+

Ciencias de la Computación

Sistemas Operativos

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

TP Scheduling

Integrante	LU	Correo electrónico
Fabian González	076/01	utyman@gmail.com
Maximiliano Rey	037/13	rey.maximiliano@gmail.com
Diego Sueiro	075/90	dsueiro@gmail.com

1 Introducción

El siguiente trabajo tiene por objetivo evaluar distintas estrategias de scheduling de tareas. Se presentarán los detalles de implementación de los diferentes algoritmos requeridos, se mostrarán los resultados frente a diferentes escenarios y se realizará una comparación de las diferentes estrategias desarrolladas. Se implementarán también diferentes tipos de tarea (de consola, procesos batch) para poder realizar esta evaluación.

2 FCFS

El primer algoritmo evaluado fue FCFS (*First Came First Save*). Se trata de la estrategia de scheduling más sencilla. Teóricamente, está representado por una cola que concede el recurso a cada tarea en un orden estricto de llegada. Cuando cada uno de los procesos pasa a estado READY, se agrega a una cola global. A medida que el proceso que se encuentra ejecutando termina, se selecciona al proceso que más tiempo ha estado en esta cola.

Este algoritmo ya estaba implementado por la cátedra y se encuentra en el archivo `sched_fcfs.cpp`. Para evaluarlo, el **ejercicio 1** solicitaba programar una tarea TaskConsola, que simulaba un proceso interactivo. La tarea recibe como parámetro:

1. *n*: que indica la cantidad de llamadas bloqueantes de la tarea.
2. *bmin*: la cota inferior de duración de cada llamada bloqueante
3. *bmax*: la cota superior de duración de cada llamada bloqueante.

Se encuentra implementada como la función TaskConsola en el archivo `tasks.cpp`.

La función debía realizar llamadas bloqueantes de duración aleatoria. Para simular ese comportamiento, se usó la función *rand* que devuelve un número pseudoaleatorio entre 0 y `MAX_RAND`[1]. Para que la duración entrara en un rango, le sacamos el módulo de a la diferencia entre *bmax* y *bmin* y le sumamos el valor *bmin*:

```
unsigned int bmax = params[2]; // la duracion maxima de la llamada bloqueante
unsigned int bmin = params[1];  // la duracion minima de la llamada bloqueante
unsigned int n = params[0]; // la cnatidad de llamadas bloqueantes
for (unsigned int i = 0; i < n; i++) { // realizo n llamadas bloqueantes
    int duracion = rand()%(bmax - bmin) + bmin; // número aleatorio
     //entre dos números
    uso_IO(pid, duracion); // realizo las llamadas bloqueantes
}
```

El **ejercicio 2** solicitaba realizar pruebas con lotes de tareas sobre el scheduler FCFS. Se debía realizar un lote de tres tareas usando el algoritmo FCFS para 1, 2 y 3 núcleos:

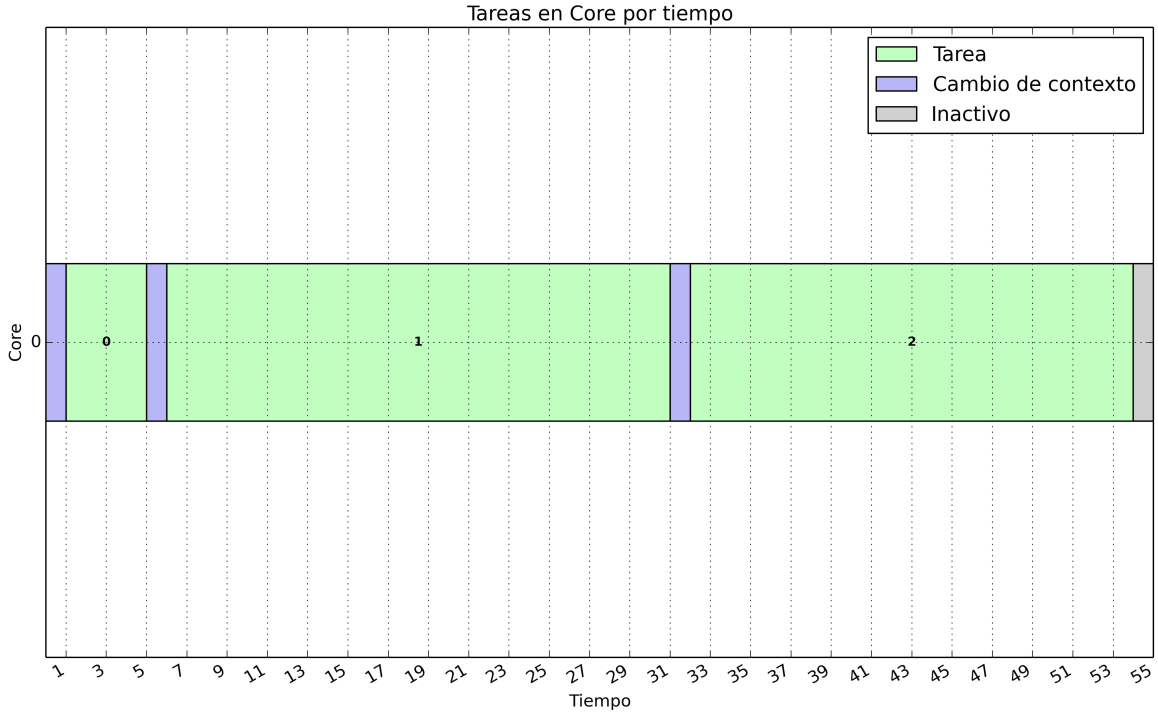
```
TaskCPU 3
TaskConsola 3 5 10
TaskConsola 3 5 10
```

Todas las tareas entran en estado READY en el instante 3 y las tares de consola realizan 3 llamadas bloqueantes de 5 a 10 ticks de reloj de duración.

Los resultados fueron los siguientes:

[1] Se trata de un macro que depende de la implementación de la librería estándar de C. Se garantiza que es al menos 32767

