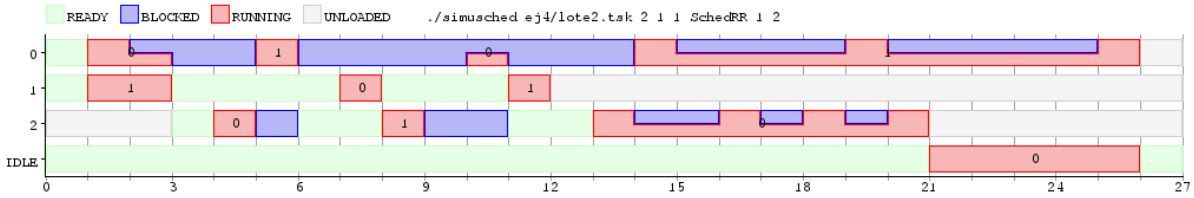


**Figura 5:** Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos con Scheduler Round Robin.

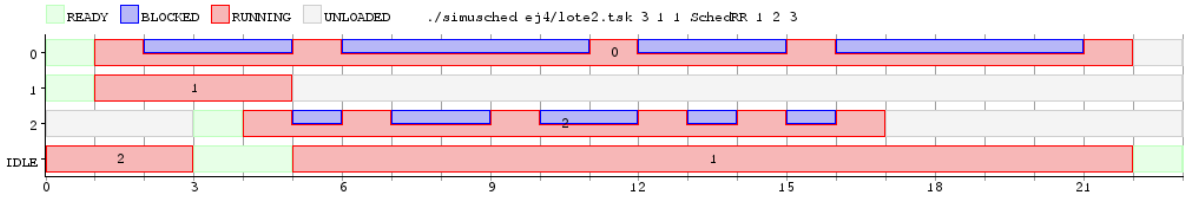


**Figura 6:** Diagrama de Gantt para la ejecución del lote de tareas bajo 3 núcleos con Scheduler Round Robin.

## 2.4. 2do caso:

Para este caso se creo un nuevo lote de tareas y lo que se busco fue analizar como varia 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 siguientes:

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

Los resultados obtenidos son:

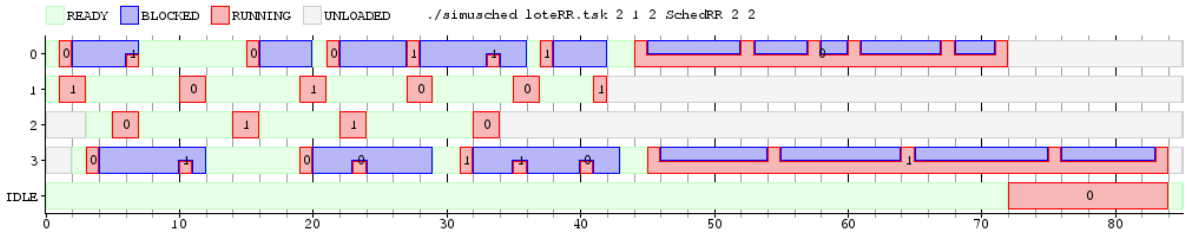


**Figura 7:** Diagrama para la ejecución del lote de tareas bajo 2 núcleos con quantum de 2 y 2 ciclos respectivamente.

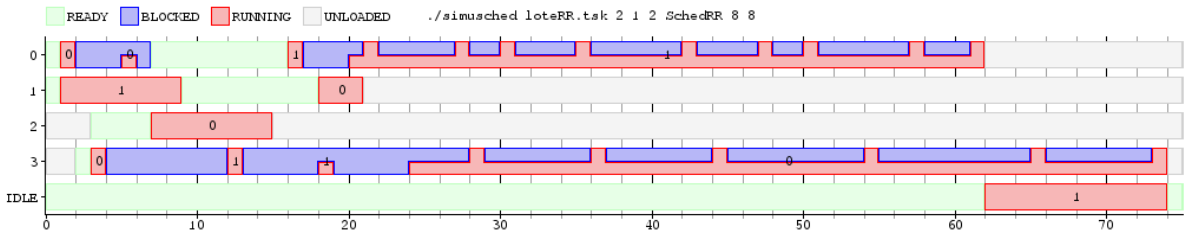


**Figura 8:** Diagrama para la ejecución del lote de tareas bajo 2 núcleos con quantumd de 8 y 8 ciclos respectivamente.

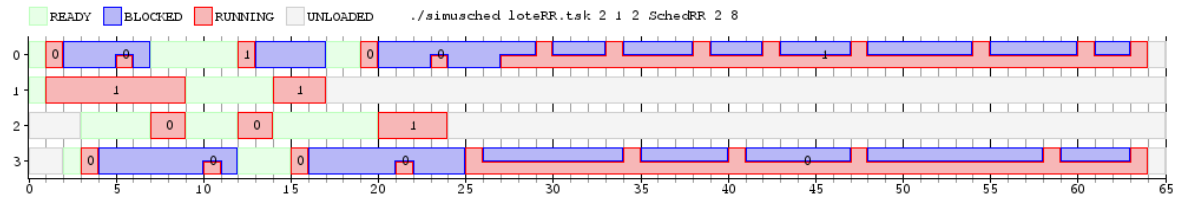


**Figura 9:** Diagrama para la ejecución del lote de tareas bajo 2 núcleos con quantumd de 2 y 8 ciclos respectivamente.

## 2.5. 3er caso:

En el último caso se observó el comportamiento para un lote de tarea. 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:

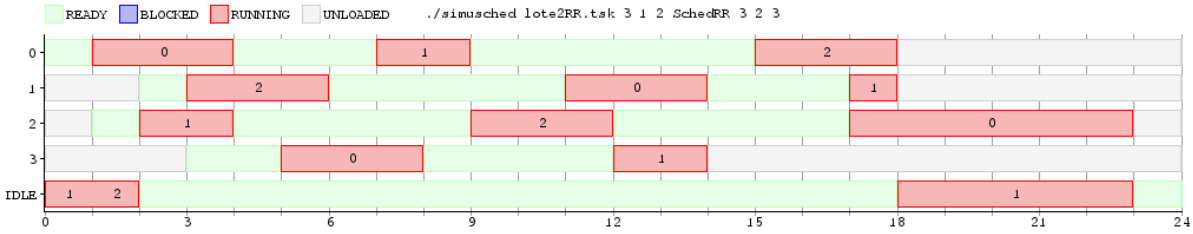


**Figura 10:** Diagrama de Gantt para la ejecución del lote de tareas bajo 3 núcleos con Scheduler Round Robin.

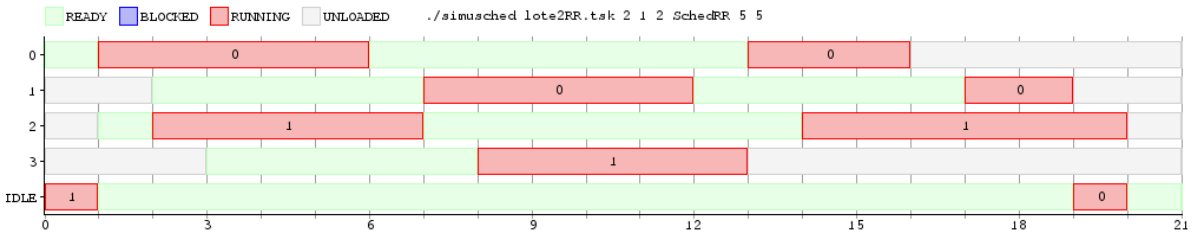


**Figura 11:** Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos con Scheduler Round Robin.

## 2.6. 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.*

1. *Responda:*

   a) *¿Qué problema están intentando resolver los autores?*

   The problem of multiprogram scheduling on a single processor is studied from the viewpoint of the characteristics peculiar to the program function that need guaranteed service. It is shown that an optimum fixed priority scheduler possesses an upper bound to processor utilization which may be as low as 70 percent for large task sets. It is also shown that full processor utilization can be achieved by dynamically assigning priorities on the basis of their current deadlines. A combination of these two scheduling techniques is also discussed.

