

# Sistemas Operativos

## TP1

14 de septiembre de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Federico Beuter	827/13	federicobeuter@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com

**Reservado para la cátedra**

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

# Índice

<b>1. Task Consola</b>	<b>3</b>
1.1. Código . . . . .	3
1.2. Diagrama GANTT . . . . .	3
<b>2. Rolando</b>	<b>4</b>
2.1. Diagrama GANTT . . . . .	4
<b>3. TaskBatch</b>	<b>6</b>
3.1. Código . . . . .	6
3.1.1. Diagrama GANTT . . . . .	6
<b>4. Round Robin Implementation</b>	<b>7</b>
4.1. Código . . . . .	7
4.1.1. Class Declaration . . . . .	7
4.1.2. Constructor . . . . .	7
4.1.3. Destructor . . . . .	7
4.1.4. Tick . . . . .	8
<b>5. Round-Robin Testing</b>	<b>9</b>
5.1. Diagramas GANTT . . . . .	9
<b>6. Comparacion Round-Robin/FCFS</b>	<b>11</b>
6.1. FCFS . . . . .	11
6.2. Round-Robin . . . . .	11
6.2.1. Diagramas GANTT . . . . .	11
<b>7. Mistery Scheduler</b>	<b>12</b>
7.1. Código . . . . .	12
7.1.1. Class Declaration . . . . .	12
7.1.2. Constructor . . . . .	12
7.1.3. Load . . . . .	12
7.1.4. Unblock . . . . .	13
7.1.5. Tick . . . . .	13
7.1.6. next_pid . . . . .	13

## 1. Task Consola

En este ejercicio programamos la tarea **Task Consola**, que lo que hace es simular una tarea interactiva que realiza  $n$  llamadas bloqueantes con una duración de ticks de reloj aleatoria entre **bmin** y **bmax**.

Una llamada se clasifica como **bloqueante** cuando el procesador no puede seguir ejecutando instrucciones hasta que algun tipo de recurso este disponible. Las llamadas bloqueantes en general son de entrada y salida. Este tipo de recursos se acceden comunmente mediante un **syscall**. El **syscall** en si debe ser ejecutado en primera instancia, lo que asumiremos que toma un ciclo de reloj. Luego el recurso toma un total de  $t$  ticks de reloj de forma aleatoria en responder. Por lo tanto, una tarea inactiva toma un ciclo de reloj para la llamada y permanecera bloqueada durante  $t$  ciclos adicionales.

### 1.1. Código

```
1 void TaskConsola(int pid, vector<int> params) {
2     srand(1); // set seed
3
4     for (int i = 0; i < params[0]; ++i) {
5         int t = params[1] + rand() % (params[2] - params[1] + 1);
6         uso_IO(pid, t);
7     }
8 }
```

Esta tarea toma un **pid** y un vector **params** de 3 parámetros. Dado que nuestros resultados dependerán de un generador de números pseudo-aleatorio, setteamos una semilla con el objetivo de poder replicar nuestros resultados.

Luego, el loop ejecuta la llamada **bloqueante** simulada con un request de entrada/salida utilizando como parámetro un numero aleatorio entre **bmin** y **bmax** que se encuentran en el vector en ese orden respectivamente.

Este numero no necesariamente tiene una distribución uniforme perfecta, pero se le parece demasiado dado que genera un valor entre 0 y  $RAND\_MAX^1$  (definido en la std) y luego le aplica la función modulo con un numero chico a un dominio grande. La función devuelve un numero  $t \in [0, RAND\_MAX]$ . La función modulo luego achica ese dominio a  $[0, bmax - bmin]$ . Finalmente, al sumarle **bmin** logramos que  $t \in [bmin, bmax]$ .