Figura 1: 1 TaskCPU y 2 TaskConsole con 1 core



++—-

Figura 2: 1 TaskCPU y 2 TaskConsole con 2 cores

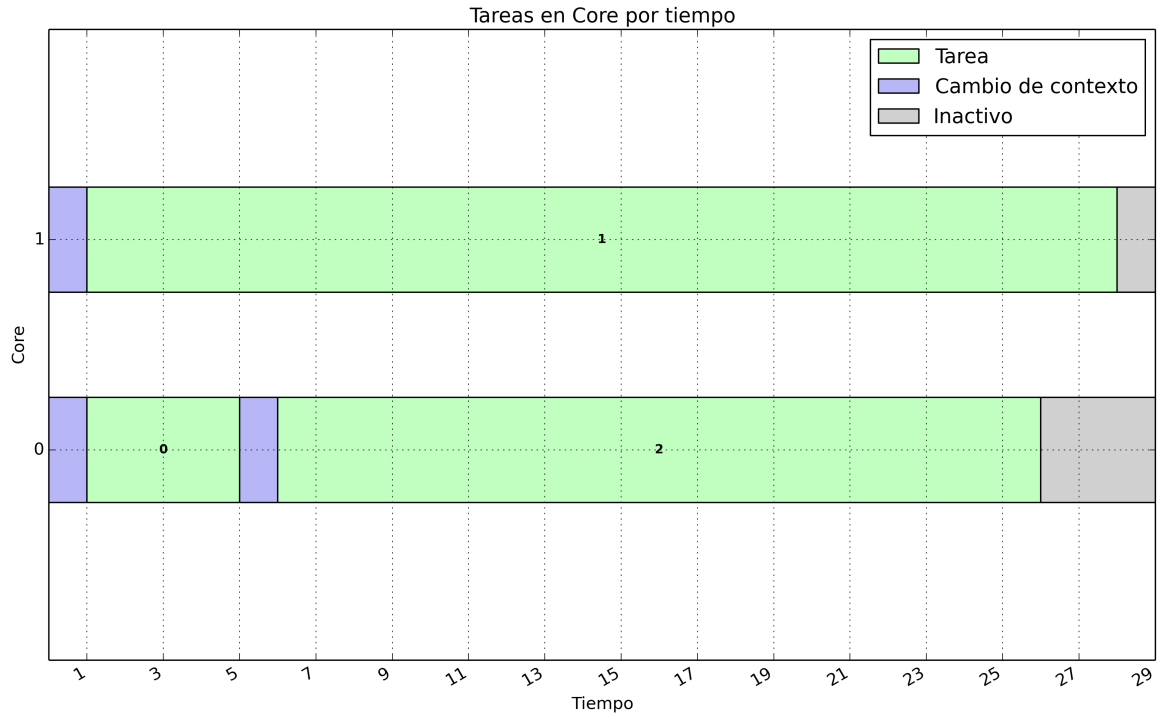
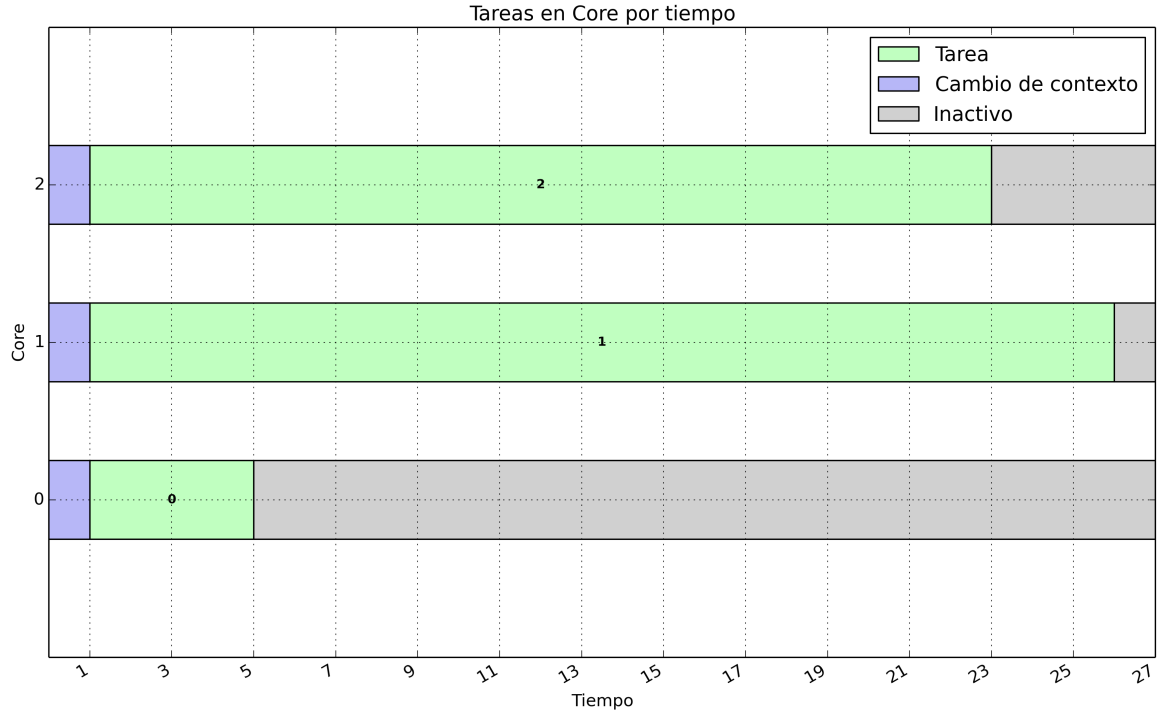


Figura 3: 1 TaskCPU y 2 TaskConsole con 3 cores



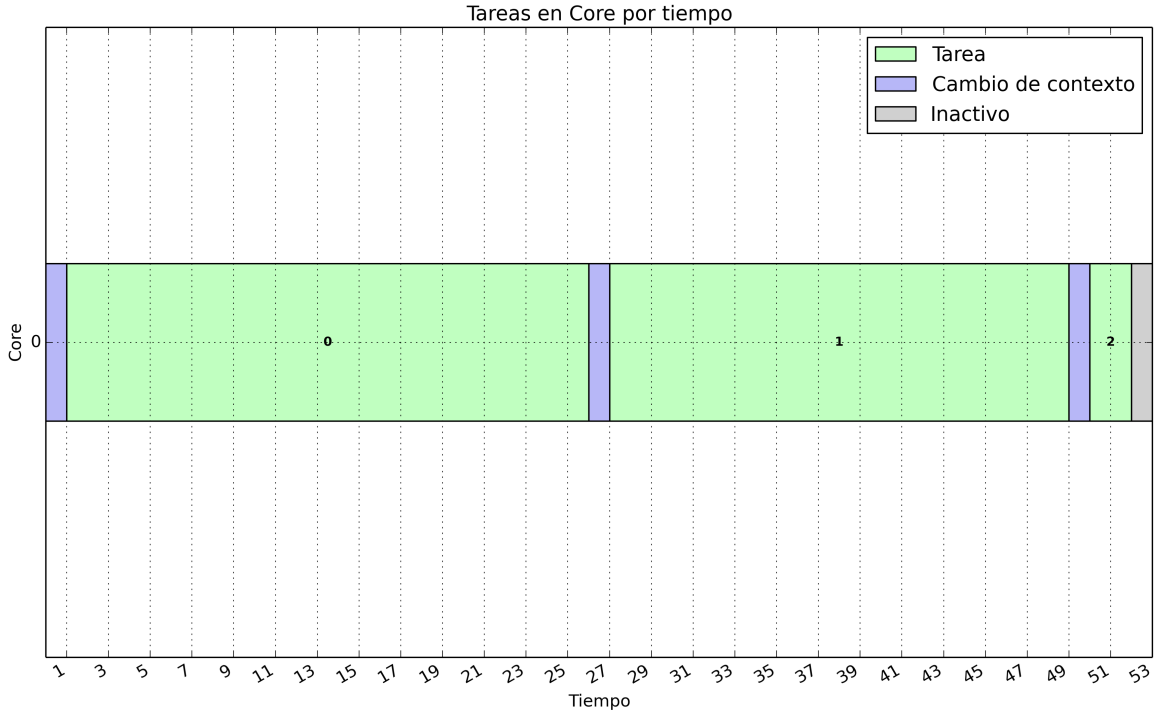
Se observa que no hay desalojo en esta estrategia de scheduling. De manera que cada vez que a una tarea se le concede el recurso CPU no lo libera hasta que termina su ejecución independientemente de los bloqueos que realice. Esto se implementó en el código no realizando ninguna lógica en la función que maneja el evento block.

```
void SchedFCFS::unblock(int pid) {
    // Uy! unblock!... bueno, ya seguir'a en el próximo tick
}
```

Observemos qué sucede si las primera tareas que toman el procesador es la que que tarda más en terminar. Usemos el siguiente lote, ejecutado en un core :

```
TaskConsola 3 5 10
TaskConsola 3 5 10
TaskCPU 1
```

Figura 4: 1 TaskCPU de poca duración y 2 TaskConsole con 1 core



Como señalan varios autores [1, p. 414] [2, p. 134], este tipo de estrategias de scheduling funciona mejor para procesos cortos que para procesos de larga duración. Observamos en el lote anterior, que la tarea 3 tuvo un *waiting time* de 49 ticks de reloj. Esto se atenúa en los lotes en los que se puede enviar la tarea a otro procesador. FCSF no parece muy atractivo para el modelo de único core.

