Sistemas Operativos
TP1

27 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

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 comunmanete 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 ineractiva 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

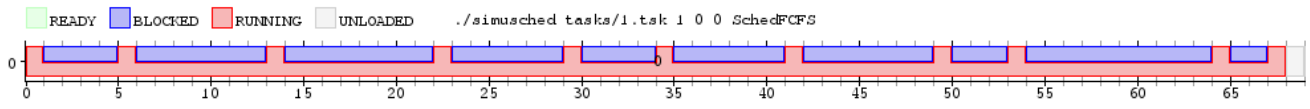

Figura 1: Task Consola

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 *<10,1,10>* como prametro. En consecuencia, y como a su vez el *switch cost* también es nulo, se ejecuta inmediatamente dicha tarea sin tener que mantenerse esperarando en estado *ready*. Como el FCFS, cada vez que una tarea entra en estado *runing*, 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 *runing* durante un ciclo entre cada una para llamar a *uso_IO()*, y finalmente una vez mas para llamar a *exit()* en el último ciclo.

[1] 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

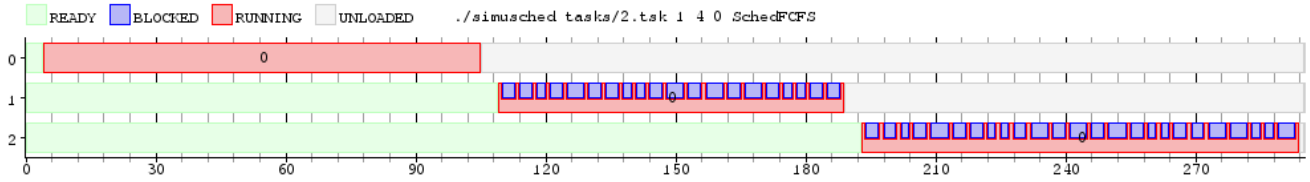

Figura 2: Rolando con 1 núcleo

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 rejoj, 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 *<20, 2, 4>* 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 *<25, 2, 4>* 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