### 1.2. Diagrama GANTT

El siguiente diagrama fue generado con los siguientes parametros:

1. lote\_tsk: 1.tsk
2. num\_cores: 1
3. switch\_cost: 0
4. sched\_class: SchedFCFS

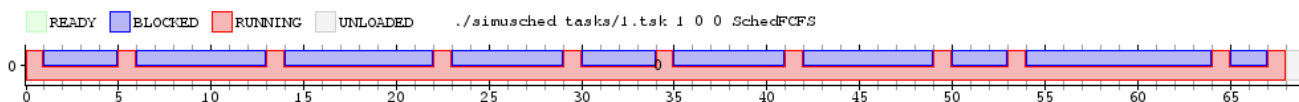


Figura 1: Task Consola

La Figura 1 nos muestra el diagrama de Gantt asociado a la gestión que desempeña el Scheduler FCFS bajo la ejecución del lote *lote1* de tareas. Este lote sólo contiene una única tarea de tipo TaskConsola con un *release time* nulo, y un vector  $\langle 10, 1, 10 \rangle$  como parametro. En consecuencia, y como a su vez el *switch cost* también es nulo, se ejecuta inmediatamente dicha tarea sin tener que mantenerse esperando en estado *ready*. Como el FCFS, cada vez que una tarea entra en estado *running*, no la interrumpe hasta que ésta finalice (y de cualquier forma es la única tarea corriendo), la actual puede concluir sin ningun tipo de desalojo de por medio. Con lo cual pasará a efectuar sus 10 llamadas bloqueantes, de un rango de entre 1 y 10 ciclos de reloj, consecutivamente y sólo volviendo al estado *running* durante un ciclo entre cada una para llamar a *uso\_IO()*, y finalmente una vez mas para llamar a *exit()* en el último ciclo.

<sup>1</sup>[http://www.cplusplus.com/reference/cstdlib/RAND\\_MAX](http://www.cplusplus.com/reference/cstdlib/RAND_MAX)

## 2. Rolando

### 2.1. Diagrama GANTT

El siguiente diagrama fue generado con los siguientes parametros:

1. lote\_tsk: 2.tsk
2. num\_cores: 1
3. switch\_cost: 4
4. sched\_class: SchedFCFS

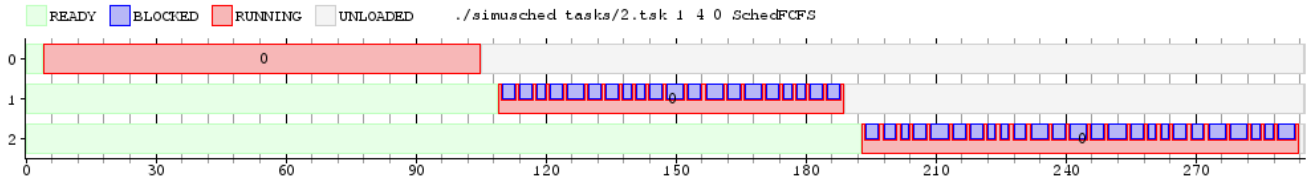


Figura 2: Rolando con 1 núcleo

En la figura 2 se presenta el diagrama de Gantt del *lote2* bajo el algortimo del scheduler FCFS. Este lote simula la situación de *Rolando*, para la que se requiere ejecutar 3 tareas. Las tres tendrán un *release time* nulo, de modo que cada tarea sólo deberá esperar al costo de cambio de contexto para entrar en estado *runing*. La primera, que es del tipo TaskCPU y con su parametro igual a 100, tras esperar 4 ciclos de reloj (por el *switch\_cost*), como su parametro indica, hace uso del CPU durante 100 ciclos de rejoy, y uno extra por la llamada a *exit()*. Tras finalizar la anterior, y después de otros 4 ciclos por el cambio de contexto, pasa al estado *runing* la segunda tarea, esta vez de tipo TaskConsola y con el vector  $\langle 20, 2, 4 \rangle$  como parametro. Esta efectua 20 llamadas bloqueantes, que demoran entre 2 a 4 ciclos, haciendo uso de 20 llamadas consecutivas a *uso\_IO()* que requieren a su vez de 1 ciclo cada una. Por último, luego de emplear su último ciclo asignado para llamar a *exit()*, y tras demorarse otros 4 por el cambio de contexto, pasa al estado *runing* la última tarea del lote. Nuevamente es del tipo TaskConsola, esta vez con el vector  $\langle 25, 2, 4 \rangle$  como parametro. De igual manera que en la tarea anterior se hace uso de *uso\_IO()* para efectuar las 25 llamadas bloqueantes, y de *exit()* en su último ciclo asignado.