3 Round Robin

Con esta nueva estrategia, podemos enfrentar el problema que tiene FCFS con las tareas de corta duración. SchedRR es el primer algoritmo que presentamos en el trabajo que usa desalojo. Cada vez que se produce un click de reloj, el algoritmo decide cuál es la tarea a la que se le asigna el procesador.

En la primera de las estrategias *round robin*, de acuerdo a lo solicitado en el **ejercicio 3**, se usa una única cola global. Cuando una tarea está lista para ejecutarse pasa a esta cola, a la que llamaremos cola de ready.

Periódicamente, cuando se produce la interrupción de reloj, se verifica si finalizó el quantum del núcleo correspondiente. Si es así, la estrategia toma el primer elemento de la cola, lo saca de ella y le asigna el procesador. En caso de que la cola se encuentre vacía, se ejecuta la tarea IDLE. La tarea que se encontraba ejecutando, si todavía no terminó, se vuelve a encolar.

Cuando una tarea se bloquea por pedido de E/S, sale de la cola de READY. Recién vuelve a encolarse cuando se desbloquea.

Esta estrategia se encuentra implementada en los archivos `sched_rr.h` y `sched_rr.cpp`. La cola se define en el header como un atributo privado:

```
std::queue<int> q;
```

Tenemos atributos que nos sirven para manejar los quantums correspondientes a cada núcleo:

```
        std::vector<int> contadorQuantums; // se usa para controlar los quantums
       std::vector<int> contadorQuantumsOriginal; // guardo la cantidad de
               // quantums de cada nucleo
```

contadorQuantums lo utilizaremos para saber en cada tick de reloj si se terminó el quantum, mientras que contadorQuantumsOriginal es un vector que guarda para cada núcleo la cantidad de ticks (interrupciones de reloj) que abarca el quantum.

El comportamiento frente al tick de reloj se define en la función *tick*.

En caso de que el proceso desalojado termine, se toma el próximo en la cola. Si la cola está vacía, se retorna la `TASK_IDLE`:

Esta función se ocupa también de volver el contador del quantum del núcleo correspondiente al valor original en caso de que el proceso salga o se bloquee, mientras que si el proceso se encuentra todavía ejecutándose se fija si se agotó el quantum.

```
    if (m == EXIT) {
        // Si el pid actual terminó, sigue el proximo encolado
        if (q.empty()) return IDLE_TASK; // si la cola esta vacia, se retorna IDLE_TAS
        else {
            int sig = q.front(); q.pop(); // sino se toma el primero y se desencola
            return sig;
         }
    }
```

En el caso de que el proceso desalojado se bloquee por E/S, se sigue el mismo comportamiento:

```
    if (m == BLOCK) {
         if (!q.empty()) { // en caso de bloqueo se usa la misma estrategia
            int sig = q.front(); q.pop();
            return sig;
         } else {
             return IDLE_TASK; // si el único proceso está bloquedo
                               // ejecuta IDLE_TASK
         }
    }
```

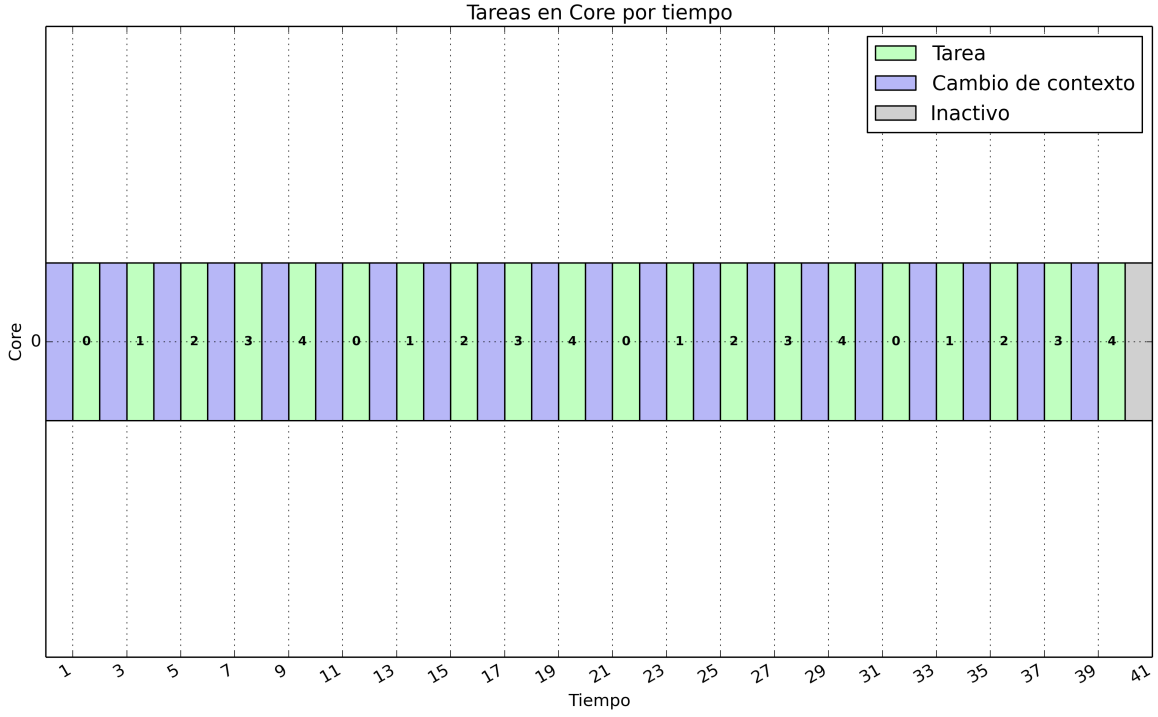
Si la tarea no terminó, se realiza lo mismo pero se vuelve a encolar la tarea desalojada:

```
    if (m == TICK) {
        if (!q.empty()) { // si se produjo una interrupcion de reloj se hace lo mismo
                  // pero se vuelve a encolar la tarea
            int sig = q.front(); q.pop();
            int des = current_pid(cpu);
            if (des != IDLE_TASK) {
                q.push(des); // vuelvo a encolar el proceso desalojado
            }
            return sig;
         } else {
               return current_pid(cpu);
         }
    }
```

De acuerdo a lo pedido en el **ejercicio 4**, presentamos varios lotes de tareas, observándo como el comportamiento se corresponde a la estrategia *round robin*:

```
TaskCPU 3
TaskCPU 3
TaskCPU 3
TaskCPU 3
TaskCPU 3
```