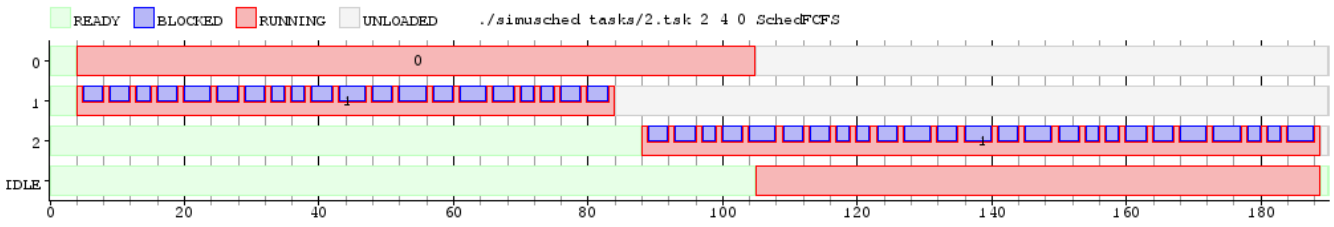

Figura 3: Rolando con 2 núcleos

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

<20, 2, 4>), 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 *<25, 2, 4>*) 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 TaskConsola, 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 bloquedas 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];
9      fill_n(config, total_cpu, false);
10     for (int i = 0; i < cant_bloqueos; i++) config[i] = true;
11     random_shuffle(&config[0], &config[total_cpu-1]);
12
13     for (int i = 0; i < total_cpu; ++i) {
14         if (config[i] == true) { // block
15             uso_IO(pid, 1);
16         } else {
17             uso_CPU(pid, 1);
18         }
19     }
20 }
```

El codigo que simula la tarea `TaskBatch` recibe dos parametros, estos son la cantidad de total de ciclos de reloj que lleva ejecutar la tarea y la cantidad de llamadas bloqueantes que ocurriran durante la ejecucion de la misma. A partir de la cantidad total de ciclos de CPU, se arma un vector en donde van a estar colocados los momentos donde ocurra una llamada bloqueante, estos se colocan de manera pseudoaleatoria con la funcion `random_shuffle`. Una vez que termine este proceso, procedemos a simular la ejecucion de los ciclos de CPU utilizando el vector, si este indica que se debe producir una llamada bloqueante hace una llamada a `uso_IO`, en el caso contrario se ejecuta un ciclo de CPU.

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: 1
4. sched_class: SchedFCFS

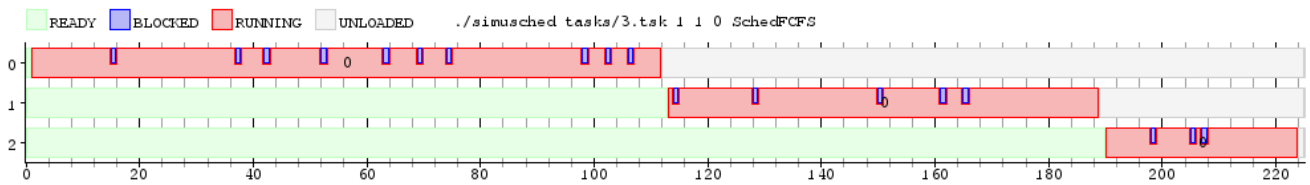

Figura 4: Task Batch

En la figura 4 tenemos el diagrama de Gantt de la ejecucion de tareas del lote 3, las tareas ejecutadas en el mismo cumplen con los siguientes parametros:

- Tarea 0: 100 ciclos de CPU, 10 llamadas bloqueantes
- Tarea 1: 70 ciclos de CPU, 5 llamadas bloqueantes

- Tarea 2: 30 ciclos de CPU, 3 llamadas bloqueantes

Como podemos apreciar, el diagrama cumple perfectamente con el scheduler FCFS, es decir, las tareas no son desalojadas y se las mantiene en ejecucion hasta que concluyan. Ademas podemos ver que el diagrama respeta el costo de hacer una llamada bloqueante, tomando un ciclo adicional antes de efectuar las mismas.

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

```
class SchedRR : public SchedBase {
    public:
        SchedRR(std::vector<int> argn);
        ~SchedRR();
        virtual void load(int pid);
        virtual void unblock(int pid);
        virtual int tick(int cpu, const enum Motivo m);

    private:
        int* quantum;
        int* cycles;
        std::queue<int> q;
};
```

Por un lado, el puntero de enteros llamado `quantum` nos dice por cuantos ciclos de clock nuestras tareas se van a ejecutar por CPU. 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 y Destructor

```
SchedRR::SchedRR(vector<int> argn) {
    // Round robin recibe la cantidad de cores y sus cpu_quantum por parametro
    quantum = new int[argn.size()-1];

    for (int i = 1; i < (int) argn.size(); i++) {
        quantum[i-1] = argn[i];
    }

    cycles = new int[argn[0]];
}
```

Aqui tenemos en `quantum` el arreglo del quantum por CPU, mientras que en `cycles` tenemos un arreglo que mantiene la cantidad de ciclos restantes del quantum de cada CPU.

```
SchedRR::~SchedRR() {
    delete[] cycles;
    delete[] quantum;
}
```

El destructor se encarga de liberar la memoria alocada por los dos arreglos.

4.2. Load y Unblock

```
void SchedRR::load(int pid) {
    q.push(pid);
}

void SchedRR::unblock(int pid) {
    q.push(pid);
}
```

Estas funciones se encargan de cargar tareas y de volverlas a cargar cuando se desbloquean. En este caso la implementacion es simple, ya que la migracion de procesos entre CPUs nos permite tener una unica cola, con lo cual alcanza con agregarlas a la misma para que entren en la rotacion de tareas.

4.2.1. Tick

Aqui tenemos el ciclo de ejecucion del *tick* del scheduler, el mismo se encarga de manejar el desalojo de tareas:

```
1  int SchedRR::tick(int cpu, const enum Motivo m) {
2      if (m == EXIT || m == BLOCK) {
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[cpu];
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[cpu];
14             return sig;
15         } else {
16             cycles[cpu]--;
17
18             if (cycles[cpu] == 0) {
19                 if (q.empty()) {
20                     cycles[cpu] = quantum[cpu];
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[cpu];
26                     return sig;
27                 }
28             } else {
29                 return current_pid(cpu);
30             }
31         }
32     }
33 }
```

Como podemos apreciar, primero se considera si la tarea actual ejecutandose en la CPU termino u ocurrio un bloqueo, en ambos casos se la desaloja y se la cambia por la siguiente tarea en la cola, resetando el quantum en el proceso. Si el procesador se encuantra ejecutando la tarea IDLE y hay tareas esperando a ser ejecutadas, se procede a ejecutar la primera que este en la cola y se resetea el quantum para la CPU. Si no se presentan ninguno de los dos casos anteriores, el scheduler chequea el quantum actual, si aun no termino se mantiene la tarea actual y se concluye el *tick*. En el caso que haya terminado el quantum se chequea la cola de tareas, si no hay tareas en espera se resetea el quantum, en el caso contrario se efectua el cambio de tarea con la primer tarea de la cola tambien reseteando el quantum.

Para verificar la correcta implementacion de los mecanismos de *RoundRobin*, tenemos la siguiente configuracion del scheduler:

1. lote_tsk: 2.tsk
2. num_cores: 2
3. switch_cost: 1
4. sched_class: SchedRR
5. params: 5 10

Esta configuracion fue ejecutada con las siguientes tareas:

- Tarea 0: 40 ciclos de CPU, 0 llamadas bloqueantes
- Tarea 1: 40 ciclos de CPU, 0 llamadas bloqueantes
- Tarea 2: 40 ciclos de CPU, 0 llamadas bloqueantes
- Tarea 3: 20 ciclos de CPU, 5 llamadas bloqueates, incorporada en el momento 10

El resultado obtenido fue el siguiente:

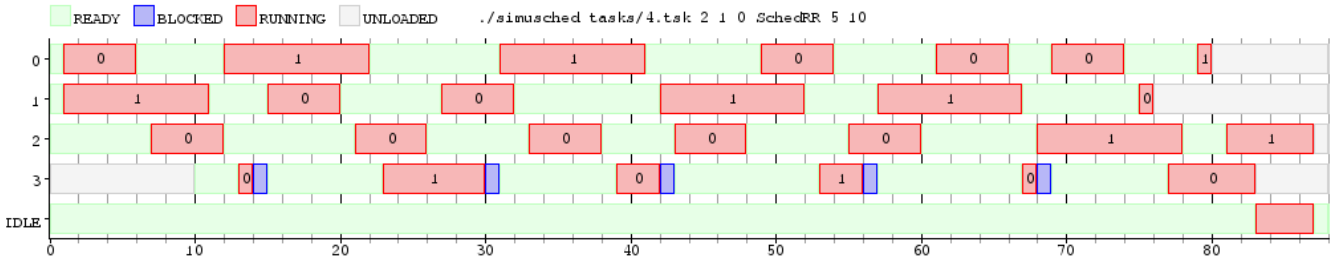

Figura 5: Task Batch