1. lote\_tsk: 2.tsk
2. num\_cores: 2
3. switch\_cost: 4
4. sched\_class: SchedFCFS

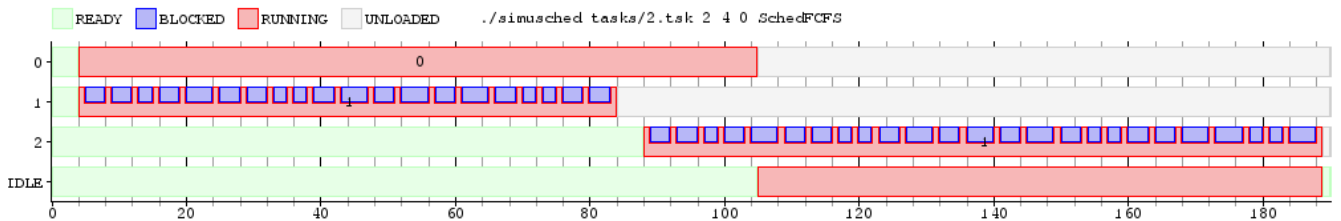


Figura 3: Rolando con 2 núcleos

En la Figura 3 se observa el diagrama de Gantt del mismo lote de tareas y scheduler de antes, pero esta vez bajo una CPU que cuenta con dos núcleos. Al igual que el caso anterior, el *switch\_cost* es de 4 ciclos, por lo que se demora esa cantidad de tiempo en empezar a correr las primeras tareas. A diferencia del caso anterior (en el que se contaba con un sólo núcleo) tras este tiempo, pasan al estado *runing* las primeras 2 tareas, TaskCPU y TaskConsola (de parametro

$\langle 20, 2, 4 \rangle$ ), corriendo paralelamente (una en cada núcleo). De esta forma, la tarea TaskConsola en actual ejecución puede efectuar sus llamadas bloqueantes y mientras, la tarea TaskCPU continuar trabajando ininterrumpidamente. Tras finalizar la tarea identificada por el *label* 1 del grafico (la TaskConsola 20 2 4), y el usual tiempo empleado para el cambio de contexto, el núcleo asignado a dicha tarea pasa a otorgarle tiempo en el mismo a la última tarea restante. Mientras esta aún se mantiene corriendo, la tarea TaskCPU continua en estado *runing*, y al finalizar pasa a asignarsele la tarea *IDLE* al núcleo previamente ocupado por TaskCPU; pues no le quedan mas tareas en el lote. La tarea TaskConsola (de parametro  $\langle 25, 2, 4 \rangle$ ) efectua sus 25 llamadas bloqueantes y termina como siempre (en el FCFS), con un *exit()* en su último ciclo asignado.

A continuación se muestra en una tabla las latencias de cada tarea del *lote2*. En el caso de las tareas de tipo Task-Consola, debido a que sus llamadas bloqueantes no duran un tiempo predeterminado, la latencia de la siguiente tarea es estimada entre un rango que resulta de asumir el mínimo y máximo tiempo que pueden durar bloqueadas tras cada llamada.

Cuadro 1: Latencias

Tarea	1 núcleo	2 núcleos
TaskCPU 100	4	4
TaskConsola 20 2 4	109	4
TaskConsola 25 2 4	174-214	69-109

### 3. TaskBatch

#### 3.1. Código

---

```
1 void TaskBatch(int pid, vector<int> params) {
2
3     srand(1); // set seed
4
5     int total_cpu = params[0];
6     int cant_bloqueos = params[1];
7
8     bool config[total_cpu] = {false};
9     for (int i = 0; i < cant_bloqueos; i++) config[i] = true;
10    random_shuffle(&config[0], &config[total_cpu - 1]);
11
12    for (int i = 0; i < total_cpu; ++i) {
13        uso_CPU(pid, 1); // syscall or normal usage
14        if (config[i] == true) { // block
15            uso_IO(pid, 1);
16            cant_bloqueos--;
17        }
18    }
19 }
```

---

##### 3.1.1. Diagrama GANTT

El siguiente diagrama fue generado con los siguientes parametros:

1. lote\_tsk: 3.tsk
2. num\_cores: 1
3. switch\_cost: 0
4. sched\_class: SchedFCFS
5. n: 10
6. bmin: 1
7. bmax: 10