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

   We turn now to study a dynamic scheduling algorithm which we call the deadline driven scheduling algorithm. Using this algorithm, priorities are assigned to tasks according to the deadlines of their current requests. A task will be assigned the highest priority if the deadline of its current request is the nearest, and will be assigned the lowest if the deadline of its current request is the furthest. At any instant, the task with the highest priority and yet fulfilled request will be executed. Such a method of assigning priorities to the tasks is a dynamic one, in contrast to a static assignment in which priorities of tasks do not change with time. We want now to establish a necessary and sufficient condition for the feasibility of the deadline driven scheduling algorithm.

   c) *Explicar coloquialmente el significado del teorema 7.*

   For a given set of m tasks, the deadline driven scheduling algorithm is feasible if and only if

   (C1/T1) + (C2/T2) + ... + (Cm/Tm) <=1:

2. *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.*

# 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).*

## 3.2. Ejercicio 7: Ejecución lote de tareas

*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.*

## 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 (dado que ahora en cada core tengo una tarea). Una vez alcanda las n distribuciones se las futuras tareas seran asignadas a los CPU's con menor cantidad de tareas running + blocked + ready. Es decir, se encolaran 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. Pág 6. Vamos a utilizar el mismo lote de tareas que se uso para este último en los casos 2 y 3. Manteniendo las mismas condiciones en las que fueron obtenidos los resultados de esta forma para el lotes de tarea:

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

Se obtuvo como resultado:

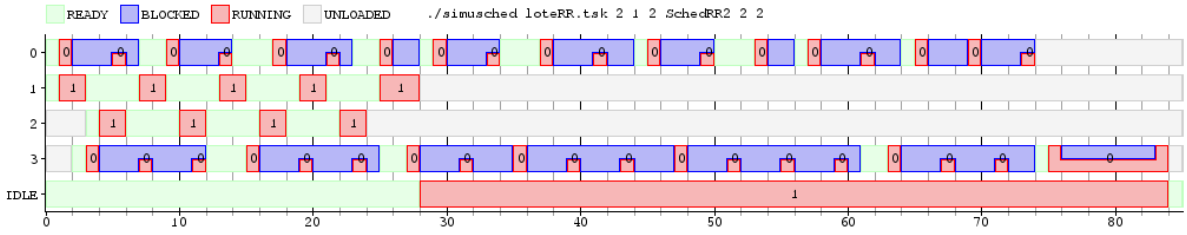


**Figura 12:** Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos con quantums de 2 y 2 ciclos respectivamente. En un Scheduler Round Robin sin migración de procesos.

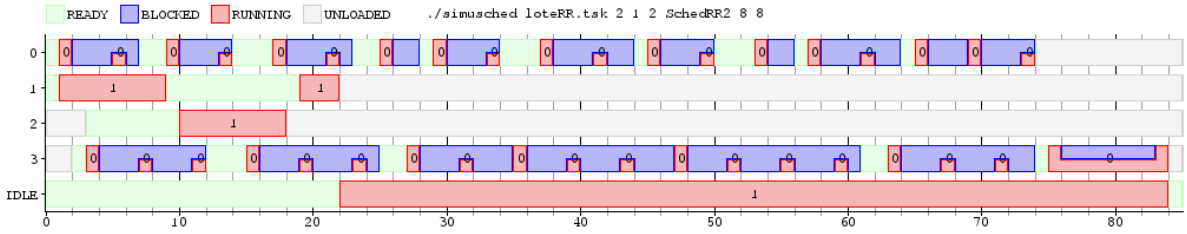


**Figura 13:** Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos con quantums de 8 y 8 ciclos respectivamente. En un Scheduler Round Robin sin migración de procesos.