Como podemos apreciar, cada una de las CPU tiene su propio quantum. Las tareas se ejecutan durante el quantum, luego son desalojadas y se procede a la siguiente tarea en la cola. Por ultimo, podemos ver que el comportamiento en los bloqueos es adecuado, cuando una tarea se bloquea esta es desalojada y se procede a buscar la proxima tarea.

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

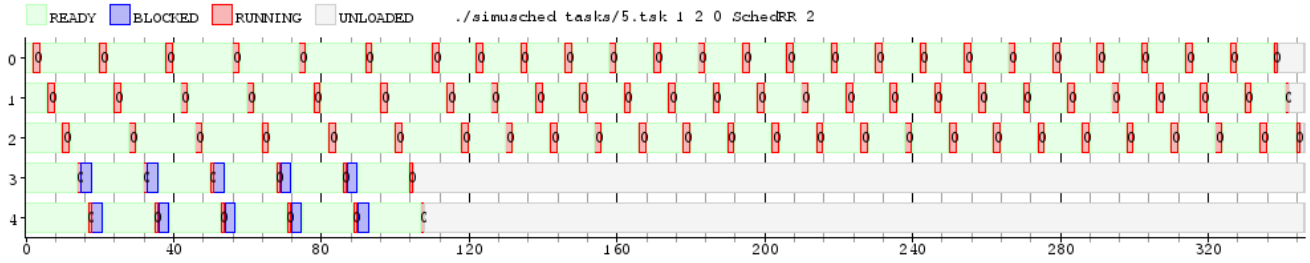

Figura 6: Round-Robin (quantum=2)

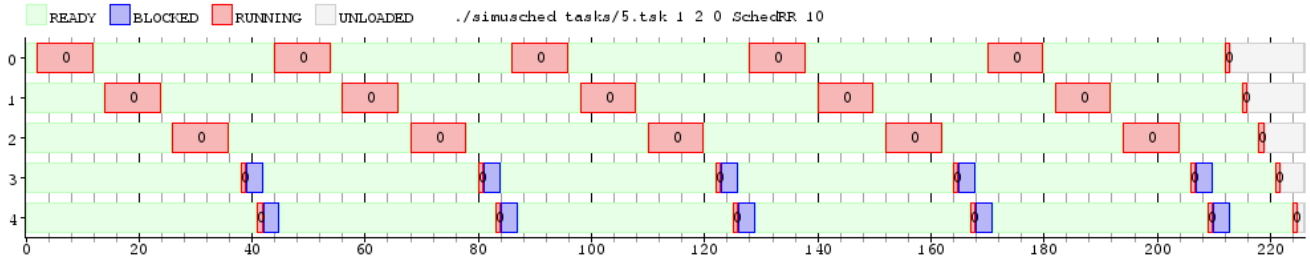

Figura 7: Round-Robin (quantum=10)

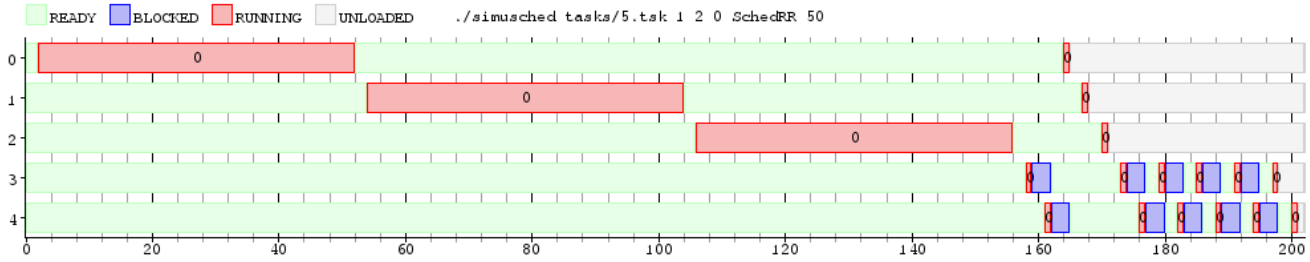

Figura 8: Round-Robin (quantum=50)

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 *runing*. 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 aumenta 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 *runing* 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 permitiendoles permanecer en *runing* sin conmutarlas con otras hasta que estas primeras finalicen; de modo que lidiada con cada tarea una sóla 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 *runing*; finalizado este tiempo, si hay otras tareas en estado *ready* esperando a ser ejecutaras, 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

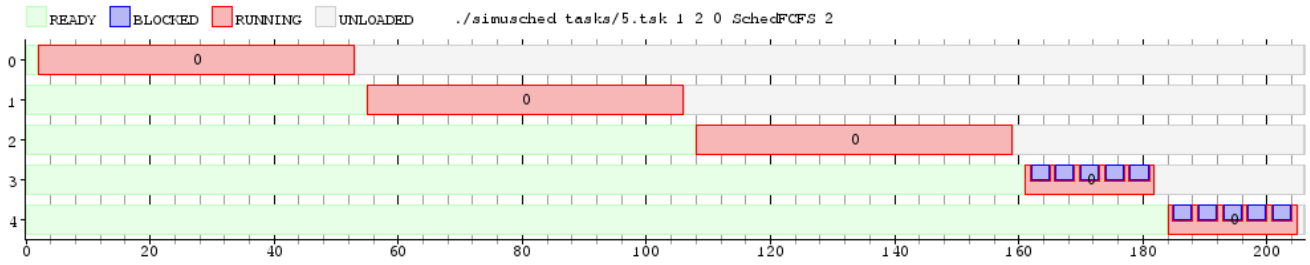

Figura 9: FCFS

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 exihibir 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; dandole 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 mayoria 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. Analisis

Para tener una idea de lo que hace el scheduler probamos con los siguientes parametros:

1. lote_tsk: 2.tsk
2. num_cores: 1
3. switch_cost: 0
4. sched_class: SchedFCFS
5. params: 5 6 7 8 9 10

Primero ejecutamos las siguientes tareas

- Tarea 0: 90 ciclos de CPU, 0 llamadas bloqueantes
- Tarea 1: 90 ciclos de CPU, 0 llamadas bloqueantes
- Tarea 2: 90 ciclos de CPU, 0 llamadas bloqueantes

El resultado fue el siguiente:

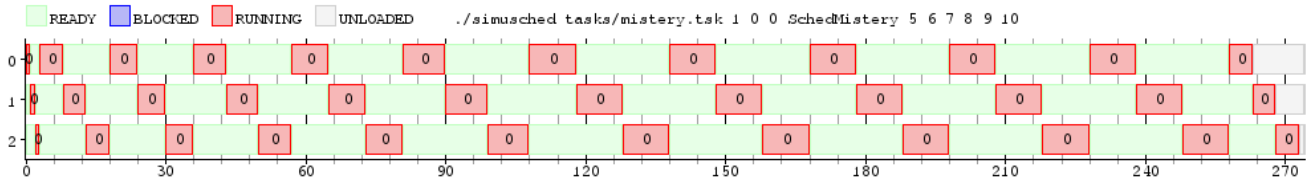

Figura 10: Primera prueba