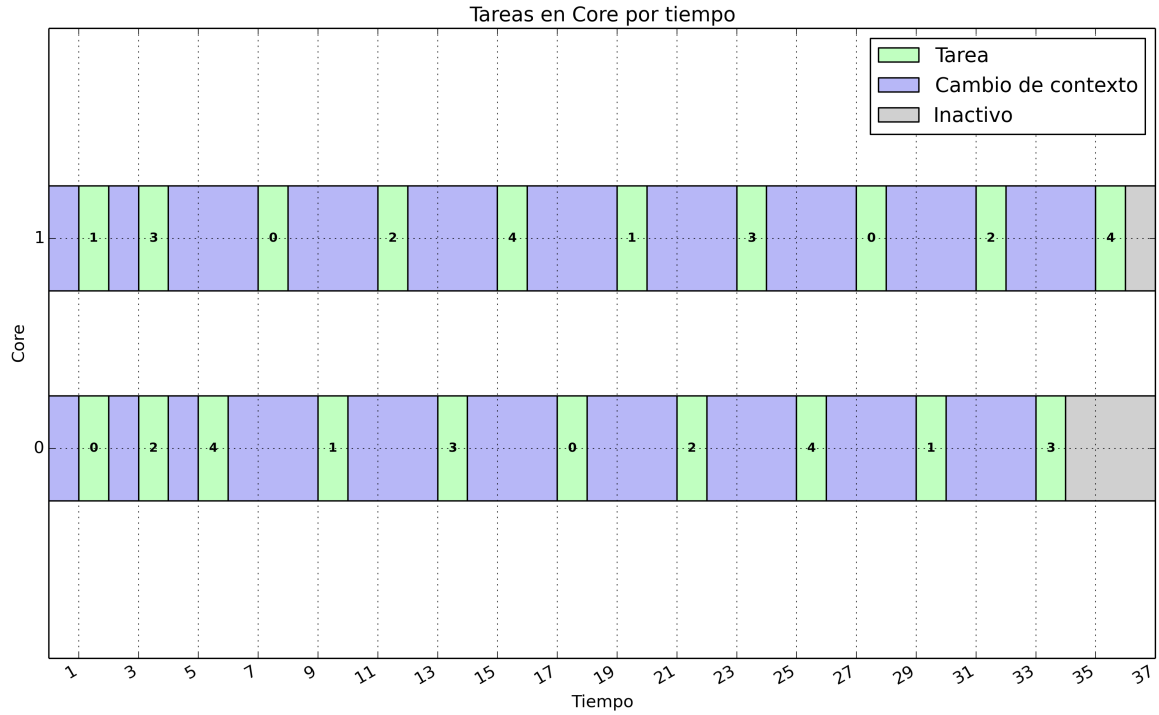
Figura 5: 5 TaskCPU sin llamadas de E/S, 1 core



En este lote, se observa cómo actúa round-robin en un ambiente de un solo core. Las tareas se van alternando en el uso del procesador, con el tiempo necesario para el cambio de contexto. Cada vez que una tarea es desalojada, se vuelve a encolar hasta terminar su procesamiento. En este caso en particular, no se produjo ninguna llamada a E§: cuando la tarea es desalojada vuelve en el mismo acto a la cola de READY. Vemos, entonces que la secuencia de ejecución fue: `1-2-3-4-1-2-3-4...`

Ejecutamos el mismo lote pero con dos cores:

Figura 6: 5 TaskCPU sin llamadas de E/S, 2 cores



En este caso, vemos como al tener una única cola global, se produce el pasaje de una tarea de un core a otro.

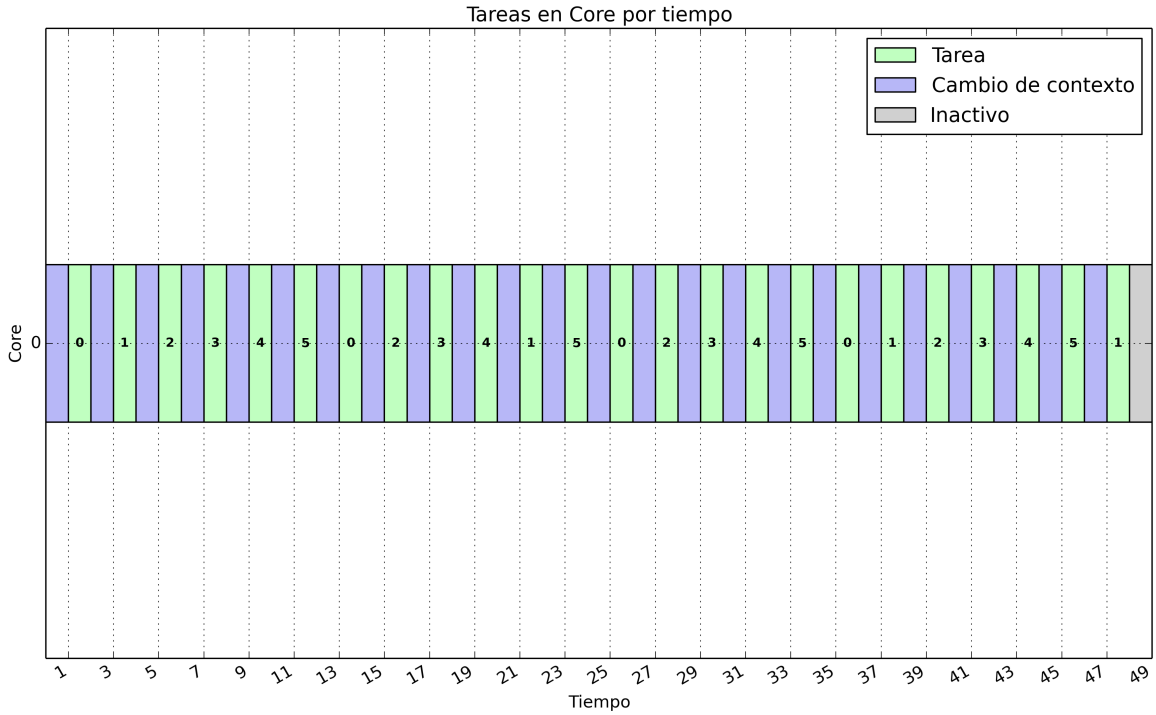
Se debe notar que hasta al tick 4 de reloj, entre el quantum de cada tarea, nada más tenemos el overhead del cambio de contexto. Sin embargo, al pasar las tareas a los otros cores, el overhead entre tarea aumenta ya que le pedimos que el cambio de core tome 2 ticks de reloj. Esta primera versión de *round robin* no toma en cuenta la penalidad de cambio de núcleo, por lo que cuando se toma una tarea de la cola no se toma en cuenta si estuvo ejecutando en el otro core.

De todos modos, vemos como se continúa globalmente la idea de una cola circular. La selección de la tarea asignada al core del tick sigue teniendo la secuencia `1-2-3-4-1-2-3-4...`

Por último, tomamos un lote con una tarea que llama a E/S y se bloquea:

```
TaskCPU 3
TaskConsola 3 5 10
TaskCPU 3
TaskCPU 3
TaskCPU 3
TaskCPU 3
```

Figura 7: 5 TaskCPU y un TASKCONSOLE que llama a E/S, 1 core



Observemos que en este caso la tarea 1 se bloquea y no veulve a la cola hasta tick 22.

4 Scheduling para tiempo real

Al analizar los algoritmos de scheduling, Tanembaum [2, p. 136] señala que es relevante distinguir tres ambientes diferentes:

- Batch
- Interactivo
- Tiempo Real

En los ambientes batch, no existe la exigencia planteada por los usuarios frente a algún tipo de periférico o terminal. Como no se necesita una respuesta interactiva rápida, son aceptables algoritmos sin desalojo o bien algoritmos con desalojo que tienen un quantum largo. Se requiere un uso intensivo y largo del CPU. Conviene entonces minimizar el overhead del cambio de contexto.

En el caso de los ambientes interactivos, lo que se intenta es que las tareas no acaparen el recurso CPU e impidan que otras tareas accedan a él. Por ello, conviene plantear una estrategia de desalojo.