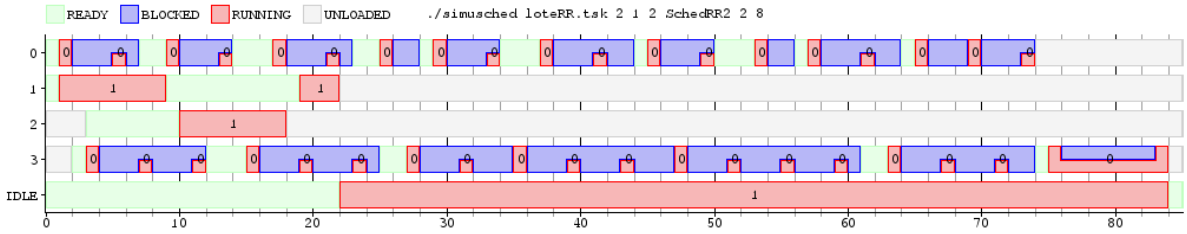


**Figura 14:** Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos con quantums de 2 y 8 ciclos respectivamente. En un Scheduler Round Robin sin migración de procesos.

Y para las tareas:

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

Obtuvimos los siguientes resultados:

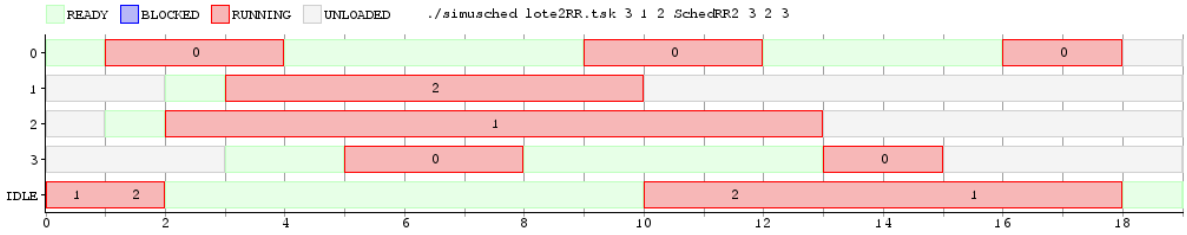


**Figura 15:** Diagrama de Gantt para la ejecución del lote de tareas bajo 3 núcleos con quantums de 3, 2 y 3 ciclos respectivamente. En un Scheduler Round Robin sin migración de procesos.

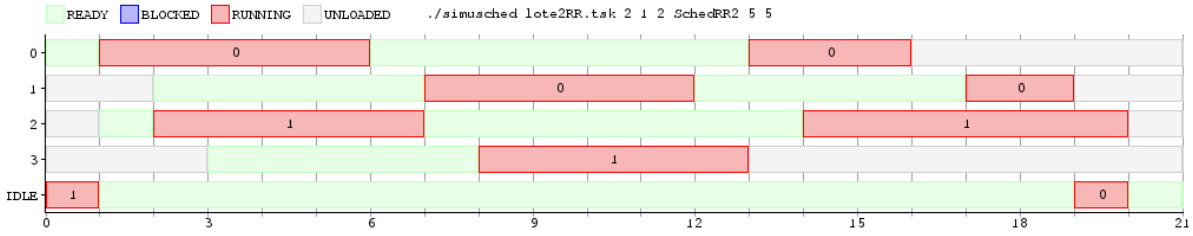


**Figura 17:**Diagrama de Gantt para la ejecución del lote de tareas bajo 2 núcleos con quantums de 5 y 5 ciclos respectivamente. En un Scheduler Round Robin sin migración de procesos.

## 3.4. 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.*

## 3.5. Ejercicio 10: Ejecución lote de tareas

*Diseñar un lote de tareas, cuyo scheduling sí sea factible con el algoritmo de prioridades fijas, donde se observe un mejor uso del CPU por parte del algoritmo de prioridades dinámicas.*