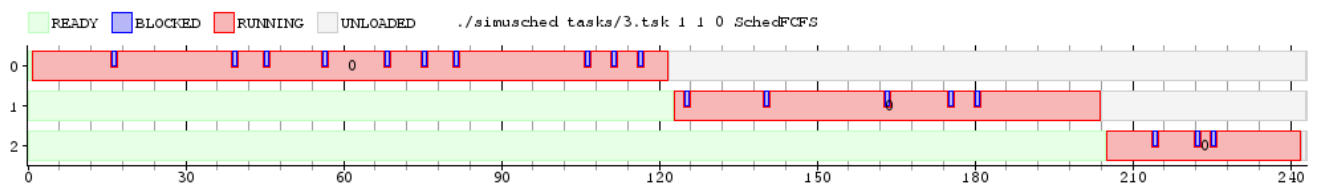


Figura 4: Task Batch

**TODO: Explicar bien el diagrama!**

## 4. Round Robin Implementation

### 4.1. Código

En primer lugar, modificamos la declaracion de la clase `SchedRR` agregando una serie de atributos privados:

#### 4.1.1. Class Declaration

---

```
1 class SchedRR : public SchedBase {
2     public:
3         SchedRR(std::vector<int> argn);
4         ~SchedRR();
5         virtual void load(int pid);
6         virtual void unblock(int pid);
7         virtual int tick(int cpu, const enum Motivo m);
8
9     private:
10        int quantum;
11        int* cycles;
12        std::queue<int> q;
13 };
```

---

Por un lado, el `quantum`, que nos dice por cuantos ciclos de clock nuestras tareas se van a ejecutar. Por otro lado, tenemos un puntero de enteros llamado `cycles`, que lo que hace es mantener la cuenta de cuantos ciclos le quedan a cada tarea en ejecución por CPU antes de llegar al quantum. Finalmente, tenemos una cola de tareas `q` compartida entre CPUs.

#### 4.1.2. Constructor

---

```
1 SchedRR::SchedRR(vector<int> argn) {
2     // Round robin recibe la cantidad de cores y sus cpu-quantum por parametro
3
4     quantum = argn[1];
5     cycles = new int[argn[0]];
6     fill_n(cycles, argn[0], quantum);
7 }
```

---

#### 4.1.3. Destructor

---

```
1 SchedRR::~~SchedRR() {
2     delete [] cycles;
3 }
```

---

#### 4.1.4. Tick

---

```
1 int SchedRR::tick(int cpu, const enum Motivo m) {
2     if (m == EXIT) {
3         // Si el pid actual termino, sigue el proximo.
4         if (q.empty()) return IDLE_TASK;
5         else {
6             int sig = q.front(); q.pop();
7             cycles[cpu] = quantum;
8             return sig;
9         }
10    } else {
11        if (current_pid(cpu) == IDLE_TASK && !q.empty()) {
12            int sig = q.front(); q.pop();
13            cycles[cpu] = quantum;
14            return sig;
15        } else {
16            cycles[cpu]--;
17
18            if (cycles[cpu] == 0) {
19                if (q.empty()) {
20                    cycles[cpu] = quantum;
21                    return current_pid(cpu);
22                } else {
23                    int sig = q.front(); q.pop();
24                    q.push(current_pid(cpu)); // re-add to queue
25                    cycles[cpu] = quantum;
26                    return sig;
27                }
28            } else {
29                return current_pid(cpu);
30            }
31        }
32    }
33 }
```