El tercer ambiente es el que concierne al paper de Liu y Leilan [3]: tiempo real. Se trata de ambientes en los que el control del tiempo de ejecución de cada proceso es crítico. En estos ambientes **es necesario que la ejecución de cada tarea termine antes de que pase un período determinado de tiempo (deadline)**[2]. En caso de que no se cumpla esta restricción, se producirá un *overflow*. Esto es lo que distingue a un ambiente *hard real time* de un *soft real time*, en donde basta con cumplir estadísticamente el control de tiempo de ejecución.

Dadas las condiciones del ambiente, se hace necesario asignar algún tipo de prioridad a las tareas. El paper presenta dos algoritmos: uno de *prioridad fija*, que asigna a cada tarea su prioridad cuando son cargadas en el sistema, y uno de *prioridad dinámica* que asigna prioridades dinámicamente en el momento en que el scheduler tiene que decidir qué tarea va a ejecutar (cuando se termina el quantum o bien cuando el procesador queda libre por bloqueo o por terminarse la tarea que estaba ejecutándose). En el primer caso, las prioridades no cambian a lo largo de tiempo: se trata de un sistema de asignación de prioridades estática, mientras que en el segundo caso se trata de un sistema dinámico en cuanto a la asignación de prioridades.

En el paper de Liu y Layland se explicitan una serie de conceptos relevantes para el problema:

- T, el *request period* es el tiempo que pasa entre una ejecución de la tarea y la próxima llamada a la tarea. Se supone que el período comprendido entre el momento en que la tarea entra al sistema y el momento en que sale, tiene que ser menor al *request period* para evitar overflow.
- C, el tiempo de ejecución o *runtime*, el tiempo total en que la tarea realizará operaciones en el sistema.
- el *request rate*, la inversa del *request period*.

El algoritmo de asignación fija determina cuál es la prioridad de la tarea en el momento de la carga, asignándole importancia a cada proceso de acuerdo a la inversa de su request period, es decir a su *request rate*.

Este algoritmo se encuentra implementado en los archivos `sched_fixed.h` y `sched_fixed.cpp`.

En `sched_fixed.h` definimos la clase TaskComparable:

```
class TaskComparable
{
  public:
     TaskComparable() {};                                    //constructor default
    TaskComparable(int pid, int priority)
    { this->pid = pid; this->period = period;}
     bool operator<(const TaskComparable& right) const{

         return (this->period)  < (right.period);
```

[2] Esta es la respuesta al ejercicio 5 a

```
    }

     int get_pid() const { return pid; }              //accessor methods
    int get_period() const { return period; }

  private:
    int pid, period;                                  //data fields
};
```

Cuando una tarea entra en el sistema guardamos la información de su período y de su id de proceso en una instancia de esta clase, que define un operador de menor. Esto lo hacemos para poder encolarla en una `periority_queue` y así mantener los procesos siempre ordenados de acuerdo al request rate (o lo que es lo mismo, darle más prioridad a aquellas tareas que tenga menor request period).

El manejo del fin del quantum es similar al round-robin en la medida en que se tomará el primero de una cola de prioridad y se volverá a encolar el que estaba ejecutando en el caso de que no se bloquee o termine.

La única diferencia es que el proceso se volverá a encolar en la `priority_queue` antes de obtener el siguiente proceso a aejecutar ya que el proceso que se encuentre en RUNNING puede seguir ejecutando si no entró otro proceso de mayor prioridad:

```
    if (m == TICK) {
          if (!q.empty()) {
            int des = current_pid(cpu);
            if (des != IDLE_TASK) {
                TaskComparable tsk;
                tsk = TaskComparable(des, period(des));
                 q.push(tsk); // vuelvo a encolar la tarea que esta ejecutando
            }
            int sig = q.top().get_pid(); q.pop();
            return sig;
            } else {
            return current_pid(cpu);
         }
     }
```

En la sección 7 del paper se presenta lo que se llama el *deadline driven algorithm*[3]. Como dijimos, se trata de un algoritmo de asignación dinámica de prioridades. Al finalizar cada quantum o al quedar libre el procesador, la estrategia de asignación del recurso no se basa en la prioridad dada en la carga de la tarea, sino que se dará una mayor prioridad a aquellos procesos cuya *deadline* esté más próxima y una menor prioridad a aquellos en los que el deadline esté más alejado en el tiempo.

En el teorema 7[4] se da una condición en la que esta asignación dinámica es factible de ser realizada sin overflow. El algoritmo dinámico es factible si las sumas de la relación entre sus runtimes y sus period request es menor o igual a 1. Intuitivamente esto indica que cada tarea tiene, por una parte, un tiempo de ejecución menor a su período, y que las relaciones entre estos dos componentes en todas las tareas hace que el evitar el overflow de un proceso no afecte el proceso de otro.

El algoritmo de asignación dinámica se encuentra implementado en los archivos `sched_dynamic.h` y `sched_dynamic.cpp`. Como en los otros casos, en el header se define contadorQuantums y contadorQuantumsOriginal original, que guarda respectivamente la cantidad de ticks que faltan para que se termine el quantum en cada núcleo y la cantidad de ticks que comprende un quantum en cada uno de los cores.

Por otra parte, tenemos un vector de redimensionable en el que vamos guardando las tareas que acceden al sistema, otro vector paralelo a este guarda la información de si una determinada tarea se encuentra habilitada (ready o running) o inhabilitada (bloqueada o terminada). Finalmente tenemos otro vector que guarda el tiempo faltante de las tareas para alcanzar la deadlone y por lo tanto, entrar en overflow.

```
        std::vector<int> tareas; // guarda las ids de las
```

[3]Ejercicio 5 b
[4]ejercicio 5 c

```
                                //tareas que se estan ejecutando
        std::vector<int> tiempoFaltante; // guarda lo que le falta
                            //a cada tarea para el overflow
        std::vector<int> habilitadas; // con 0 indica que la tarea
                                    // esta bloqueda o termino con 1 en READY o RUNNING
```

En el constructor de las clases, como en el caso de los demás algoritmos con desalojo seteamos los vectors contadorQuantums y contadorQuantumsOriginal.

Cuando se carga una tarea, se agrega a las tareas que pasan (y pasaron por el sistema), se setea como tiempo faltante para el overflow el period completo y se establece que está habilitada (READY):

```
void SchedDynamic::load(int pid) {
    tareas.push_back(pid); // la agrego para indicar que existe esta tarea
    tiempoFaltante.push_back(period(pid)); // guardo el periodo, falta periodo para overflow
    habilitadas.push_back(1); // indico que esta en ready
}
```

Cuando una tarea se desbloquea, pasa a tener valor 1 en el vector de habilitadas:

```
void SchedDynamic::unblock(int pid) {
    for (unsigned int i = 0; i < habilitadas.size(); i++) {
        if (tareas[i] == pid) {
             habilitadas[i] = 1;
        }
    }
}
```

Cuando se produce un tick de reloj, lo que se realiza en primer momento, es reducir en 1 tick el tiempo que falta en cada tarea para el overflow:

```
    for (unsigned int i = 0; i < tareas.size(); i++) {
        if (habilitadas[i] == 1) {
             tiempoFaltante[i] = tiempoFaltante[i]-1;
        }
    }
```

Tanto en el caso de BLOCK, EXIT se deshabilita la tarea. Para elegir cuál es la siguiente tarea a ejecutar, se recorre todas las tareas habilitadas y nos quedamos con aquella a la que le falta menos para el deadline. Para ello implementamos la función *obtenerTareaPrioritaria*:

```
int obtenerTareaPrioritaria(vector<int> tareas, vector<int> tiempoFaltante,
vector<int> habilitadas) {
    int tareaPrioritaria = IDLE_TASK;
    int primeraHabilitada = obtenerPrimeraHabilitada(habilitadas);
    if (primeraHabilitada == -1) {
        return IDLE_TASK;
    }
    int minimoFaltanteOverflow = tiempoFaltante[primeraHabilitada];
   tareaPrioritaria = tareas[primeraHabilitada];
    for (unsigned int i = 0; i < tareas.size(); i++) { // itero por todas las tareas habilitadas
        if ((habilitadas[i] == 1 && tiempoFaltante[i] < minimoFaltanteOverflow) ) {
            tareaPrioritaria = tareas[i];
            minimoFaltanteOverflow = tiempoFaltante[i];
        }
    }
    return tareaPrioritaria;
}
```

El **ejercicio 9** nos pide definir un lote de tareas que no sea factible para el algoritmo de prioridades fijas, pero sí para el de prioridades dinámicas.