Aqui podemos apreciar que el scheduler tiene una cierta similitud con *RoundRobin*, donde el quantum varia de manera rotativa segun los parametros pasados al scheduler. La primera vez que el proceso ejecuta lo hace por un ciclo y es desalojado, luego el quantum es determinado por los parametros recibidos hasta llegar al ultimo, a partir de ese momento el quantum es siempre el mismo hasta que el proceso termina de ejecutarse.

Con esto en mente, procedimos a ver como operaba el scheduler con la incorporacion de tareas. Para esto se ejecutaron las siguientes tareas:

- Tarea 0: 90 ciclos de CPU, 2 llamadas bloqueantes, incorporada en el momento 0
- Tarea 1: 90 ciclos de CPU, 4 llamadas bloqueantes, incorporada en el momento 20
- Tarea 2: 90 ciclos de CPU, 6 llamadas bloqueantes, incorporada en el momento 40

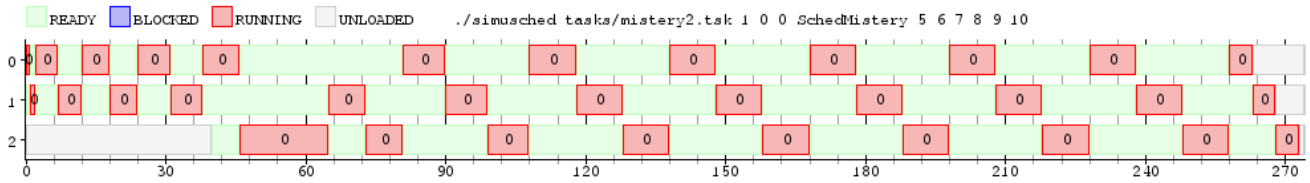

Figura 11: Segunda prueba

En este caso, al incorporar la tarea podemos observar que su quantum asignado es sumamente grande. Luego de mucho análisis observamos que en realidad estaba sumando los quantums que habíamos pasado por parámetro. Aquí es donde nos dimos cuenta que probablemente el Scheduler tenia múltiples colas con diferentes prioridades dadas por los parámetros de entrada. Siempre se ejecuta la tarea de máxima prioridad. La tarea 2 en un inicio se ejecuta por un quantum grande dado que se le suma el quantum de las colas de prioridad 1, 5, 6 y 7.

Finalmente, buscamos ver como funcionaba el scheduler al incorporarse una tarea bloqueante.

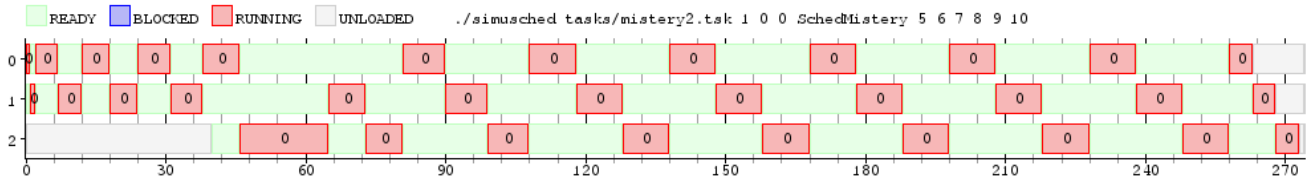

Figura 12: Tercera prueba **PONER LA POSTA**

Notamos que al entrar una tarea bloqueante seguía ejecutando la próxima de la cola y continuaba el Round Robin. Sin embargo, también notamos que al reincorporarse la tarea la misma tenia el quantum correspondiente a una cola con exactamente un nivel mayor de prioridad. De esta manera inferimos que al momento de reincorporar una tarea el scheduler si puede la asigna a una cola con un nivel mas de prioridad. A su vez, esto significa que en muchos casos esta sera la primera en ejecutarse al reincorporarse, sin tener que esperar al resto de las tareas que ya estaban en ejecucion.

7.2. Código

7.2.1. Class Declaration