---



## 5. Round-Robin Testing

### 5.1. Diagramas GANTT

1. lote\_tsk: 5.tsk
2. num\_cores: 1
3. switch\_cost: 2
4. sched\_class: SchedRR

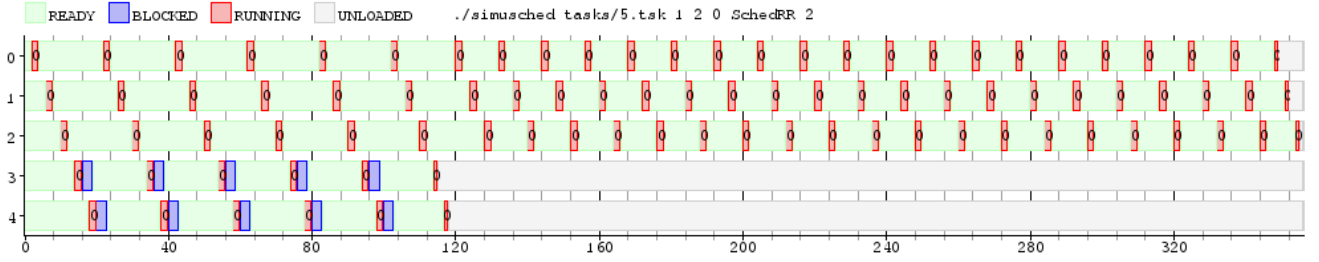


Figura 5: Round-Robin (quantum=2)

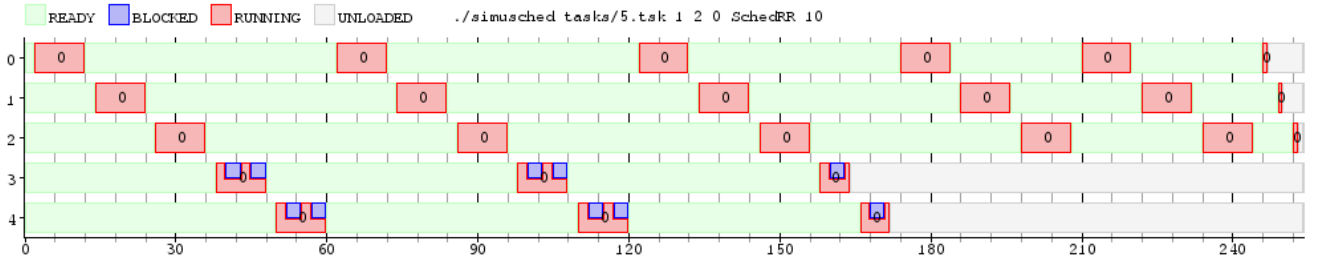


Figura 6: Round-Robin (quantum=10)

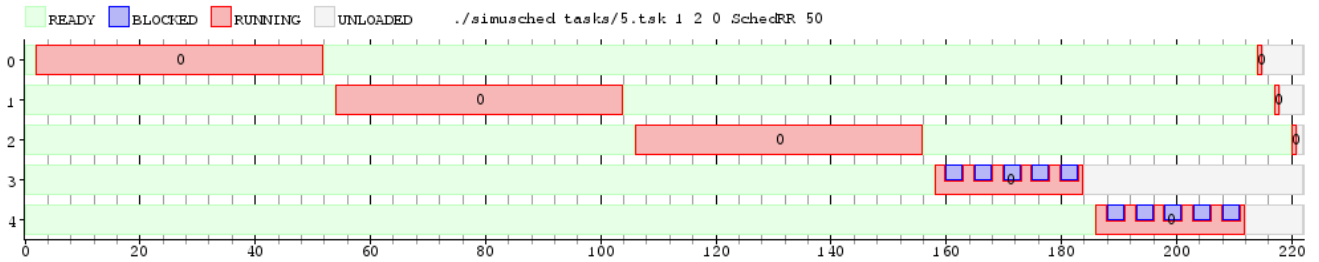


Figura 7: Round-Robin (quantum=50)

En las 3 figuras anteriores podemos observar los diagramas de Gantt para el algoritmo del scheduler Round-Robin bajo el cual corre un mismo lote (el *lote5*). En los 3 casos se trata de una CPU de un único núcleo, en el cual se tendrá un *switch\_cost* de 2 ciclos de reloj. La particular diferencia entre los 3 es el *quantum* pasado como parametro al Round-Robin; en el que su valor es de 2, 10, 50 respectivamente (según el orden de aparición). Este *quantum* es el que determina cuantos ciclos le son lícitos a cada tarea permanecer en estado *running*. Ejecutando las tareas de manera alternada por el orden que rige una cola de tareas, cada vez que la CPU se encuentra disponible, se desencola la siguiente tarea y se vuelve a encolar aquella sin finalizar que haya agotado el *quantum* de tiempo asignado. Para un *quantum* no muy grande y tareas que requieran un tiempo no muy corto (menor que el *quantum*), como es el escenario