Tomemos el siguiete lote:

```
&A1,10,8
@6:
&B1,8,1
&B1,8,1
&B1,8,1
&B1,8,1
```

Una tarea de la familia A se carga en el instante 0 con un período de 10 ticks de reloj y un runtime de 8. En el momento 6 se cargan 4 tareas de la familia 8 con un período de 8 ticks y un runtime de 8. Veamos los resultados para cada uno de los dos algoritmos de *realtime*. Se utiliza un solo core con 1 de penalización por carga y 1 por cambio de procesador (irrelevante):

Figura 8: Dynamic

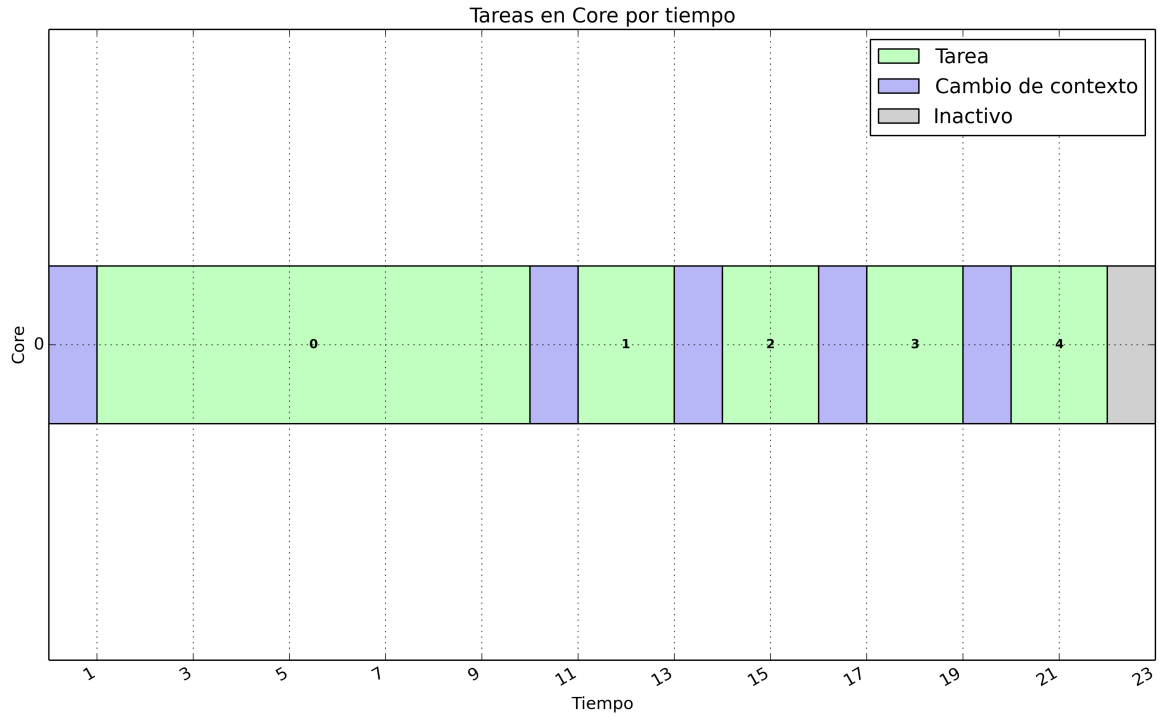
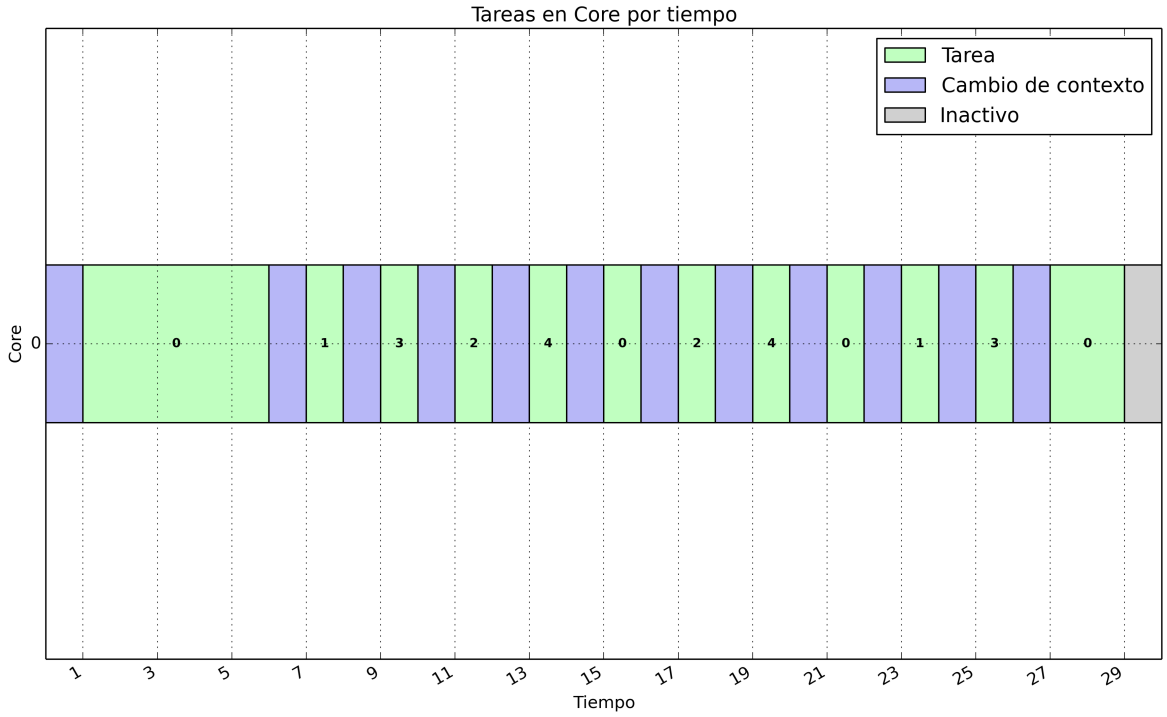


Figura 9: Dynamic



Se observa que en el caso del fixed se llegó a un overflow para la tarea 0. Esto se produjo porque al ingresar las tareas de las familia B en el momento 6, fueron más prioritarias en la medida en que ellas tenían un request rate de 1/8, mientras que el request rate de la tarea de la familia A era de 1/10.

Sin embargo, en el caso de la estrategia dinámica no se produce el overflow porque al ingresar al sistema las tareas de la familia B, la tarea 0 sigue siendo más prioritaria ya que su deadline se encuentra más cercana.