```
class SchedNoMistery : public SchedBase {
  public:
    SchedNoMistery(std::vector<int> argn);
    virtual void load(int pid);
    virtual void unblock(int pid);
    virtual int tick(int cpu, const enum Motivo m);
  private:
    std::vector<int> quantum_list;
    std::vector<std::queue<int> > q;
  std::map<int, int> blockedQueue;
    int cycles_left, current_queue, tasks;
    int next_pid(void);
};
```

7.2.2. Constructor

```
SchedNoMistery::SchedNoMistery(vector<int> argn) {
    // cpu cores, rest of params
    cycles_left = 1;
    current_queue = 0;
    tasks = 0;

    quantum_list.push_back(1);
    q.push_back(queue<int>());

    if (argn.size() > 1) {
        for (vector<int>::iterator it = ++argn.begin(); it != argn.end(); ++it) {
            quantum_list.push_back(*it);
            q.push_back(queue<int>());
        }
    }
}
```

7.2.3. Load y Unblock

```
void SchedNoMistery::load(int pid) {
    q.at(0).push(pid);
    tasks++;
}
```

```
void SchedNoMistery::unblock(int pid) {
    q.at(max(0,blockedQueue[pid]-1)).push(pid);
    blockedQueue.erase(pid);
    tasks++;
}
```

7.2.4. Tick

```
int SchedNoMistery::tick(int cpu, const enum Motivo m) {
    if (m == EXIT || m == BLOCK) {
        if (m == BLOCK) {
            blockedQueue[current_pid(cpu)] = current_queue;
        }
```

```
6
7           tasks--;
8           // current pid ended, get next
9           if (tasks == 0) return IDLE_TASK;
10          else { // get task from list
11              return next_pid();
12          }
13      } else {
14          if (current_pid(cpu) == IDLE_TASK && tasks > 0) {
15              return next_pid();
16          } else {
17              cycles_left--;
18              if (current_pid(cpu) != IDLE_TASK && cycles_left == 0) {
19
20                  current_queue = min(current_queue + 1, ((int) q.size()) - 1);
21                  if (tasks == 0) {
22                      cycles_left = quantum_list.at(current_queue);
23                      return current_pid(cpu);
24                  } else {
25                      q.at(current_queue).push(current_pid(cpu));
26                      return next_pid();
27                  }
28              }
29              return current_pid(cpu);
30          }
31      }
32  }
33
34  /* Requires some queue not to be empty */
35  int SchedNoMistery::next_pid() {
36      int selectQueue = 0;
37      for (vector<queue<int> >::iterator it = q.begin(); it != q.end(); ++it) {
38          if ((*it).size() > 0) break;
39          selectQueue++;
40      }
41      int pid = q.at(selectQueue).front();
42      q.at(selectQueue).pop();
43      current_queue = selectQueue;
44      cycles_left = quantum_list.at(selectQueue);
45      return pid;
46  }
```

BORRAR LO SIGUIENTE?

7.3. Diagramas y analisis del scheduler

Para poder analizar el comportamiento del scheduler, decidimos armar el siguiente lote de tareas:

```
1       *3 TaskCPU 10
2  @4
3  *3 TaskCPU 5
4
5  TaskConsola 2 1 4
6  TaskConsola 5 1 1
7  TaskConsola 10 1 2
8  *3 TaskCPU 10
9  @20:
10 TaskCPU 10
```

De este gráfico podemos inferir lo siguiente:

- El scheduler es un round robin con algunas particularidades
- Al incoporarse una nueva tarea, se la coloca en el tope de la cola del scheduler.

Este ultimo punto es importante, si dos tareas se incoporan durante la ejecucion de otra tarea, primero concluye el quantum de la tarea actual y se procede a cambiar la tarea con la primera que llego. Es por esta razon que elegimos implementar el scheduler sobre una lista, ya que nos permite agregar los elementos en las posiciones que precisamos.

Otra cosa que determinamos mediante experimentacion, es que el scheduler toma una cantidad arbitraria de parametros. Despues de varias pruebas, pudimos determinar el comportamiento del scheduler respecto a los parametros. Si tomamos los parametros `5 4 3 2 1`, todas las tareas van a ejecutarse por primera vez con un quantum de un ciclo (esto es independiente de los parametros). Posteriormente el scheduler toma los valores del quantum a partir de los parametros del scheduler de forma ordenada y ciclica, es decir, el quantum de una tarea seria `1`, `5`, `4`, `3` ,`2`, `1`, `5`, `4` y asi sucesivamente hasta que concluye la ejecucion de la tarea.

Por ultimo, esta el analisis como responde el scheduler ante las llamadas bloqueantes. Si una tarea tiene una llamada bloqueante se procede a desalojarla hasta que la misma se resuelve, una vez que se resuelve se procede a agregarla nuevamente a la lista con un quantum de un ciclo. Esto ocurre independientemente de los parametros que recibe el scheduler.

8. Round Robin 2

8.1. Codigo

8.1.1. Class Declaration

```
class SchedRR2 : public SchedBase {
    public:
        SchedRR2(std::vector<int> argn);
        ~SchedRR2();
        virtual void load(int pid);
        virtual void unblock(int pid);
        virtual int tick(int cpu, const enum Motivo m);
    private:
        int quantum;
        int* cycles;
        std::vector<std::queue<int> > q;
        std::vector<int> totalLoad; // buscar otra estructura?

        int getCPU();
};
```

8.1.2. Constructor

```
SchedRR2::SchedRR2(vector<int> argn) {
    // Round robin recibe la cantidad de cores y sus cpu_quantum por parametro
    for (int i = 0; i < argn[0]; ++i) {
        q.push_back(queue<int>());
        totalLoad.push_back(0);
    }

    quantum = argn[1];
    cycles = new int[argn[0]];
    fill_n(cycles, argn[0], quantum);
}
```

8.1.3. Desctructor

```
SchedRR2::~SchedRR2() {
    delete[] cycles;
}
```

8.1.4. Load

```
void SchedRR2::load(int pid) {
    int cpu = getCPU();
    q.at(cpu).push(pid);
    totalLoad[cpu]++;
}
```

8.1.5. tick

```
int SchedRR2::tick(int cpu, const enum Motivo m) {
    if (m == EXIT) {
        totalLoad[cpu]--;
        // Si el pid actual termino, sigue el proximo.
```

```
5           if (q.at(cpu).empty()) return IDLE_TASK;
6           else {
7               int sig = q.at(cpu).front(); q.at(cpu).pop();
8               cycles[cpu] = quantum;
9               return sig;
10          }
11      } else {
12          if (current_pid(cpu) == IDLE_TASK && !q.at(cpu).empty()) {
13              int sig = q.at(cpu).front(); q.at(cpu).pop();
14              cycles[cpu] = quantum;
15              return sig;
16          } else {
17              cycles[cpu]--;
18
19              if (cycles[cpu] == 0) {
20                  if (q.at(cpu).empty()) {
21                      cycles[cpu] = quantum;
22                      return current_pid(cpu);
23                  } else {
24                      int sig = q.at(cpu).front(); q.at(cpu).pop();
25                      q.at(cpu).push(current_pid(cpu)); // re-add to queue
26                      cycles[cpu] = quantum;
27                      return sig;
28                  }
29              } else {
30                  return current_pid(cpu);
31              }
32          }
33      }
34  }
```

8.1.6. getCPU

```
1   int SchedRR2::getCPU() {
2       int cpu = 0;
3       int i = 1;
4       for (vector<int>::iterator it = ++totalLoad.begin() ; it != totalLoad.end(); ++it
            ) {
5           if (*it < totalLoad.at(cpu)) {
6               cpu = i;
7           }
8           i++;
9       }
10      return cpu;
11  }
```