que figura en los 3 diagramas anteriores, el Round-Robin reduce las latencias, ya que al turnar de manera más *justa* el volumen de tareas, la última no debe lidiar con la espera del tiempo completo de ejecución de sus predecesoras, como ocurre con el FCFS. No obstante, se distribuye un costo extra de intercambio de contexto, que repercute en un aumento del tiempo total de ejecución del lote, de manera acorde a la frecuencia de conmutación dada por el *quantum*. Experimentalmente esto se hace evidente en los graficos anteriores, en donde en el primer caso, de menor *quantum*, el lote abarca un tiempo significativamente mayor que el de los otros dos casos. A su vez el tercer caso, bajo un *quantum* = 50, mejora levemente su performance contra su predecesor, de *quantum* = 10. Sin embargo, no es buena idea tener un *quantum* demasiado grande, pues la rutina tendería a degenerarse hacia el FCFS.

## 6. Comparacion Round-Robin/FCFS

### 6.1. FCFS

El FCFS "First Come, First Served", como su nombre sugiere es un scheduler con una política de estilo FIFO, es decir que la primera tarea en la cola de tareas será la primera en pasar al estado *running* y la primera en finalizar (en el núcleo en el que fue asignada). Mas específicamente, dado como parametro el *switch\_cost* y un número *n* de cores, asigna las primeras *n* tareas en estar en estado *ready* a los *n* núcleos disponibles (asignando la tarea *IDLE* para aquellos para los cuales no haya alguna tarea lista), y permitiéndoles permanecer en *running* sin conmutarlas con otras hasta que estas primeras finalicen; de modo que lidiada con cada tarea una sólo vez, y sin ningun tipo de desalojo intermedio.

### 6.2. Round-Robin

El Round-Robin es un scheduler que consiste en asignarle un tiempo predeterminado llamado *quantum* (recibido como parametro) a cada tarea de la cola de tareas para permanecer en estado *running*; finalizado este tiempo, si hay otras tareas en estado *ready* esperando a ser ejecutadas, la tarea actual es desalojada y conmutada con la siguiente (en caso de no haber otra en la cola, sigue corriendo la actual). La tarea recién desalojada pasa a tomar el último lugar en la cola (si es que no había finalizado), teniendo que esperar a que el resto previamente encolado consuma el mismo *quantum* de tiempo cada una. En caso de tratarse de una CPU con varios núcleos, la cola de tareas es desencolada a medida que se vaya desocupando algún núcleo (sin tener en cuenta para esto a la tarea *IDLE*).

#### 6.2.1. Diagramas GANTT

El siguiente diagrama fue generado con los siguientes parametros:

1. lote\_tsk: 5.tsk
2. num\_cores: 1
3. switch\_cost: 2
4. sched\_class: SchedFCFS