5 Segunda Versión de Round Robin

La segunda versión de round robin impide el pasaje de una tarea de un core a otro, de acuerdo a lo que requiere el **ejercicio 8**. De manera que en la carga del proceso se determina a qué core tiene que estar asignado. El criterio para tomar esta decisión se basa en la cantidad de tareas asociadas a un core: la nueva tarea que entra al sistema irá a aquel core que tenga asignadas menos tareas en estado RUNNING o BLOCKED o READY. Esta segunda versión de *round robin* se encuentra implementada en los archivos `sched_rr2.h` y `sched_rr2.cpp`.

Para implementarla se usó un vector de colas. En el header `sched_rr2.h` se define este vector como un atributo privado.

Por otra parte, tenemos otro atributo privado, que guarda las tareas asignadas a cada core:

```
        std::vector< std::queue<int> > qs; // guarda las colas de prioridades asignadas
                                           //a cada nucleo
        std::vector< std::vector<int> > nucleos; // guarda las tareas
                                                 // asignadas a cada nucleo
```

En el constructor del SchedRR2, inicializamos los vectores. Creamos tantas colas y vectores como cores existan en el ambiente:

```
SchedRR2::SchedRR2(vector<int> argn) {
// Round robin recibe la cantidad de cores y sus cpu_quantum por parámetro
    unsigned int cantidadCores = argn[0];
    for (unsigned int i = 0; i < cantidadCores; i++) {
        queue<int> pids;
        vector<int> procesos;
        qs.push_back(pids);
        nucleos.push_back(procesos);
    }
}
```

Cuando una tarea entra al sistema, se carga y se la asignamos al core que tiene menos tareas año terminadas asociadas. Para ello iteramos por cada de las colas para ver cuál tiene menor cantidad:

```
    unsigned int nucleoMenosCongestionado = 0;
    unsigned int cantidadEnNucleoMenosCongestionado = qs[0].size();
    for (unsigned int i = 0; i < qs.size(); i++) { // se iteran todos los nucleos para ver
                        // cual es el menos congestionado
        if (qs[i].size() < cantidadEnNucleoMenosCongestionado) {
            nucleoMenosCongestionado = i;
            cantidadEnNucleoMenosCongestionado = qs[i].size();
        }
      }

     qs[nucleoMenosCongestionado].push(pid); // se pushea a este nucleo
      nucleos[nucleoMenosCongestionado].push_back(pid); // se lo asocia al nucleo
      //menos congestionado
```

La selección para la siguiente tarea a ejecutarse, es similar a SchedRR, con la diferencia de que se consideran solamente la cola asociada al core que envió la interrupción de reloj. Lo mismo ocurre en el caso del desbloqueo. La tarea se vuelve a asociar a la cola a la que se vinculó en el momento de la carga.

La otra diferencia a tener en cuenta es que cuando una tarea termina, no solamente se desencola sino que se elimina del vector que tiene las tareas asociadas al core en donde culminó su ciclo de vida.

El primer experimento que hicimos consiste en ejecutar el siguiente lote de tareas:

```
TaskCPU 3
TaskCPU 3
TaskCPU 3
TaskCPU 3
TaskCPU 3
```

con 2 cores, tomando como overhead de cambio de core 1, 2, 3 y 4 quantums. El tiempo de cambio de contexto se mantuvo en 1 en cada una de las mediciones. Se hizo este experimento para verificar cómo incidía el cambio del overhead de cambio de core.