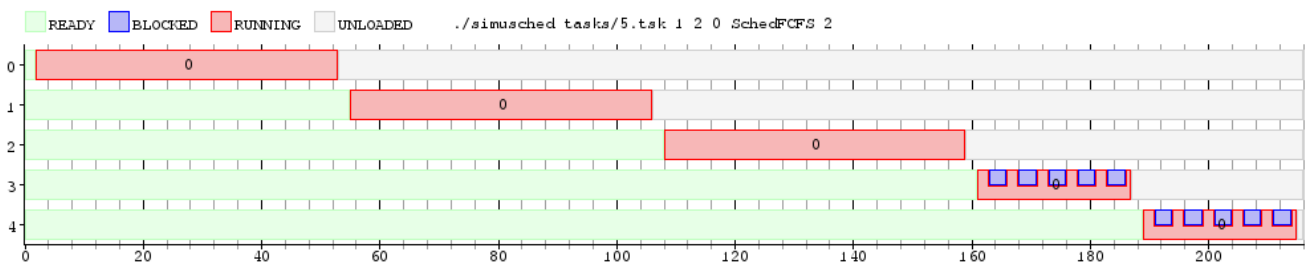


Figura 8: FCFS

Ya hemos experimentado con este lote bajo el algoritmo de Round-Robin, y con diferentes *quantums* durante el ejercicio anterior. Ahora en esta oportunidad, lo hacemos bajo el algoritmo del scheduler FCFS, y con el mismo lote. La idea es exhibir las diferencias ya advertidas en esta sección entre ambos schedulers. Para empezar, como en el caso anterior, se requieren de 2 ciclos de reloj para cada intercambio de contexto; dándole este tiempo de espera a la primer tarea, de tipo TaskCPU. La diferencia mas notable con la rutina anterior, es que cada tarea no es interrumpida sino hasta que la misma finalice; esto para ciertos lotes de tareas podría traer una penalización severa de performance, pues mientras una tarea realiza una llamada bloqueante, la CPU permanece ociosa hasta que la misma termine, en especial en aquellos casos en que el bloqueo se prolongue por mucho tiempo. En este caso, la mayoría de las tareas del lote son del tipo TaskCPU, que hacen pleno uso del CPU, por lo que el tiempo perdido en los bloqueos cortos de las otras dos tareas se compensan en ahorros que demora conmutar frecuentemente las tareas (lo que ocurre en el Round-Robin), dando una performance parecida o hasta mejor que con el algoritmo de Round-Robin.

## 7. Mistery Scheduler

A partir del .o de un scheduler, hicimos ingeniería inversa de su comportamiento a partir de la ejecución de varias instancias.

### 7.1. Código

#### 7.1.1. Class Declaration

---

```
1 class SchedNoMystery : public SchedBase {
2     public:
3         SchedNoMystery(std::vector<int> argn);
4         virtual void load(int pid);
5         virtual void unblock(int pid);
6         virtual int tick(int cpu, const enum Motivo m);
7     private:
8         std::vector<int> quantum_list;
9         std::vector<int> quantum_curr;
10        std::list<int> q;
11        int cycles_left, not_executed, unblocks_left;
12        int next_pid(void);
13 };
```

---

#### 7.1.2. Constructor

---

```
1 SchedNoMystery::SchedNoMystery(vector<int> argn) {
2     // cpu cores, rest of params
3     cycles_left = 1;
4     not_executed = 0;
5
6     quantum_list.push_back(1);
7
8     if (argn.size() > 1) {
9         for (vector<int>::iterator it = ++argn.begin(); it != argn.end(); ++it) {
10             quantum_list.push_back(*it);
11         }
12     }
13 }
```

---

#### 7.1.3. Load

---

```
1 void SchedNoMystery::load(int pid) {
2     list<int>::iterator it = q.begin();
3     advance(it, not_executed);
4     q.insert(it, 1, pid);
5     not_executed++;
6
7     quantum_curr.push_back(0);
8 }
```

---

#### 7.1.4. Unblock

---

```
1 void SchedNoMistry::unblock(int pid) {
2     list<int>::iterator it = q.begin();
3     advance(it, unblocks_left);
4     q.insert(it, 1, pid);
5     unblocks_left++;
6 }
```

---

#### 7.1.5. Tick

---

```
1 int SchedNoMistry::tick(int cpu, const enum Motivo m) {
2     if (m == EXIT || m == BLOCK) {
3         // current pid ended, get next
4         if (q.empty()) return IDLE_TASK;
5         else { // get task from list
6             return next_pid();
7         }
8     } else {
9         if (current_pid(cpu) == IDLE_TASK && !q.empty()) {
10             return next_pid();
11         } else {
12             cycles_left--;
13
14             if (current_pid(cpu) != IDLE_TASK && cycles_left == 0) {
15                 quantum_curr[current_pid(cpu)] = min(quantum_curr.at(current_pid(cpu))
16                                                         +1,
17                                                         ((int)quantum_list.size()-1));
18                 if (q.empty()) {
19                     cycles_left = quantum_list.at(quantum_curr.at(current_pid(cpu)));
20                     not_executed = max(not_executed - 1, 0);
21                     return current_pid(cpu);
22                 } else {
23                     q.push_back(current_pid(cpu));
24                     return next_pid();
25                 }
26             } else {
27                 return current_pid(cpu);
28             }
29         }
30 }
```

---

#### 7.1.6. next\_pid

---

```
1 int SchedNoMistry::next_pid() {
2     int pid = q.front(); q.pop_front();
3     if (unblocks_left > 0) {
4         cycles_left = 1;
5         unblocks_left--;
6     } else {
7         cycles_left = quantum_list.at(quantum_curr.at(pid));
8     }
9     not_executed = max(not_executed - 1, 0);
10
11     return pid;
12 }
```

---