Figura 10: SchedRR 5 TaskCPU 2 cores 1, 2, 3, 4 para cambio de core

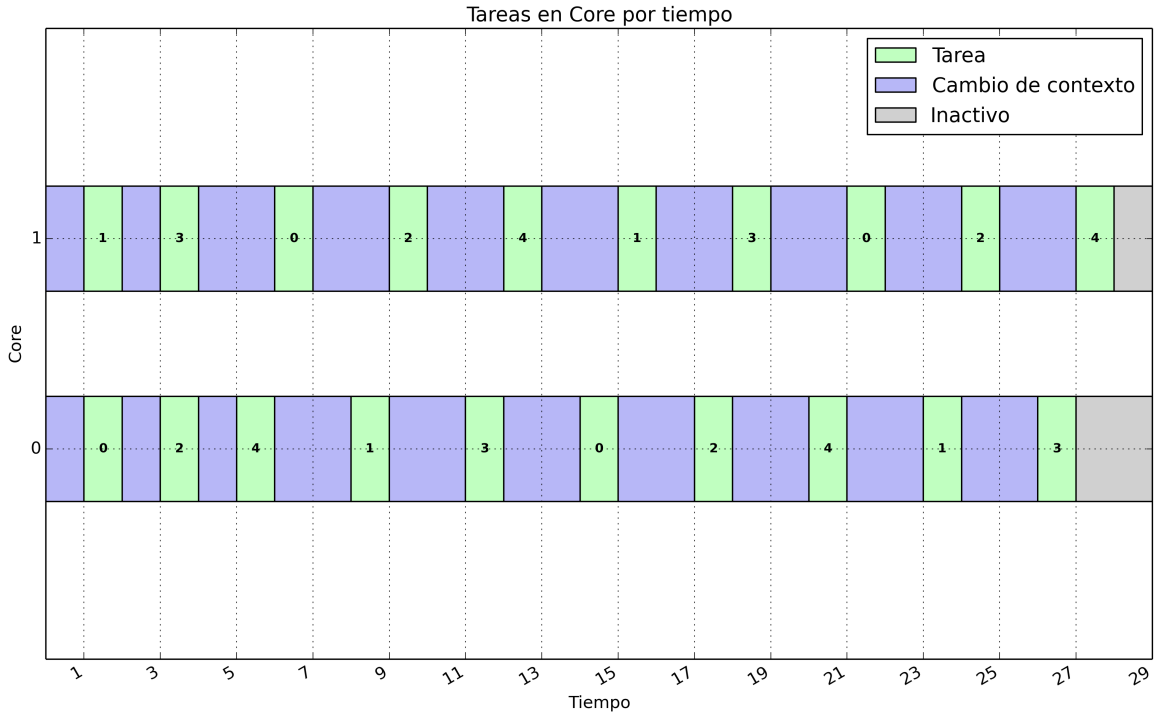


Figura 11: SchedRR2 5 TaskCPU 2 cores 1 para cambio de core

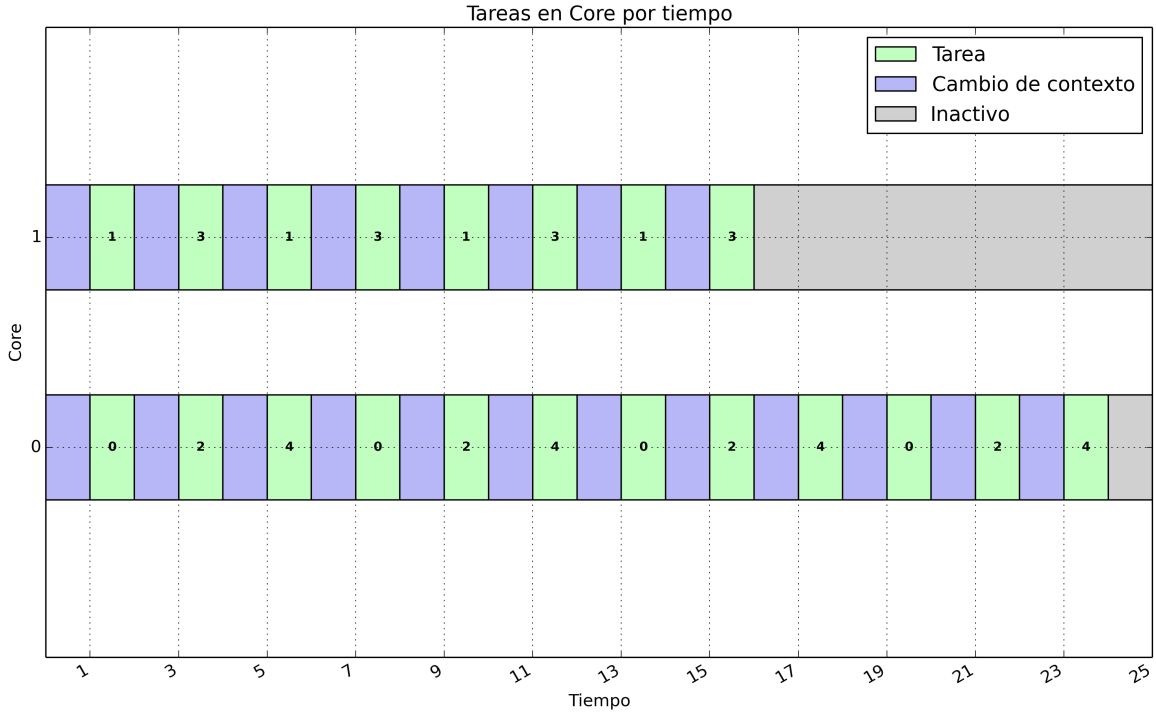


Figura 12: SchedRR2 5 TaskCPU 2 cores 2 para cambio de core

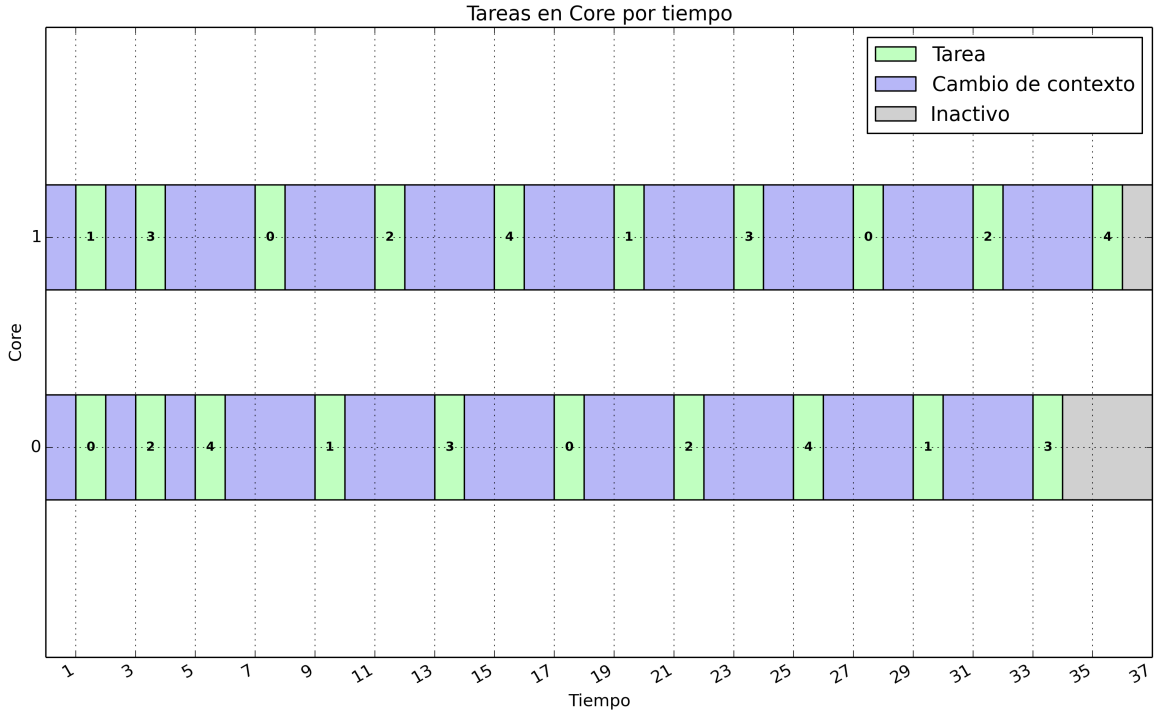
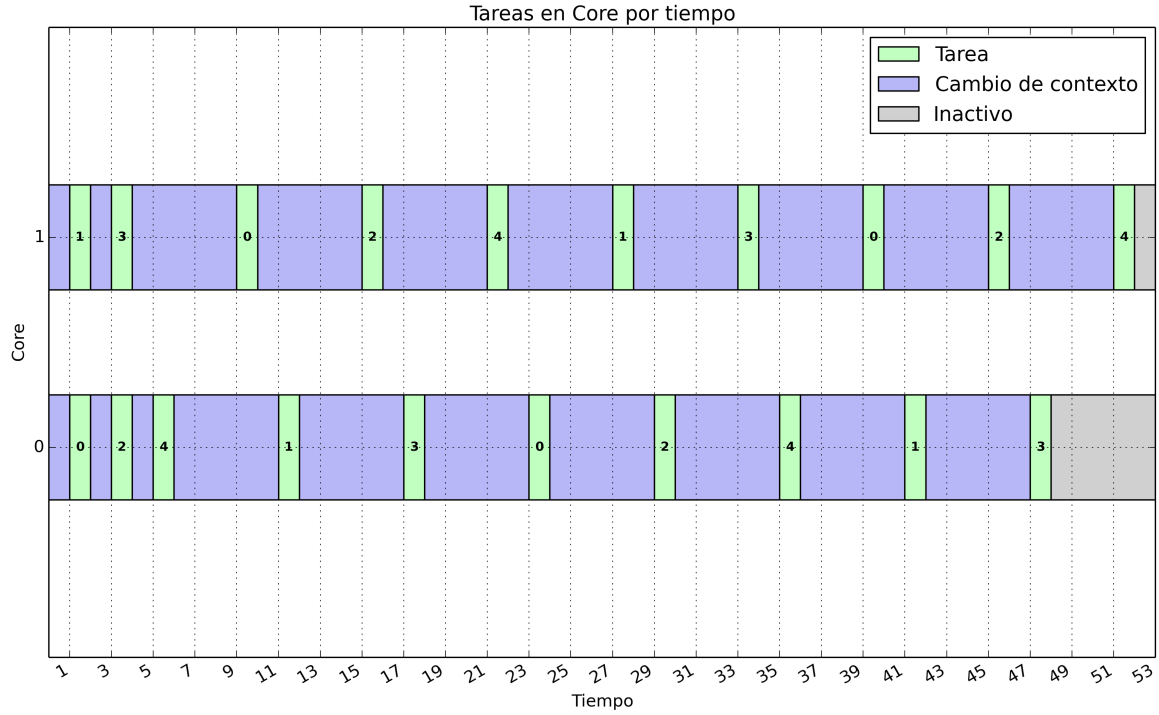


Figura 13: SchedRR2 5 TaskCPU 2 cores 3 para cambio de core



Se observa que el tiempo de turnaround promedio va aumentando en la medida en que aumenta el costo del cambio de core para SchedRR, mientras que en SchedRR2 se mantiene estable, ya que cada una de las tareas se mantiene asociada a una cola.

References

[1] William Stallings, Operating Systems, Prentice Hall, Upper Saddle River 2008.

[2] Andrew S. Tanenbaum, Modern operating systems, Prentice Hall, Upper Saddle River 2001.

[3] C. L. Liu y James W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, Journal of the ACM (JACM), 1973.